Abstraction Logic in Isabelle/HOL

Steven Obua

June 27, 2022

Abstract

This is work in progress. Its ultimate goal is the formalisation in Isabelle/HOL of Abstraction Logic and its properties as described in [3] and [2].

Contents

```
theory General
  imports Main HOL−Library.LaTeXsugar HOL−Library.OptionalSugar
begin
```

1 General

1.1 nats

```
definition nats :: nat ⇒ nat set where
  nats n = {..< n }

lemma finite-nats[iff]: finite (nats n)
  using nats-def by auto

lemma nats-elem[simp]: (d ∈ nats n) = (d < n)
  using nats-def by auto

lemma nats-0[simp]: nats 0 = {}
  by (simp add: nats-def)

lemma card-nats[simp]: card (nats n) = n
  by (simp add: nats-def)

lemma nats-eq-nats[simp]: (nats n = nats m) = (n = m)
  by (metis card-nats)

lemma Max-nats: n > 0 =⇒ 1 + Max (nats n) = n
  by (metis Max-gr-iff Max-in Suc-eq-plus1-left Suc-leI finite-nats lessI
      linorder-neqE-nat nats-0 nats-elem not-le)
```

1.2 Lists

1.2.1 Tools for Indices

```
lemma nats-length-nths:
  assumes A ⊆ nats (length xs)
  shows length (nths xs A) = card A
proof −
  have l1: length (nths xs A) = card {i. i < length xs ∧ i ∈ A}
    using length-nths by force
  have l2: card {i. i < length xs ∧ i ∈ A} = card A
    by (smt (verit, ccfv-SIG) Collect-cong Collect-mem-eq Orderings.order-eq-iff
        assms le-fun-def less-eq-set-def nats-elem subsetI)
  from l1 l2 show ?thesis by presburger
qed

fun index-of :: 'a ⇒ 'a list ⇒ nat option where
  index-of x [] = None
| index-of x (a#as) = (if x = a then Some 0 else
    (case index-of x as of
```

```
    None ⇒ None
  | Some i ⇒ Some (Suc i)))

lemma index-of-head: index-of x (x # xs) = Some 0
  by simp

lemma index-of-exists: x ∈ set xs ⟹ ∃ i. index-of x xs = Some i
proof (induct xs)
  case Nil
  then show ?case by auto
next
  case (Cons a xs)
  then show ?case
  proof (cases a = x)
    case True
    then show ?thesis
      by auto
  next
    case False
    then show ?thesis
      using Cons.hyps Cons.prems by fastforce
  qed
qed

lemma index-of-is-None: index-of x xs = None ⟹ x ∉ set xs
  using index-of-exists by fastforce

lemma index-of-is-Some: index-of x xs = Some i ⟹ i < length xs ∧ xs!i = x
proof(induct xs arbitrary: i)
  case Nil
  then show ?case by auto
next
  case (Cons a as)
  then show ?case
  proof(cases a = x)
    case True
    then have i = 0 using Cons by auto
    then show ?thesis using Cons True by simp
  next
    case False
    then have rec: index-of x (a#as) = (case index-of x as of
              None ⇒ None
            | Some k ⇒ Some (Suc k))
      by auto
    then have ∃ k. index-of x (a#as) = Some (Suc k) ∧ index-of x as = Some k
      by (metis Cons.prems not-None-eq option.simps(4) option.simps(5))
    then obtain k where k:  index-of x (a#as) = Some (Suc k) ∧ index-of x as
= Some k by blast
    then show ?thesis using Cons by force
```

qed
qed

definition *shift-index* :: *nat* ⇒ (*nat* ⇒ *'a*) => (*nat* => *'a*) **where**
shift-index d f x = *f* (*x* + *d*)

lemma *shift-index-0*[*simp*]: *shift-index 0* = *id*
by (*subst fun-eq-iff*, *auto simp add*: *shift-index-def*)

lemma *shift-index-acc-append*[*simp*]:
shift-index d (λ*i acc x*. *acc* @ [*f i x*]) = (λ *i acc x*. *acc* @ [*shift-index d f i x*])
by (*auto simp add*: *shift-index-def*)

lemma *shift-index-gather*:
shift-index d (λ*i acc x*. *g* (*f i x*) *acc*) = (λ*i acc x*. *g* (*shift-index d f i x*) *acc*)
by (*auto simp add*: *shift-index-def*)

lemma *shift-index-applied-twice*[*simp*]:
shift-index a (*shift-index b f*) = *shift-index* (*a*+*b*) *f*
apply (*subst fun-eq-iff*)
apply (*auto simp add*: *shift-index-def*)
by (*metis ab-semigroup-add-class.add-ac*(*1*))

lemma *shift-index-unindexed*[*simp*]: *shift-index d* (λ *i*. *F*) = (λ *i*. *F*)
by (*auto simp add*: *shift-index-def*)

definition *sorted-list* :: *nat set* ⇒ *nat list*
where *sorted-list js* = (*THE l*. *sorted l* ∧ *distinct l* ∧ *set l* = *js*)

lemma *set-sorted-list*: *finite js* ⟹ *set* (*sorted-list js*) = *js*
apply (*simp add*: *sorted-list-def*)
using *finite-sorted-distinct-unique*
by (*smt* (*verit*, *del-insts*) *the-equality*)

lemma *sorted-sorted-list*: *finite js* ⟹ *sorted* (*sorted-list js*)
apply (*simp add*: *sorted-list-def*)
using *finite-sorted-distinct-unique*
by (*smt* (*verit*, *del-insts*) *the-equality*)

lemma *distinct-sorted-list*: *finite js* ⟹ *distinct* (*sorted-list js*)
apply (*simp add*: *sorted-list-def*)
using *finite-sorted-distinct-unique*
by (*smt* (*verit*, *del-insts*) *the-equality*)

lemma *sorted-list-intro*: *sorted l* ∧ *distinct l* ∧ *set l* = *js* ⟹ *sorted-list js* = *l*
by (*meson List.finite-set distinct-sorted-list set-sorted-list sorted-distinct-set-unique*

sorted-sorted-list)

lemma *sorted-list-nats*: *sorted-list* (*nats n*) = [*0* ..< *n*]
  using *atLeast-upt distinct-upt nats-def sorted-list-intro sorted-upt* **by** *presburger*

lemma *no-index-sorted-list*:
  assumes *finite*: *finite js*
  assumes *j*: $j \notin js$
  shows *index-of j* (*sorted-list js*) = *None*
proof −
  {
    fix *i* :: *nat*
    assume *index-of j* (*sorted-list js*) = *Some i*
    then have $j \in js$
      by (*metis index-of-is-Some local.finite nth-mem set-sorted-list*)
  }
  then show *?thesis* **using** *j* **by** *fastforce*
qed

lemma *index-sorted-list*:
  assumes *finite*: *finite js*
  assumes *j*: $j \in js$
  shows ∃ *i*. *index-of j* (*sorted-list js*) = *Some i*
  by (*simp add*: *index-of-exists j local.finite set-sorted-list*)

lemma *upper-bound-index-sorted-list*:
  assumes *finite*: *finite js*
  assumes *j*: $j \in js$
  shows *the* (*index-of j* (*sorted-list js*)) < *card js*
  by (*smt* (*verit*, *best*) *distinct-sorted-list index-of-is-None index-of-is-Some j*
    local.finite option.exhaust-sel set-sorted-list sorted-list-of-set.idem-if-sorted-distinct

      sorted-list-of-set-unique sorted-sorted-list)

1.2.2 Indexed Quantification

definition *list-indexed-forall* :: *'a list* ⇒ (*nat* ⇒ *'a* ⇒ *bool*) ⇒ *bool* **where**
  list-indexed-forall xs f = (∀ *i* < *length xs*. *f i* (*xs* ! *i*))

syntax
  -list-indexed-forall :: *pttrn* ⇒ *'a list* ⇒ *pttrn* ⇒ *bool* ⇒ *bool*
    ((*3*∀ - = -!-./ -) [*1000*, *100*, *1000*, *10*] *10*)

translations
  ∀ *x* = *xs*!*i*. *P* ⇌ *CONST list-indexed-forall xs* (λ *i x*. *P*)

lemma *list-indexed-forall-cong*[*fundef-cong*]:
  assumes *xs* = *ys*
  assumes $\bigwedge i\ x.\ i < length\ ys \Longrightarrow x = ys!i \Longrightarrow P\ i\ x = Q\ i\ x$
  shows (∀ *x* = *xs*!*i*. *P i x*) = (∀ *y* = *ys*!*i*. *Q i y*)
  by (*simp add*: *assms list-indexed-forall-def*)

lemma *size-nth*[*termination-simp*]: $i < length\ ts \Longrightarrow size\ (ts\ !\ i) < Suc\ (size\text{-}list\ size\ ts)$
  by (*meson Suc-n-not-le-n linorder-not-less nth-mem size-list-estimation'*)

lemma *list-indexed-forall-empty*[*simp*]: *list-indexed-forall* [] $f = True$
  by (*simp add*: *list-indexed-forall-def*)

lemma *list-indexed-forall-cons*[*simp*]:
  list-indexed-forall $(x\#xs)\ f = (f\ 0\ x \wedge list\text{-}indexed\text{-}forall\ xs\ (shift\text{-}index\ 1\ f))$
  using *less-Suc-eq-0-disj*
  by (*auto simp add*: *list-indexed-forall-def shift-index-def*)

1.2.3 Indexed Fold

definition *list-indexed-fold* :: $(nat \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b \Rightarrow 'b$ **where**
  list-indexed-fold $f\ xs\ y = fold\ (\lambda\ (i,\ x)\ y.\ f\ i\ x\ y)\ (zip\ [0\ ..< length\ xs]\ xs)\ y$

syntax
  -list-indexed-fold :: $pttrn \Rightarrow 'b \Rightarrow pttrn \Rightarrow 'a\ list \Rightarrow pttrn \Rightarrow 'b \Rightarrow 'b$
    ((*3*§*fold* - =/ -,/ - =/ -!-./ -) [*1000*, *51*, *1000*, *100*, *1000*, *10*] *10*)

translations
  §*fold* $a = a0,\ x = xs!i.\ F \rightleftharpoons CONST\ list\text{-}indexed\text{-}fold\ (\lambda\ i\ x\ a.\ F)\ xs\ a0$

lemma *list-indexed-fold-empty*[*simp*]: *list-indexed-fold* f [] $y = y$
  by (*simp add*: *list-indexed-fold-def*)

lemma *list-indexed-fold-cong*[*fundef-cong*]:
  assumes $xs = ys$
  assumes $\bigwedge i\ a\ x.\ i < length\ ys \Longrightarrow x = ys!i \Longrightarrow F\ i\ a\ x = G\ i\ a\ x$
  shows (§*fold* $a = a0,\ x = xs!i.\ F\ i\ a\ x$) = (§*fold* $a = a0,\ y = ys!i.\ G\ i\ a\ y$)
  apply (*simp add*: *list-indexed-fold-def*)
  apply (*rule fold-cong*)
  using *assms*
  apply *auto*
  by (*metis add.left-neutral assms*(*2*) *in-set-zip nth-upt prod.sel*(*1*) *prod.sel*(*2*))

lemma *list-indexed-fold-eq*:
  assumes $\bigwedge i\ a\ x.\ i < length\ xs \Longrightarrow F\ i\ a\ (xs!i) = G\ i\ a\ (xs!i)$
  shows (§*fold* $a = a0,\ x = xs!i.\ F\ i\ a\ x$) = (§*fold* $a = a0,\ x = xs!i.\ G\ i\ a\ x$)
  by (*metis assms list-indexed-fold-cong*)

lemma *list-unindexed-forall*[*simp*]: $(\forall\ x = xs!i.\ P\ x) = (\forall\ x \in set\ xs.\ P\ x)$
  apply (*auto simp add*: *list-indexed-forall-def*)
  by (*metis in-set-conv-nth*)

lemma *fold-zip-interval-shift*:
  $i + length\ xs = j \Longrightarrow$

fold (λ (*i*, *x*) *a*. *F* (*i* + *d*) *x a*) (*zip* [*i* ..< *j*] *xs*) *a* =
fold (λ (*i*, *x*) *a*. *F i x a*) (*zip* [*i*+*d* ..< *j*+*d*] *xs*) *a*
proof (*induct xs arbitrary*: *i j a*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a xs*)
have *ij*: *i* + *length xs* + *1* = *j*
using *Cons*(*2*)
by *auto*
then have *ij-interval*: [*i*..<*j*] = *i* # [*Suc i* ..< *j*]
by (*simp add*: *upt-conv-Cons*)
from *ij* **have** *ijd-interval*: [*i*+*d*..<*j*+*d*] = (*i*+*d*) # [*Suc* (*i*+*d*) ..< *j*+*d*]
by (*simp add*: *upt-conv-Cons*)
show *?case* **using** *Cons*(*1*) *ij*
by (*auto simp add*: *ij-interval ijd-interval*)
qed

lemma *fold-zip-interval-shift1*:
assumes *i* + *length xs* = *j*
shows *fold* (λ (*i*, *x*) *a*. *F* (*Suc i*) *x a*) (*zip* [*i* ..< *j*] *xs*) *a* =
fold (λ (*i*, *x*) *a*. *F i x a*) (*zip* [*Suc i* ..< *Suc j*] *xs*) *a*
proof −
have *add*: ⋀ *i*. *Suc i* = *i* + *1* **by** *auto*
show *?thesis*
apply ((*subst add*)+)
apply (*rule fold-zip-interval-shift*)
by (*simp add*: *assms*)
qed

lemma *list-indexed-fold-cons*[*simp*]:
(§*fold a* = *a0*, *x* = (*u*#*us*)!*i*. *F i a x*) = (§*fold a* = *F 0 a0 u*, *x* = *us*!*i*. *shift-index 1 F i a x*)
proof (*induct us arbitrary*: *a0*)
case *Nil*
then show *?case*
by (*simp add*: *list-indexed-fold-def*)
next
case (*Cons a us*)
have *interval*: ⋀ *n*. [*0*..<*n*] @ [*n*, *Suc n*] = *0* # [(*Suc 0*) ..< *Suc* (*Suc n*)]
by (*simp add*: *upt-conv-Cons*)
have *app1*: ⋀ *i n*. *i* ≤ *n* ⟹ [*i* ..<*n*] @ [*n*] = [*i* ..< *Suc n*]
by *auto*
have *app2*: ⋀ *i n*. *i* ≤ *n* ⟹ [*i* ..<*n*] @ [*n*, *Suc n*] = [*i* ..< *Suc* (*Suc n*)]
by *auto*
have *empty*: ⋀ *us*. ¬ *Suc 0* ≤ *length us* ⟹ *us* = []
by (*meson Suc-leI length-greater-0-conv*)
from *Cons* **show** *?case*
apply (*auto simp add*: *list-indexed-fold-def interval shift-index-def*)

apply (*simp only*: *app1 app2*)
apply (*subst fold-zip-interval-shift1*)
by (*auto simp add*: *empty*)
qed

lemma *list-unindexed-fold*:
(§*fold a* = *a0*, *x* = *xs*!*i*. *F x a*) = *fold F xs a0*
proof (*induct xs arbitrary*: *a0*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
show *?case*
apply *simp*
apply (*simp add*: *list-indexed-fold-def shift-index-def*)
by (*metis list-indexed-fold-def local.Cons*)
qed

1.2.4 Indexed Map

definition *list-indexed-map* :: (*nat* ⇒ *'a* ⇒ *'b*) ⇒ *'a list* ⇒ *'b list* **where**
list-indexed-map f xs = (§*fold acc* = [], *x* = *xs*!*i*. *acc* @ [*f i x*])

syntax
-list-indexed-map :: *pttrn* ⇒ *'a list* ⇒ *pttrn* ⇒ *'b* ⇒ *'b list*
((*3*§*map* - =/ -!-./ -) [*1000*, *100*, *1000*, *10*] *10*)

translations
§*map x* = *xs*!*i*. *F* ⇌ *CONST list-indexed-map* (λ *i x*. *F*) *xs*

lemma *list-indexed-map-cong*[*fundef-cong*]:
assumes *xs* = *ys*
assumes ⋀*i x*. *i* < *length ys* ⟹ *x* = *ys*!*i* ⟹ *F i x* = *G i x*
shows (§*map x* = *xs*!*i*. *F i x*) = (§*map y* = *ys*!*i*. *G i y*)
apply (*simp add*: *list-indexed-map-def*)
apply (*rule list-indexed-fold-cong*)
by (*auto simp add*: *assms*)

lemma [*9*, *49*] = (§*map x* = [*3* :: *nat*, *7*]!*i*. *x* ∗ *x*)
by (*simp add*: *list-indexed-map-def shift-index-def*)

lemma *list-indexed-map-empty*[*simp*]: *list-indexed-map F* [] = []
by (*simp add*: *list-indexed-map-def*)

lemma *list-indexed-map-append-gen1*: (§*fold acc* = *acc0*, *x* = (*as*@*bs*)!*i*. *acc* @ [*f i x*]) =
(§*fold acc* = (§*fold acc* = *acc0*, *x* = *as*!*i*. *acc* @ [*f i x*]), *x* =
bs!*i*. *acc* @ [*shift-index* (*length as*) *f i x*])
proof (*induct as arbitrary*: *acc0 bs f*)

case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a as*)
show *?case* **by** (*auto*, *subst Cons*, *simp*)
qed

lemma *list-indexed-map-append-gen2*:
(§*fold acc* = *as*@*bs*, x = *xs*!i. *acc* @ [f i x]) =
as @ (§*fold acc* = *bs*, x = *xs*!i. *acc* @ [f i x])
proof (*induct xs arbitrary*: *bs f*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons c cs*)
show *?case* **using** *Cons* **by** *simp*
qed

lemma *list-indexed-map-append*:
(§*map* x = (*as*@*bs*)!i. F i x) = (§*map* x = *as*!i. F i x)@(§*map* x = *bs*!i. *shift-index* (*length as*) F i x)
by (*metis list-indexed-map-def append.right-neutral list-indexed-map-append-gen1*

list-indexed-map-append-gen2)

lemma *list-indexed-map-single*[*simp*]: *list-indexed-map* F [a] = [F *0* a]
by (*simp add*: *list-indexed-map-def*)

lemma *list-indexed-map-cons*: (§*map* x = (a#*as*)!i. F i x) = F *0* a # (§*map* x = *as*!i. *shift-index 1* F i x)
using *list-indexed-map-append*[**where** *as*=[a], *simplified*]
by *force*

lemma *map-cons*: *map* f (a#*as*) = f a # (*map* f *as*)
by *force*

lemma *map-snoc*: *map* f (*as*@[a]) = (*map* f *as*) @ [f a]
by *auto*

lemma *map* (λi. F i ((a # *xs*) ! i)) [*0*..<*length xs*] @ [F (*length xs*) ((a # *xs*) ! *length xs*)] =
map (λi. F i ((a # *xs*) ! i)) [*0*..<*Suc*(*length xs*)]
by *simp*

lemma *map-eq-intro*:
length xs = *length ys* $\Longrightarrow$
($\bigwedge$ i. i < *length xs* $\Longrightarrow$ f (*xs*!i) = g (*ys*!i)) $\Longrightarrow$
map f xs = *map g ys*
by (*simp add*: *list-eq-iff-nth-eq*)

lemma *list-indexed-map-alt*:
  (§*map* x = *xs*!*i*. F i x) = *map* (λ i. F i (*xs*!*i*)) [*0* ..< *length xs*]
proof (*induct xs arbitrary*: F)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  have *m1*: *map* (λ*i*. F i ((*a* # *xs*) ! *i*)) [*0*..<*length xs*] @ [F (*length xs*) ((*a* # *xs*) ! *length xs*)] =
      map (λ*i*. F i ((*a* # *xs*) ! *i*)) [*0*..<*Suc*(*length xs*)]
    by *simp*
  have *m2*: *map* (λ*i*. F (*Suc i*) (*xs* ! *i*)) [*0*..<*length xs*] =
      map (λ *i*. F i ((*a*#*xs*) ! *i*)) [*Suc 0* ..< *Suc*(*length xs*)]
    apply (*rule map-eq-intro*)
    apply *auto*
    using *Suc-le-eq* **apply** *blast*
   by (*metis Suc-le-eq add-Suc comm-monoid-add-class.add-0 not-less-eq-eq not-less-zero*

      nth-Cons-Suc nth-upt upt-Suc-append zero-less-iff-neq-zero)
  have *m3*: F *0* *a* # *map* (λ*i*. F i ((*a* # *xs*) ! *i*)) [*Suc 0*..<*Suc* (*length xs*)] =
      map (λ*i*. F i ((*a* # *xs*) ! *i*)) [*0*..<*Suc* (*length xs*)]
    by (*metis* (*no-types*, *lifting*) *map-cons nth-Cons-0 upt-conv-Cons zero-less-Suc*)
  show *?case*
    apply (*auto simp add*: *list-indexed-map-cons*)
    apply (*subst Cons*)
    apply (*simp add*: *shift-index-def*)
    apply (*subst m1*)
    apply (*subst m2*)
    apply (*subst m3*)
    by *blast*
qed

lemma *list-unindexed-map*: (§*map* x = *xs*!*i*. F x) = *map F xs*
proof (*induct xs*)
  case *Nil*
  then show *?case*
    by (*simp add*: *list-indexed-map-def*)
next
  case (*Cons a xs*)
  show *?case* **by** (*simp add*: *list-indexed-map-cons Cons*)
qed

lemma *list-indexed-map-length*[*simp*]: *length* (§*map* x = *xs*!*i*. F i x) = *length xs*
  by (*simp add*: *list-indexed-map-alt*)

lemma *list-indexed-map-at*[*simp*]: i < *length xs* $\Longrightarrow$ (§*map* x = *xs*!*i*. F i x) ! i = F i (*xs*!*i*)
  by (*simp add*: *list-indexed-map-alt*)

1.2.5 Fold over Indexed Map

lemma *fold-indexed-map*: $(\S fold\ acc = a,\ x = xs!i.\ g\ (F\ i\ x)\ acc) = fold\ g\ (\S map\ x{=}xs!i.\ F\ i\ x)\ a$
proof (*induct xs arbitrary*: *a F*)
  case *Nil*
  show *?case* **by** *simp*
next
  case (*Cons u us*)
  show *?case* **using** *Cons*
    by (*simp add*: *list-indexed-map-cons shift-index-gather*)
qed

lemma *fold-union*: $fold\ (\lambda a\ b.\ b \cup a)\ xs\ a0 = a0 \cup \bigcup\ (set\ xs)$
proof (*induct xs arbitrary*: *a0*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** *auto*
qed

lemma *Un-indexed-nats*: $(\bigcup i{\in}\{0..{<}n{::}nat\}.\ F\ i) = \bigcup\ \{\ F\ i \mid i.\ i < n\ \}$
  by (*auto*, *blast*)

lemma *union-indexed-fold*:
  $(\S fold\ X = X0,\ x = xs!i.\ X \cup F\ i\ x) = X0 \cup \bigcup\ \{\ F\ i\ (xs!i) \mid i.\ i < length\ xs\ \}$
  apply (*subst fold-indexed-map*)
  apply (*subst fold-union*)
  apply (*subst list-indexed-map-alt*)
  by (*simp add*: *Un-indexed-nats*)

lemma *union-unindexed-fold*:
  $(\S fold\ X = X0,\ x = xs!\text{-}.\ X \cup F\ x) = X0 \cup \bigcup\ \{\ F\ x \mid x.\ x \in set\ xs\ \}$
  apply (*subst union-indexed-fold*)
  by (*metis in-set-conv-nth*)

1.3 Other

type-synonym $('a,\ 'b)\ map = 'a \Rightarrow 'b\ option$

definition *map-forced-get* :: $('a,\ 'b)\ map \Rightarrow 'a \Rightarrow 'b$ (**infixl** !! *100*) **where**
  $m\ !!\ x = the\ (m\ x)$

end
theory *Shape* **imports** *General*
begin

2 Shape

2.1 Preshapes

type-synonym *preshape* = (*nat set*) *list*

definition *preshape-alldeps* :: *preshape* ⇒ *nat set* **where**
  preshape-alldeps s = $\bigcup$ {*s* ! *i* | *i*. *i* < *length s*}

definition *wellformed-preshape* :: *preshape* ⇒ *bool* **where**
  wellformed-preshape s = (∃ *m*. *nats m* = *preshape-alldeps s*)

lemma *wellformed-preshape-empty*[*intro*]: *wellformed-preshape* []
  using *nats-def preshape-alldeps-def wellformed-preshape-def* **by** *auto*

2.2 Shapes are Wellformed Preshapes

typedef *shape* = {*s* . *wellformed-preshape s*} **morphisms** *Preshape Shape*
  by *auto*

lemma *wellformed-Preshape*[*iff*]: *wellformed-preshape* (*Preshape s*)
  using *Preshape* **by** *auto*

2.3 Valence and Arity

definition *shape-valence* :: *shape* ⇒ *nat* (§*val*) **where**
  §*val s* = (*THE m*. *nats m* = *preshape-alldeps* (*Preshape s*))

definition *shape-arity* :: *shape* ⇒ *nat* (§*ar*) **where**
  §*ar s* = *length* (*Preshape s*)

lemma *preshape-alldeps*[*intro*]: *wellformed-preshape s* ⟹ ∃ *m*. *nats m* = *preshape-alldeps s*
  using *wellformed-preshape-def* **by** *auto*

lemma *preshape-valence*: *preshape-alldeps* (*Preshape s*) = *nats* (*shape-valence s*)
  by (*metis* (*mono-tags*, *lifting*) *nats-eq-nats shape-valence-def the-equality*
    wellformed-Preshape wellformed-preshape-def)

lemma *empty-deps-Shape-valence*:
  preshape-alldeps s = {} ⟹ (*shape-valence* (*Shape s*) = *0*)
 by (*metis CollectI Shape-inverse nats-0 nats-eq-nats preshape-valence wellformed-preshape-def*)

lemma *nonempty-deps-Shape-valence*:
  assumes *wf*: *wellformed-preshape s*
  assumes *nonemtpy*: *preshape-alldeps s* ≠ {}
  shows *shape-valence* (*Shape s*) = *1* + *Max* (*preshape-alldeps s*)
proof –
  have ∃ *m*. *preshape-alldeps s* = *nats m*
    using *wf wellformed-preshape-def* **by** *blast*

then obtain m **where** *preshape-alldeps s* = *nats m* **by** *blast*
then show *?thesis* **using** *Max-nats wf*
by (*metis CollectI nats-0 nonemtpy preshape-valence Shape-inverse zero-less-iff-neq-zero*)
qed

lemma *Shape-arity*[*intro*]: *wellformed-preshape s* $\Longrightarrow$ *shape-arity* (*Shape s*) = *length s*
by (*simp add*: *Shape-inverse shape-arity-def*)

2.4 Dependencies

definition *shape-deps* :: *shape* $\Rightarrow$ *nat* $\Rightarrow$ *nat set* (**infixl** .♮ *100*)
where *s*.♮*i* = (*Preshape s*) ! *i*

abbreviation *shape-select-deps* :: *shape* $\Rightarrow$ *nat* $\Rightarrow$ (*'a list* $\Rightarrow$ *'a list*) (-.@-'(-') [*100*, *101*, *0*] *100*)
where *s*.@*i*(*xs*) $\equiv$ *nths xs* (*s*.♮*i*)

abbreviation *shape-deps-card* :: *shape* $\Rightarrow$ *nat* $\Rightarrow$ *nat* (**infixl** .# *100*)
where *s*.#*i* $\equiv$ *card*(*s*.♮*i*)

lemma *shape-deps-in-alldeps*:
$i <$ *shape-arity s* $\Longrightarrow$ *shape-deps s i* $\subseteq$ *preshape-alldeps* (*Preshape s*)
using *preshape-alldeps-def shape-arity-def shape-deps-def* **by** *auto*

lemma $i <$ *shape-arity s* $\Longrightarrow$ *shape-deps s i* $\subseteq$ *nats* (*shape-valence s*)
using *preshape-valence shape-deps-in-alldeps* **by** *presburger*

lemma *shape-valence-deps*:
assumes d: $d <$ *shape-valence s*
shows $\exists$ $i <$ *shape-arity s*. $d \in$ *shape-deps s i*
proof −
have d': $d \in$ *preshape-alldeps* (*Preshape s*)
using d *preshape-valence* **by** *auto*
have $\exists$ i. $i <$ *length* (*Preshape s*) $\wedge$ $d \in$ (*Preshape s*) ! *i*
using d' **by** (*auto simp add*: *preshape-alldeps-def*)
then obtain i **where** i: $i <$ *length* (*Preshape s*) $\wedge$ $d \in$ (*Preshape s*) ! *i* **by** *blast*
show *?thesis* **using** i
using *shape-arity-def shape-deps-def* **by** *auto*
qed

lemma *shape-deps-valence*:
assumes i: $i <$ *shape-arity s* $\wedge$ $d \in$ *shape-deps s i*
shows $d <$ *shape-valence s*
by (*metis i nats-elem preshape-valence shape-deps-in-alldeps subsetD*)

lemma *nats-shape-valence-is-union*:
nats (*shape-valence s*) = $\bigcup$ { *shape-deps s i* | *i* . $i <$ *shape-arity s* }
using *preshape-alldeps-def preshape-valence shape-arity-def shape-deps-def* **by**

presburger

lemma *zero-arity-valence*: *shape-arity s* = *0* ⟹ *shape-valence s* = *0*
  by (*metis less-nat-zero-code not-gr-zero shape-valence-deps*)

lemma *zero-valence-deps*: *i* < *shape-arity s* ⟹ *shape-valence s* = *0* ⟹ *shape-deps s i* = {}
  by (*metis all-not-in-conv less-nat-zero-code shape-deps-valence*)

definition *shape-valence-at* :: *shape* ⇒ *nat* ⇒ *nat* **where**
  shape-valence-at s i = *card*(*shape-deps s i*)

2.5 Common Concrete Shapes

2.5.1 *value-shape*

definition *value-shape* :: *shape* **where**
  value-shape = *Shape* []

lemma *value-shape-valence*[*iff*]: *shape-valence* (*value-shape*) = *0*
  by (*simp add*: *value-shape-def empty-deps-Shape-valence preshape-alldeps-def*)

lemma *Preshape-Shape*[*intro*]: *wellformed-preshape s* ⟹ *Preshape* (*Shape s*) = *s*
  by (*auto simp add*: *Shape-inverse*)

lemma *value-Preshape*[*simp*]: *Preshape value-shape* = []
  by (*auto simp add*: *value-shape-def*)

lemma *value-shape-arity*[*simp*]: §*ar value-shape* = *0*
  by (*simp add*: *Shape-arity value-shape-def wellformed-preshape-empty*)

2.5.2 *unop-shape*

definition *unop-shape* :: *shape* **where**
  unop-shape = *Shape* [{}]

lemma *wf-unop-preshape*: *wellformed-preshape* [{}]
  using *preshape-alldeps-def wellformed-preshape-def*
  by *auto*

lemma *unop-Preshape*[*simp*]: *Preshape* (*unop-shape*) = [{}]
  by (*simp add*: *Shape-inverse unop-shape-def wf-unop-preshape*)

lemma *unop-shape-arity*[*simp*]: §*ar unop-shape* = *1*
  by (*simp add*: *Shape-arity unop-shape-def wf-unop-preshape*)

lemma *unop-shape-valence*[*simp*]: §*val unop-shape* = *0*
  by (*simp add*: *empty-deps-Shape-valence preshape-alldeps-def unop-shape-def*)

lemma *unop-shape-deps-0*[*simp*]: *shape-deps unop-shape 0* = {}

by (*simp add*: *zero-valence-deps*)

2.5.3 *binop-shape*

definition *binop-shape* :: *shape* **where**
binop-shape = *Shape* [{}, {}]

lemma *wf-binop-preshape*: *wellformed-preshape* [{}, {}]
using *preshape-alldeps-def wellformed-preshape-def*
by (*metis Sup-insert Union-empty List.set-simps nats-0 preshape-valence set-conv-nth unop-Preshape unop-shape-valence*)

lemma *binop-Preshape*[*simp*]: *Preshape* (*binop-shape*) = [{}, {}]
by (*simp add*: *Shape-inverse binop-shape-def wf-binop-preshape*)

lemma *binop-shape-arity*[*simp*]: §*ar binop-shape* = *Suc* (*Suc 0*)
by (*simp add*: *shape-arity-def*)

lemma *binop-shape-valence*[*simp*]: §*val binop-shape* = *0*
by (*metis Preshape-inverse Sup-insert binop-Preshape empty-deps-Shape-valence list.set*(*2*) *nats-0 preshape-alldeps-def preshape-valence set-conv-nth sup.idem unop-Preshape unop-shape-valence*)

lemma *binop-shape-deps-0*[*simp*]: *binop-shape*.♮*0* = {}
by (*simp add*: *zero-valence-deps*)

lemma *binop-shape-deps-1*[*simp*]: *binop-shape*.♮*1* = {}
by (*simp add*: *zero-valence-deps*)

2.5.4 *operator-shape*

definition *operator-shape* :: *shape* **where**
operator-shape = *Shape* [{*0*}]

lemma *wf-operator-preshape*: *wellformed-preshape* [{*0*}]
by (*auto simp add*: *wellformed-preshape-def preshape-alldeps-def nats-def*)

lemma *operator-Preshape*[*simp*]: *Preshape* (*operator-shape*) = [{*0*}]
by (*simp add*: *Shape-inverse operator-shape-def wf-operator-preshape*)

lemma *operator-shape-arity*[*simp*]: §*ar operator-shape* = *Suc 0*
by (*simp add*: *shape-arity-def*)

lemma *operator-shape-valence*[*simp*]: §*val operator-shape* = *Suc 0*
by (*metis atMost-0 ccpo-Sup-singleton empty-set lessThan-Suc-atMost lessThan-def*

list.set(*2*) *nats-def nats-eq-nats operator-Preshape preshape-alldeps-def preshape-valence*
set-conv-nth)

lemma *operator-shape-deps-0*[*iff*]: *operator-shape*.♮*0* = {*0*}
  by (*simp add*: *shape-deps-def*)

end
theory *Signature* **imports** *Shape*
begin

3 Signature

3.1 Abstractions

datatype *abstraction* = *Abs string*

definition *abstr-true* :: *abstraction* **where** *abstr-true* = *Abs ''true''*
definition *abstr-implies* :: *abstraction* **where** *abstr-implies* = *Abs ''implies''*
definition *abstr-forall* :: *abstraction* **where** *abstr-forall* = *Abs ''forall''*
definition *abstr-false* :: *abstraction* **where** *abstr-false* = *Abs ''false''*

lemma *noteq-abstr-true-implies*[*simp*]: *abstr-true* ≠ *abstr-implies*
  by (*simp add*: *abstr-implies-def abstr-true-def*)

lemma *noteq-abstr-implies-forall*[*simp*]: *abstr-implies* ≠ *abstr-forall*
  by (*simp add*: *abstr-implies-def abstr-forall-def*)

lemma *noteq-abstr-true-forall*[*simp*]: *abstr-true* ≠ *abstr-forall*
  by (*simp add*: *abstr-true-def abstr-forall-def*)

3.2 Signatures

type-synonym *signature* = (*abstraction*, *shape*) *map*

definition *empty-sig* :: *signature* **where**
  empty-sig = (λ *a*. *None*)

definition *has-shape* :: *signature* ⇒ *abstraction* ⇒ *shape* ⇒ *bool* **where**
  has-shape S a shape = (*S a* = *Some shape*)

definition *extends-sig* :: *signature* ⇒ *signature* ⇒ *bool* (**infix** ⪰ *50*) **where**
  extends-sig T S = (∀ *a*. *S a* = *None* ∨ *T a* = *S a*)

lemma *has-shape-extends*: *T* ⪰ *S* ⟹ *has-shape S a s* ⟹ *has-shape T a s*
  by (*metis extends-sig-def has-shape-def option.discI*)

definition *sig-contains* :: *signature* ⇒ *abstraction* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
  sig-contains sig abstr valence arity =
    (*case sig abstr of*
      Some s ⇒ §*val s* = *valence* ∧ §*ar s* = *arity*
    | *None* ⇒ *False*)

lemma *has-shape-sig-contains*: *has-shape sig a s* $\Longrightarrow$ *sig-contains sig a* (§*val s*) (§*ar s*)
by (*simp add*: *has-shape-def sig-contains-def*)

lemma *has-shape-get*: *has-shape sig a s* $\Longrightarrow$ *sig* !! *a* = *s*
by (*simp add*: *has-shape-def map-forced-get-def*)

lemma *extends-sig-contains*: *V* $\succeq$ *U* $\Longrightarrow$ *sig-contains U a val ar* $\Longrightarrow$ *sig-contains V a val ar*
by (*smt* (*verit*) *extends-sig-def option.case-eq-if sig-contains-def*)

3.3 Logic Signatures

definition *deduction-sig* :: *signature* ($\mathfrak{D}$) **where**
$\mathfrak{D}$ = *empty-sig*(
abstr-true := *Some value-shape*,
abstr-implies := *Some binop-shape*,
abstr-forall := *Some operator-shape*)

lemma *deduction-sig-true*[*iff*]: *has-shape deduction-sig abstr-true value-shape*
by (*simp add*: *has-shape-def deduction-sig-def*)

lemma *deduction-sig-implies*[*iff*]: *has-shape* $\mathfrak{D}$ *abstr-implies binop-shape*
by (*simp add*: *has-shape-def deduction-sig-def*)

lemma *deduction-sig-forall*[*iff*]: *has-shape* $\mathfrak{D}$ *abstr-forall operator-shape*
by (*simp add*: *has-shape-def deduction-sig-def*)

lemma *deduction-sig-contains-true*[*iff*]: *sig-contains* $\mathfrak{D}$ *abstr-true 0 0*
by (*simp add*: *deduction-sig-def sig-contains-def*)

lemma *deduction-sig-contains-implies*[*iff*]: *sig-contains* $\mathfrak{D}$ *abstr-implies 0* (*Suc* (*Suc 0*))
by (*simp add*: *deduction-sig-def sig-contains-def*)

lemma *deduction-sig-contains-forall*[*iff*]: *sig-contains* $\mathfrak{D}$ *abstr-forall* (*Suc 0*) (*Suc 0*)
using *has-shape-sig-contains deduction-sig-forall* **by** *fastforce*

end
theory *Quotients* **imports** *Main*
begin

4 Quotient

4.1 Quotients

We define a *quotient* to be a set with custom equality. In fact, we identify the set with the custom equivalence relation. We can do this because the set is uniquely determined by the equivalence relation.

Our approach does not replace *HOL.Equiv-Relations*, but builds on top of it by encoding as a type invariant the property of a relation to be an equivalence relation.

typedef *'a quotient* = { *r*::*'a rel*. ∃ *A*. *equiv A r* } **morphisms** *Rel Quotient*
  by (*metis empty-iff equivI mem-Collect-eq refl-on-def subsetI sym-def trans-def*)

definition *QField* :: *'a quotient* ⇒ *'a set* **where**
  QField q = *Field* (*Rel q*)

lemma *equiv-Field*:
  assumes *equiv A r*
  shows *Field r* = *A*
proof −
  have *A* ⊆ *Field r*
    by (*meson FieldI2 assms equivE refl-onD subsetI*)
  moreover have *Field r* ⊆ *A*
    by (*metis Field-square assms equiv-type mono-Field*)
  ultimately show *Field r* = *A*
    by *blast*
qed

lemma *equiv-QField-Rel*: *equiv* (*QField q*) (*Rel q*)
  by (*smt* (*verit*, *ccfv-SIG*) *Rel equiv-Field mem-Collect-eq QField-def*)

definition *qin* :: *'a* ⇒ *'a quotient* ⇒ *bool* (**infix** *'/∈ 50*) **where**
  (*a* /∈ *q*) = (*a* ∈ *QField q*)

abbreviation *qnin* :: *'a* ⇒ *'a quotient* ⇒ *bool* (**infix** *'/∉ 50*) **where**
  (*a* /∉ *q*) ≡ (*a* ∈ *QField q*)

4.2 Equality Modulo

definition *qequals* :: *'a* ⇒ *'a* ⇒ *'a quotient* ⇒ *bool* (- = - *'*(*mod* -*'*) [*51*, *51*, *0*] *50*) **where**
  (*a* = *b* (*mod q*)) = ((*a*, *b*) ∈ *Rel q*)

abbreviation *qnequals* :: *'a* ⇒ *'a* ⇒ *'a quotient* ⇒ *bool* (- ≠ - *'*(*mod* -*'*) [*51*, *51*, *0*] *50*) **where**
  (*a* ≠ *b* (*mod q*)) ≡ ¬ (*a* = *b* (*mod q*))

lemma *qin-mod*: (*a* /∈ *q*) = (*a* = *a* (*mod q*))

by (*metis equiv-QField-Rel equiv-class-eq-iff qequals-def qin-def*)

lemma *qequals-in*: $a = b\ (mod\ q) \Longrightarrow a\ /\!\in q \wedge b\ /\!\in q$
by (*metis FieldI1 FieldI2 QField-def qequals-def qin-def*)

lemma *qequals-sym*: $a = b\ (mod\ q) \Longrightarrow b = a\ (mod\ q)$
by (*meson equiv-QField-Rel equiv-def qequals-def sym-def*)

lemma *qequals-trans*: $a = b\ (mod\ q) \Longrightarrow b = c\ (mod\ q) \Longrightarrow a = c\ (mod\ q)$
by (*meson equiv-QField-Rel equiv-def qequals-def trans-def*)

4.3 Subsets of Quotients

There isn't a unique definition of what a subset of quotients is. There are at least 3 different notions that all make sense.

definition *qsubset-weak* :: $'a\ quotient \Rightarrow 'a\ quotient \Rightarrow bool$ (**infix** $'/\!\le$ *50*) **where**
$(p\ /\!\le q) = (\forall\ x\ y.\ x = y\ (mod\ p) \longrightarrow x = y\ (mod\ q))$

definition *qsubset-bishop* :: $'a\ quotient \Rightarrow 'a\ quotient \Rightarrow bool$ (**infix** $'/\!\sqsubseteq$ *50*) **where**
$(p\ /\!\sqsubseteq q) = (\forall\ x\ y.\ x\ /\!\in p \wedge y\ /\!\in p \longrightarrow (x = y\ (mod\ p) \longleftrightarrow x = y\ (mod\ q)))$

definition *qsubset-strong* :: $'a\ quotient \Rightarrow 'a\ quotient \Rightarrow bool$ (**infix** $'/\!\subseteq$ *50*) **where**
$(p\ /\!\subseteq q) = (\forall\ x\ y.\ x\ /\!\in p \longrightarrow (x = y\ (mod\ p) \longleftrightarrow x = y\ (mod\ q)))$

lemma *qsubset-strong-implies-bishop*: $p\ /\!\subseteq q \Longrightarrow p\ /\!\sqsubseteq q$
by (*simp add*: *qsubset-bishop-def qsubset-strong-def*)

lemma *qsubset-strong-implies-weak*: $p\ /\!\subseteq q \Longrightarrow p\ /\!\le q$
by (*meson qequals-in qsubset-strong-def qsubset-weak-def*)

lemma *qsubset-bishop-implies-weak*: $p\ /\!\sqsubseteq q \Longrightarrow p\ /\!\le q$
by (*meson qequals-in qsubset-bishop-def qsubset-weak-def*)

lemma *qsubset-QField-strong*: $p\ /\!\subseteq q \Longrightarrow QField\ p \subseteq QField\ q$
by (*meson qin-def qin-mod qsubset-strong-def subset-iff*)

lemma *qsubset-QField-weak*: $p\ /\!\le q \Longrightarrow QField\ p \subseteq QField\ q$
by (*meson qin-def qin-mod qsubset-weak-def subset-iff*)

lemma *qsubset-QField-bishop*: $p\ /\!\sqsubseteq q \Longrightarrow QField\ p \subseteq QField\ q$
using *qsubset-QField-weak qsubset-bishop-implies-weak* **by** *blast*

lemma *qubseteq-refl-strong*[*iff*]: $q\ /\!\subseteq q$
using *qsubset-strong-def* **by** *blast*

lemma *qubseteq-refl-bishop*[*iff*]: $q\ /\!\sqsubseteq q$
using *qsubset-bishop-def* **by** *blast*

lemma *qubseteq-refl-weak*[*iff*]: $q\ /\!\le q$

using *qsubset-weak-def* **by** *auto*

lemma *qsubset-trans-strong*: $p \mathbin{/\subseteq} q \Longrightarrow q \mathbin{/\subseteq} r \Longrightarrow p \mathbin{/\subseteq} r$
by (*meson qin-mod qsubset-strong-def*)

lemma *qsubset-trans-bishop*: $p \mathbin{/\sqsubseteq} q \Longrightarrow q \mathbin{/\sqsubseteq} r \Longrightarrow p \mathbin{/\sqsubseteq} r$
by (*smt* (*verit*, *best*) *qin-mod qsubset-bishop-def*)

lemma *qsubset-trans-weak*: $p \mathbin{/\le} q \Longrightarrow q \mathbin{/\le} r \Longrightarrow p \mathbin{/\le} r$
by (*simp add*: *qsubset-weak-def*)

lemma *qsubset-antisym-weak*: $p \mathbin{/\le} q \Longrightarrow q \mathbin{/\le} p \Longrightarrow p = q$
by (*smt* (*verit*) *Rel-inverse dual-order.refl qequals-def qsubset-weak-def subsetI subset-antisym subset-iff surj-pair sym-def*)

lemma *qsubset-antisym-bishop*: $p \mathbin{/\sqsubseteq} q \Longrightarrow q \mathbin{/\sqsubseteq} p \Longrightarrow p = q$
by (*simp add*: *qsubset-antisym-weak qsubset-bishop-implies-weak*)

lemma *qsubset-antisym-strong*: $p \mathbin{/\subseteq} q \Longrightarrow q \mathbin{/\subseteq} p \Longrightarrow p = q$
by (*simp add*: *qsubset-antisym-bishop qsubset-strong-implies-bishop*)

lemma *qsubset-mod-weak*: $x = y\ (mod\ q) \Longrightarrow q \mathbin{/\le} p \Longrightarrow x = y\ (mod\ p)$
by (*simp add*: *qsubset-weak-def*)

lemma *qsubset-mod-bishop*: $x = y\ (mod\ q) \Longrightarrow q \mathbin{/\sqsubseteq} p \Longrightarrow x = y\ (mod\ p)$
by (*metis qsubset-bishop-implies-weak qsubset-mod-weak*)

lemma *qsubset-mod-strong*: $x = y\ (mod\ q) \Longrightarrow q \mathbin{/\subseteq} p \Longrightarrow x = y\ (mod\ p)$
by (*meson qsubset-mod-bishop qsubset-strong-implies-bishop*)

4.4 Equivalence Classes

definition *qclass* :: $'a \Rightarrow 'a\ quotient \Rightarrow 'a\ set$ (**infix** $'/\%$ *80*) **where**
$a \mathbin{/\%} q = (Rel\ q)``\{a\}$

lemma *qequals-implies-equal-qclasses*: $a = b\ (mod\ q) \Longrightarrow a \mathbin{/\%} q = b \mathbin{/\%} q$
by (*metis equiv-QField-Rel equiv-class-eq-iff qclass-def qequals-def*)

lemma *empty-qclass*: $(a \mathbin{/\%} q = \{\}) = (\neg\ (a \mathbin{/\in} q))$
by (*metis Image-singleton-iff ex-in-conv qclass-def qequals-def qequals-in qin-mod*)

lemma *qclass-elems*: $(b \in a \mathbin{/\%} q) = (a = b\ (mod\ q))$
by (*simp add*: *qclass-def qequals-def*)

4.5 Construction via Symmetric and Transitive Predicate

definition *QuotientP* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ quotient$ **where**
$QuotientP\ eq = Quotient\ \{\ (x,\ y)\ .\ eq\ x\ y\ \}$

lemma *QuotientP-eq-refl*: $symp\ eq \Longrightarrow transp\ eq \Longrightarrow eq\ x\ y \Longrightarrow eq\ x\ x \wedge eq\ y\ y$

by (*meson sympD transpE*)

lemma *QuotientP-equiv*:
assumes *symp eq*
assumes *transp eq*
shows *equiv* { *x* . *eq x x*} { (*x*, *y*) . *eq x y* }
proof −
let *?A* = { *x* . *eq x x*}
let *?r* = { (*x*, *y*) . *eq x y* }
have *?r* $\subseteq$ *?A* $\times$ *?A* **using** *QuotientP-eq-refl assms* **by** *fastforce*
then show *equiv ?A ?r*
by (*smt* (*verit*, *best*) *CollectD CollectI Sigma-cong assms case-prodD case-prodI equivI*
refl-onI sym-def symp-def trans-def transp-def)
qed

lemma *QuotientP-Rel*: *symp eq* $\Longrightarrow$ *transp eq* $\Longrightarrow$ *Rel* (*QuotientP eq*) = { (*x*, *y*) . *eq x y* }
by (*metis* (*no-types*, *lifting*) *CollectI QuotientP-def QuotientP-equiv Quotient-inverse*)

lemma *QuotientP-mod*: *symp eq* $\Longrightarrow$ *transp eq* $\Longrightarrow$ (*x* = *y* (*mod QuotientP eq*)) = (*eq x y*)
by (*metis CollectD CollectI QuotientP-Rel case-prodD case-prodI qequals-def*)

lemma *QuotientP-in*: *symp eq* $\Longrightarrow$ *transp eq* $\Longrightarrow$ (*x* /$\in$ *QuotientP eq*) = *eq x x*
by (*simp add*: *QuotientP-mod qin-mod*)

4.6 Set with Identity as Quotient

definition *qequal-set* :: *'a set* $\Rightarrow$ *'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool* **where**
qequal-set U x y = (*x* $\in$ *U* $\wedge$ *x* = *y*)

lemma *qequal-set-sym*: *symp* (*qequal-set U*)
by (*simp add*: *qequal-set-def symp-def*)

lemma *qequal-set-trans*: *transp* (*qequal-set I*)
by (*simp add*: *qequal-set-def transp-def*)

definition *set-quotient* :: *'a set* $\Rightarrow$ *'a quotient* ('/$\equiv$) **where**
/$\equiv$*U* = *QuotientP* (*qequal-set U*)

lemma *set-quotient-Rel*: *Rel*(/$\equiv$*U*) = { (*x*, *y*) . *x* $\in$ *U* $\wedge$ *x* = *y* }
by (*simp add*: *QuotientP-Rel qequal-set-def qequal-set-sym qequal-set-trans set-quotient-def*)

lemma *set-quotient-mod*: (*x* = *y* (*mod* /$\equiv$*U*)) = (*x* $\in$ *U* $\wedge$ *x* = *y*)
by (*simp add*: *QuotientP-mod qequal-set-def qequal-set-sym qequal-set-trans set-quotient-def*)

lemma *set-quotient-in*: (*x* /$\in$ /$\equiv$*U*) = (*x* $\in$ *U*)
by (*simp add*: *qin-mod set-quotient-mod*)

lemma *set-quotient-subset-strong*: $(/\equiv U \ /\sqsubseteq \ /\equiv V) = (U \subseteq V)$
by (*smt* (*verit*, *ccfv-SIG*) *qsubset-strong-def set-quotient-in set-quotient-mod subsetD subsetI*)

lemma *set-quotient-subset-weak*: $(/\equiv U \ /\leq \ /\equiv V) = (U \subseteq V)$
by (*meson qsubset-mod-weak qsubset-strong-implies-weak set-quotient-mod set-quotient-subset-strong subset-iff*)

lemma *set-quotient-subset-bishop*: $(/\equiv U \ /\sqsubseteq \ /\equiv V) = (U \subseteq V)$
by (*meson qsubset-bishop-implies-weak qsubset-strong-implies-bishop set-quotient-subset-strong set-quotient-subset-weak*)

4.7 Empty and Universal Quotients

definition *empty-quotient* :: *'a quotient* $('/\emptyset)$ **where**
$/\emptyset = /\equiv\{\}$

definition *univ-quotient* :: *'a quotient* $('/\mathcal{U})$ **where**
$/\mathcal{U} = /\equiv UNIV$

lemma *empty-quotient-Rel*: $Rel\ /\emptyset = \{\}$
by (*simp add*: *empty-quotient-def set-quotient-Rel*)

lemma *empty-quotient-mod*: $\neg\ (x = y\ (mod\ /\emptyset))$
by (*simp add*: *empty-quotient-def set-quotient-mod*)

lemma *empty-quotient-in*: $\neg\ (x\ /\in\ /\emptyset)$
by (*simp add*: *empty-quotient-def set-quotient-in*)

lemma *univ-quotient-Rel*: $Rel\ /\mathcal{U} = Id$
by (*auto simp add*: *set-quotient-Rel univ-quotient-def*)

lemma *univ-quotient-in*: $x\ /\in\ /\mathcal{U}$
by (*simp add*: *set-quotient-in univ-quotient-def*)

lemma *univ-quotient-mod*: $(x = y\ (mod\ /\mathcal{U})) = (x = y)$
by (*simp add*: *set-quotient-mod univ-quotient-def*)

4.8 Singleton Quotients

definition *qequal-singleton* :: *'a set* $\Rightarrow$ *'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool* **where**
qequal-singleton $U\ x\ y = (x \in U \wedge y \in U)$

lemma *qequal-singleton-sym*: *qequal-singleton* $U\ x\ y \Longrightarrow$ *qequal-singleton* $U\ y\ x$
by (*simp add*: *qequal-singleton-def*)

lemma *qequal-singleton-trans*:
qequal-singleton $U\ x\ y \Longrightarrow$ *qequal-singleton* $U\ y\ z \Longrightarrow$ *qequal-singleton* $U\ x\ z$
by (*simp add*: *qequal-singleton-def*)

definition *singleton-quotient* :: $'a\ set \Rightarrow 'a\ quotient$ $('/\mathbf{1})$ **where**
$/\mathbf{1}\,U = QuotientP\ (qequal\text{-}singleton\ U)$

lemma *singleton-quotient-Rel*: $Rel\ (/\mathbf{1}\,U) = \{\ (x,\ y).\ qequal\text{-}singleton\ U\ x\ y\ \}$
by (*metis QuotientP-Rel qequal-singleton-def singleton-quotient-def sympI transpI*)

lemma *singleton-quotient-mod*[*simp*]: $(x = y\ (mod\ /\mathbf{1}\,U)) = (x \in U \wedge y \in U)$
by (*simp add*: *qequal-singleton-def qequals-def singleton-quotient-Rel*)

lemma *singleton-quotient-in*: $(x\ /\!\in\ /\mathbf{1}\,U) = (x \in U)$
by (*meson qin-mod singleton-quotient-mod*)

lemma *empty-singleton-quotient*[*iff*]: $/\mathbf{1}\{\} = /\emptyset$
by (*simp add*: *empty-quotient-in qsubset-antisym-strong qsubset-strong-def singleton-quotient-in*)

abbreviation *universal-singleton-quotient*:: $'a\ quotient$ $('/\mathbf{1}\mathcal{U})$ **where**
$/\mathbf{1}\mathcal{U} \equiv /\mathbf{1}\,UNIV$

4.9 Comparing Notions of Quotient Subsets

lemma *empty-subset-singleton-quotient-weak*: $/\emptyset\ /\!\leq\ q$
by (*simp add*: *empty-quotient-mod qsubset-weak-def*)

lemma *empty-subset-singleton-quotient-bishop*: $/\emptyset\ /\!\sqsubseteq\ q$
by (*simp add*: *empty-quotient-in qsubset-bishop-def*)

lemma *empty-subset-singleton-quotient-strong*: $/\emptyset\ /\!\subseteq\ q$
by (*simp add*: *empty-quotient-in qsubset-strong-def*)

lemma *same-QField-bishop*: $QField\ p = QField\ q \Longrightarrow p\ /\!\sqsubseteq\ q \Longrightarrow p = q$
by (*simp add*: *qin-def qsubset-antisym-bishop qsubset-bishop-def*)

lemma *same-QField-strong*: $QField\ p = QField\ q \Longrightarrow p\ /\!\subseteq\ q \Longrightarrow p = q$
by (*simp add*: *qsubset-strong-implies-bishop same-QField-bishop*)

lemma *singleton-quotient-subset-weak*: $(/\mathbf{1}\,U\ /\!\leq\ /\mathbf{1}\,V) = (U \subseteq V)$
by (*meson qsubset-weak-def singleton-quotient-mod subset-iff*)

lemma *singleton-quotient-subset-bishop*: $(/\mathbf{1}\,U\ /\!\sqsubseteq\ /\mathbf{1}\,V) = (U \subseteq V)$
by (*meson qsubset-bishop-def qsubset-bishop-implies-weak singleton-quotient-in singleton-quotient-mod singleton-quotient-subset-weak subsetD*)

lemma *singleton-quotient-subset-strong*: $(/\mathbf{1}\,U\ /\!\subseteq\ /\mathbf{1}\,V) = (U = V \vee U = \{\})$
proof –
have *implies-sub*:$(/\mathbf{1}\,U\ /\!\subseteq\ /\mathbf{1}\,V) \Longrightarrow (U \subseteq V)$
by (*metis qsubset-strong-implies-weak singleton-quotient-subset-weak*)

{
  assume *singleton-UV*: $(/\mathbf{1}\,U\ /\subseteq\ /\mathbf{1}\,V)$
  have *UV*: $U \subseteq V$
    using *implies-sub singleton-UV* **by** *auto*
  fix *x y*
  assume *x*: $x \in V$
  assume *notx*: $x \notin U$
  assume *y*: $y \in U$
  have *xV*: $x\ /\!\in\ /\mathbf{1}\,V$
    by (*simp add*: *singleton-quotient-in x*)
  have *xU*: $\neg\ (x\ /\!\in\ /\mathbf{1}\,U)$
    by (*simp add*: *notx singleton-quotient-in*)
  have $x = y\ (mod\ /\mathbf{1}\,V)$
    by (*meson UV singleton-quotient-mod subsetD x y*)
  then have $x = y\ (mod\ /\mathbf{1}\,U)$
    by (*meson qequals-sym qsubset-strong-def singleton-UV singleton-quotient-in y*)
  then have *False*
    by (*simp add*: *notx*)
}
with *implies-sub* **have** $(/\mathbf{1}\,U\ /\subseteq\ /\mathbf{1}\,V) \Longrightarrow U = V \vee U = \{\}$
  by *auto*
then show *?thesis*
  using *empty-subset-singleton-quotient-strong* **by** *force*
qed

lemma *subset-universal-singleton-weak*: $q\ /\!\leq\ /\mathbf{1}\mathcal{U}$
  by (*simp add*: *qsubset-weak-def*)

lemma *subset-universal-singleton-bishop*: $(q\ /\sqsubseteq\ /\mathbf{1}\mathcal{U}) = (q = /\mathbf{1}(QField\ q))$
proof –
{
  assume *q*: $q\ /\sqsubseteq\ /\mathbf{1}\mathcal{U}$
  then have $\bigwedge x\ y.\ x\ /\!\in\ q \wedge y\ /\!\in\ q \Longrightarrow (x = y\ (mod\ q)) = (x = y\ (mod\ /\mathbf{1}\mathcal{U}))$
    by (*simp add*: *qsubset-bishop-def*)
  then have $\bigwedge x\ y.\ x\ /\!\in\ q \wedge y\ /\!\in\ q \Longrightarrow x = y\ (mod\ q)$
    by *simp*
  then have $q = /\mathbf{1}(QField\ q)$
    by (*meson qequals-in qin-def qsubset-antisym-weak qsubset-weak-def singleton-quotient-mod*)
}
then show *?thesis*
  by (*metis singleton-quotient-subset-bishop subset-UNIV*)
qed

lemma *subset-universal-singleton-strong*: $(q\ /\subseteq\ /\mathbf{1}\mathcal{U}) = (q = /\emptyset \vee q = /\mathbf{1}\mathcal{U})$
proof –
{

```
    assume q: q /⊆ /1𝒰
    then have q /⊑ /1𝒰
      using qsubset-strong-implies-bishop by auto
    then have q = /1(QField q)
      using subset-universal-singleton-bishop by blast
  }
  note isSingleton = this
  {
    assume q: q /⊆ /1𝒰
    then have QField q = UNIV ∨ QField q = {}
      by (metis q singleton-quotient-subset-strong isSingleton)
    then have q = /1𝒰 ∨ q = /∅
      using isSingleton q by auto
  }
  then show ?thesis
    using empty-subset-singleton-quotient-strong by auto
qed

lemma identity-QField-subset-weak: /≡(QField q) /≤ q
 by (metis eq-equiv-class equiv-QField-Rel qequals-def qsubset-weak-def set-quotient-mod)

lemma identity-QField-subset-bishop: (/≡(QField q) /⊑ q) = (q = /≡(QField q))
  by (metis qin-def qubseteq-refl-bishop same-QField-bishop set-quotient-in subsetI
subset-antisym)

lemma identity-QField-subset-strong: (/≡(QField q) /⊆ q) = (q = /≡(QField q))
  using identity-QField-subset-bishop qsubset-strong-implies-bishop by auto

lemma qsubset-weak-neq-bishop:
  assumes xy: (x::'a) ≠ y
  shows ((/≤) :: 'a quotient ⇒ 'a quotient ⇒ bool) ≠ (/⊑)
proof –
  let ?U = /≡{x, y}
  have sub: ?U /≤ /1𝒰
    by (simp add: subset-universal-singleton-weak)
  from xy have notsub: ¬ (?U /⊑ /1𝒰)
   by (metis insert-iff set-quotient-mod singleton-quotient-mod subset-universal-singleton-bishop)
  show ?thesis using sub notsub by auto
qed

lemma qsubset-bishop-neq-strong:
  assumes xy: (x::'a) ≠ y
  shows ((/⊑) :: 'a quotient ⇒ 'a quotient ⇒ bool) ≠ (/⊆)
proof –
  let ?U = /≡{x}
  have sub: ?U /⊑ /1𝒰
    by (simp add: qsubset-bishop-def set-quotient-in set-quotient-mod)
  from xy have notsub: ¬ (?U /⊆ /1𝒰)
   by (metis(full-types) UNIV-I empty-quotient-in insertI1 set-quotient-in set-quotient-mod
```

singleton-quotient-mod subset-universal-singleton-strong)
show *?thesis* **using** *sub notsub* **by** *auto*
qed

lemma *qsubset-weak-neq-strong*:
assumes *xy*: $(x::'a) \neq y$
shows $((/\leq) :: 'a\ quotient \Rightarrow 'a\ quotient \Rightarrow bool) \neq (/\subseteq)$
using *qsubset-bishop-implies-weak qsubset-strong-implies-bishop qsubset-weak-neq-bishop xy* **by** *fastforce*

4.10 Functions between Quotients

definition *qequal-fun* ::
$'a\ quotient \Rightarrow 'b\ quotient \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a => 'b) \Rightarrow bool$
where
$qequal\text{-}fun\ p\ q\ f\ g = (\forall\ x\ y.\ x = y\ (mod\ p) \longrightarrow f\ x = g\ y\ (mod\ q))$

lemma *qequal-fun-sym*: *symp* (*qequal-fun p q*) **by** (*metis qequal-fun-def qequals-sym sympI*)

lemma *qequal-fun-trans*: *transp* (*qequal-fun p q*)
by (*smt* (*verit*) *qclass-elems qequal-fun-def qequals-implies-equal-qclasses transpI*)

definition *fun-quotient* :: $'a\ quotient \Rightarrow 'b\ quotient \Rightarrow ('a \Rightarrow 'b)\ quotient$ (**infixr** $'/\Rightarrow$ *90*) **where**
$p\ /\Rightarrow\ q = QuotientP\ (qequal\text{-}fun\ p\ q)$

lemma *fun-quotient-Rel*: $Rel\ (p\ /\Rightarrow\ q) = \{\ (f,\ g)\ .\ qequal\text{-}fun\ p\ q\ f\ g\ \}$
by (*simp add*: *QuotientP-Rel fun-quotient-def qequal-fun-sym qequal-fun-trans*)

lemma *fun-quotient-mod*: $(f = g\ (mod\ p\ /\Rightarrow\ q)) = (qequal\text{-}fun\ p\ q\ f\ g)$
by (*metis QuotientP-mod fun-quotient-def qequal-fun-sym qequal-fun-trans*)

lemma *fun-quotient-in*: $(f\ /\in\ p\ /\Rightarrow\ q) = (qequal\text{-}fun\ p\ q\ f\ f)$
by (*simp add*: *fun-quotient-mod qin-mod*)

lemma *fun-quotient-app-in*: $f\ /\in\ p\ /\Rightarrow\ q \Longrightarrow x\ /\in\ p \Longrightarrow f\ x\ /\in\ q$
by (*meson fun-quotient-in qequal-fun-def qin-mod*)

lemma *fun-quotient-app-mod*: $f = g\ (mod\ p\ /\Rightarrow\ q) \Longrightarrow x = y\ (mod\ p) \Longrightarrow f\ x = g\ y\ (mod\ q)$
by (*meson qequal-fun-def fun-quotient-mod*)

lemma *fun-quotient-app-in-mod*: $f\ /\in\ p\ /\Rightarrow\ q \Longrightarrow x = y\ (mod\ p) \Longrightarrow f\ x = f\ y\ (mod\ q)$
by (*meson fun-quotient-app-mod qin-mod*)

lemma *fun-quotient-compose*: $(\circ)\ /\in\ (q\ /\Rightarrow\ r)\ /\Rightarrow\ (p\ /\Rightarrow\ q)\ /\Rightarrow\ (p\ /\Rightarrow\ r)$

by (*auto simp add*: *fun-quotient-in qequal-fun-def fun-quotient-mod*)

lemma *fun-quotient-empty-domain*: $(/\emptyset\ /\Rightarrow\ q) = /\mathbf{1}\mathcal{U}$
by (*metis empty-quotient-mod fun-quotient-mod qequal-fun-def qsubset-antisym-weak qsubset-weak-def subset-universal-singleton-weak*)

lemma *fun-quotient-empty-range*: $q \neq /\emptyset \Longrightarrow (q\ /\Rightarrow\ /\emptyset) = /\emptyset$
by (*metis empty-quotient-in fun-quotient-app-in qsubset-antisym-strong qsubset-strong-def*)

lemma *fun-quotient-subset-weak-intro*:
assumes $p2\ /\leq\ p1 \wedge q1\ /\leq\ q2$
shows $p1\ /\Rightarrow\ q1\ /\leq\ p2\ /\Rightarrow\ q2$
by (*smt* (*verit*, *best*) *assms fun-quotient-mod qequal-fun-def qsubset-weak-def*)

lemma *fun-quotient-subset-weakdef*:
$(p1\ /\Rightarrow\ q1\ /\leq\ p2\ /\Rightarrow\ q2) =$
$(\forall\ f\ g.\ (\forall x\ y.\ x = y\ (mod\ p1) \longrightarrow f\ x = g\ y\ (mod\ q1)) \longrightarrow$
$(\forall x\ y.\ x = y\ (mod\ p2) \longrightarrow f\ x = g\ y\ (mod\ q2)))$
by (*simp add*: *fun-quotient-mod qequal-fun-def qsubset-weak-def*)

lemma *fun-quotient-range-subset-weak*:
assumes *sub*: $((p1 :: 'a\ quotient)\ /\Rightarrow\ q1\ /\leq\ p2\ /\Rightarrow\ q2)$
assumes *nonempty*: $p2 \neq /\emptyset$
shows $q1\ /\leq\ q2$
proof –
{
assume *contra*: $\neg\ q1\ /\leq\ q2$
have $\exists\ a\ b.\ (a = b\ (mod\ q1)) \wedge \neg\ (a = b\ (mod\ q2))$
using *contra qsubset-weak-def* **by** *blast*
then obtain *a b* **where** *ab*: $(a = b\ (mod\ q1)) \wedge \neg\ (a = b\ (mod\ q2))$ **by** *blast*
let $?f = \lambda\ x :: 'a.\ a$
let $?g = \lambda\ x :: 'a.\ b$
have $\forall\ x\ y.\ x = y\ (mod\ p1) \longrightarrow ?f\ x = ?g\ y\ (mod\ q1)$
using *ab* **by** *blast*
then have $\forall\ x\ y.\ x = y\ (mod\ p2) \longrightarrow ?f\ x = ?g\ y\ (mod\ q2)$
using *fun-quotient-subset-weakdef*[*of p1 q1 p2 q2*, *simplified sub*]
by *meson*
then have $\bigwedge\ x\ y.\ x = y\ (mod\ p2) \Longrightarrow a = b\ (mod\ q2)$
by *blast*
with *nonempty* **have** *False*
by (*metis ab empty-quotient-mod fun-quotient-empty-range fun-quotient-mod qequal-fun-def*)
}
then show *?thesis* **by** *blast*
qed

lemma *trivializing-qsuperset*:
shows $(/\mathbf{1}(QField\ p)\ /\leq\ q) = (\neg\ (\exists\ x\ y.\ x\ /\in\ p \wedge y\ /\in\ p \wedge x \neq y\ (mod\ q)))$

by (*meson qin-def qsubset-weak-def singleton-quotient-mod*)

lemma *fun-quotient-domain-subset-weak*:
assumes *sub*: $((p1 :: 'a\ quotient)\ /\Rightarrow q1\ /\leq p2\ /\Rightarrow q2)$
assumes *nontrivial*: $\neg\ (/\mathbf{1}(QField\ q1)\ /\leq q2)$
shows $p2\ /\leq p1$
proof –
{
assume *ex*: $\exists\ a\ b.\ a = b\ (mod\ p2) \wedge a \neq b\ (mod\ p1)$

Construct zero and one

then have $p2 \neq /\emptyset$
using *empty-quotient-mod* **by** *fastforce*
then have *q1-sub-q2*: $q1\ /\leq q2$ **using** *fun-quotient-range-subset-weak sub* **by** *auto*
have $\exists\ x\ y.\ x\ /\in q1 \wedge y\ /\in q1 \wedge x \neq y\ (mod\ q2)$
using *nontrivial trivializing-qsuperset* **by** *blast*
then obtain *zero one* **where** *zo*: $zero\ /\in q1 \wedge one\ /\in q1 \wedge zero \neq one\ (mod\ q2)$
by *blast*
then have *zo-q1*: $zero \neq one\ (mod\ q1)$ **using** *q1-sub-q2 qsubset-mod-weak* **by** *force*

have *False*
proof (*cases* $\exists\ a\ b.\ a = b\ (mod\ p2) \wedge a \neq b\ (mod\ p1) \wedge a \neq b$)
case *True*
then obtain *a b* **where** *ab*: $a = b\ (mod\ p2) \wedge a \neq b\ (mod\ p1) \wedge a \neq b$ **by** *blast*
let $?f = \lambda\ x.\ if\ x\ /\in p1\ then\ (if\ x = a\ (mod\ p1)\ then\ one\ else\ zero)\ else\ (if\ x = a\ then\ one\ else\ zero)$
have $\forall\ x\ y.\ x = y\ (mod\ p1) \longrightarrow\ ?f\ x =\ ?f\ y\ (mod\ q1)$
by (*smt* (*verit*, *best*) *qequals-sym qequals-trans qin-mod zo*)
then have $\forall\ x\ y.\ x = y\ (mod\ p2) \longrightarrow\ ?f\ x =\ ?f\ y\ (mod\ q2)$
using *sub*[*simplified fun-quotient-subset-weakdef*] **by** *meson*
then have *contra*: $?f\ a =\ ?f\ b\ (mod\ q2)$ **using** *ab* **by** *blast*
then show *False* **by** (*metis*(*full-types*) *ab qequals-sym qin-mod zo*)
next
case *False*
then have $\exists\ a.\ a = a\ (mod\ p2) \wedge a \neq a\ (mod\ p1)$ **using** *ex* **by** *blast*
then obtain *a* **where** *a*: $a = a\ (mod\ p2) \wedge a \neq a\ (mod\ p1)$ **by** *blast*
let $?f = \lambda\ x.\ if\ x = a\ then\ one\ else\ zero$
let $?g = \lambda\ x.\ zero$
have $\forall\ x\ y.\ x = y\ (mod\ p1) \longrightarrow\ ?f\ x =\ ?g\ y\ (mod\ q1)$
by (*metis*(*full-types*) *a qequals-in qin-mod zo*)
then have $\forall\ x\ y.\ x = y\ (mod\ p2) \longrightarrow\ ?f\ x =\ ?g\ y\ (mod\ q2)$
using *sub*[*simplified fun-quotient-subset-weakdef*] **by** *meson*
then have *contra*: $?f\ a =\ ?g\ a\ (mod\ q2)$
using *a* **by** *blast*
then show *False* **sledgehammer**

using *qequals-sym zo* **by** *force*
qed
}
then show *?thesis*
using *qsubset-weak-def* **by** *blast*
qed

4.11 Vectors as Quotients

definition *qequal-vector* :: $'a$ *quotient* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ *bool* **where**
qequal-vector q n u v = (*length u* = *n* $\wedge$ *length v* = *n* $\wedge$ ($\forall$ *i* < *n*. *u* ! *i* = *v* ! *i* (*mod q*)))

lemma *qequal-vector-sym*: *symp* (*qequal-vector q n*)
by (*metis qequal-vector-def qequals-sym sympI*)

lemma *qequal-vector-trans*: *transp* (*qequal-vector q n*)
by (*smt* (*verit*, *del-insts*) *qequal-vector-def qequals-trans transpI*)

definition *vector-quotient* :: $'a$ *quotient* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *list quotient* (**infix** $'/\widehat{}$ *100*)
where
q /^ *n* = *QuotientP* (*qequal-vector q n*)

lemma *vector-quotient-Rel*: *Rel* (*q* /^ *n*) = { (*u*, *v*). *qequal-vector q n u v* }
by (*simp add*: *QuotientP-Rel qequal-vector-sym qequal-vector-trans vector-quotient-def*)

lemma *vector-quotient-in*: (*u* /∈ *q* /^ *n*) = (*qequal-vector q n u u*)
by (*simp add*: *QuotientP-in qequal-vector-sym qequal-vector-trans vector-quotient-def*)

lemma *vector-quotient-mod*: (*u* = *v* (*mod q* /^ *n*)) = (*qequal-vector q n u v*)
using *qequals-def vector-quotient-Rel* **by** *fastforce*

lemma *vector-quotient-nth*: *i* < *n* $\Longrightarrow$ (λ *u*. *u* ! *i*) /∈ *q* /^ *n* /⇒ *q*
by (*simp add*: *fun-quotient-in qequal-fun-def qequal-vector-def vector-quotient-mod*)

lemma *vector-quotient-nth-in*: *i* < *n* $\Longrightarrow$ *u* /∈ *q* /^ *n* $\Longrightarrow$ *u* ! *i* /∈ *q*
by (*blast intro*: *fun-quotient-app-in*[**where** *f* = λ *u* . *u* ! *i*] *vector-quotient-nth*)

lemma *vector-quotient-nth-mod*: *i* < *n* $\Longrightarrow$ *u* = *v* (*mod q* /^ *n*) $\Longrightarrow$ *u* ! *i* = *v* ! *i* (*mod q*)
by (*blast intro*: *fun-quotient-app-in-mod*[**where** *f* = λ *u* . *u* ! *i*] *vector-quotient-nth*)

lemma *vector-quotient-append*: (@) /∈ *q* /^ *n* /⇒ *q* /^ *m* /⇒ *q* /^ (*n*+*m*)
by (*simp add*: *fun-quotient-in fun-quotient-mod nth-append qequal-fun-def qequal-vector-def*
vector-quotient-mod)

lemma *vector-quotient-append-in*: *x* /∈ *q* /^ *n* $\Longrightarrow$ *y* /∈ *q* /^ *m* $\Longrightarrow$ *x* @ *y* /∈ *q* /^ (*n*+*m*)

by (*meson fun-quotient-app-in-mod qin-mod vector-quotient-append*)

lemma *vector-quotient-append-mod*:
$x = x'$ (*mod* q /⌢ n) $\Longrightarrow$ $y = y'$ (*mod* q /⌢ m) $\Longrightarrow$ $x@y = x'@y'$ (*mod* q /⌢ $(n+m)$)
by (*meson fun-quotient-app-in-mod fun-quotient-app-mod vector-quotient-append*)

lemma *vector-quotient-weak-subset-intro*: p /≤ q $\Longrightarrow$ p/⌢n /≤ q/⌢n
by (*simp add*: *qequal-vector-def qsubset-weak-def vector-quotient-mod*)

lemma *vector-quotient-strong-subset-intro*: p /⊆ q $\Longrightarrow$ p/⌢n /⊆ q/⌢n
by (*simp add*: *qequal-vector-def qsubset-strong-def vector-quotient-mod vector-quotient-nth-in*)

lemma *vector-quotient-bishop-subset-intro*: p /⊑ q $\Longrightarrow$ p/⌢n /⊑ q/⌢n
by (*simp add*: *qequal-vector-def qsubset-bishop-def vector-quotient-mod vector-quotient-nth-in*)

4.12 Tuples as Quotients

definition *qequal-tuple* :: *'a quotient list* ⇒ *'a list* ⇒ *'a list* ⇒ *bool* **where**
qequal-tuple qs u v = (*length u* = *length qs* ∧ *length v* = *length qs* ∧
(∀ *i* < *length qs*. *u* ! *i* = *v* ! *i* (*mod qs*!*i*)))

lemma *qequal-tuple-sym*: *symp* (*qequal-tuple qs*)
by (*metis qequal-tuple-def qequals-sym sympI*)

lemma *qequal-tuple-trans*: *transp* (*qequal-tuple qs*)
by (*smt* (*verit*, *best*) *qequal-tuple-def qequals-trans transpI*)

definition *tuple-quotient* :: *'a quotient list* ⇒ *'a list quotient* ('/×) **where**
/× *qs* = *QuotientP* (*qequal-tuple qs*)

lemma *tuple-quotient-rel*: *Rel* (/× *qs*) = { (*u*, *v*). *qequal-tuple qs u v* }
by (*simp add*: *QuotientP-Rel tuple-quotient-def qequal-tuple-sym qequal-tuple-trans*)

lemma *tuple-quotient-in*: (*u* /∈ (/× *qs*)) = (*qequal-tuple qs u u*)
by (*simp add*: *QuotientP-in tuple-quotient-def qequal-tuple-sym qequal-tuple-trans*)

lemma *tuple-quotient-mod*: (*u* = *v* (*mod* /× *qs*)) = (*qequal-tuple qs u v*)
by (*simp add*: *QuotientP-mod tuple-quotient-def qequal-tuple-sym qequal-tuple-trans*)

lemma *tuple-quotient-nth*: *i* < *length qs* $\Longrightarrow$ (λ *u*. *u* ! *i*) /∈ /× *qs* /⇒ *qs* ! *i*
by (*simp add*: *fun-quotient-in qequal-fun-def qequal-tuple-def tuple-quotient-mod*)

lemma *tuple-quotient-append*: (@) /∈ /× *ps* /⇒ /× *qs* /⇒ /× (*ps*@*qs*)
by (*simp add*: *QuotientP-mod fun-quotient-in fun-quotient-mod nth-append tuple-quotient-def*
qequal-fun-def qequal-tuple-def qequal-tuple-sym qequal-tuple-trans)

lemma *vectors-are-tuples*: q /⌢ n = /× (*replicate n q*)

by (*smt* (*verit*, *ccfv-SIG*) *vector-quotient-def tuple-quotient-def*
Collect-cong QuotientP-def case-prodD case-prodI2
length-replicate nth-replicate qequal-tuple-def qequal-vector-def)

lemma *tuple-quotient-strong-subset-intro*:
length ps = *length qs* $\Longrightarrow$ ($\bigwedge$ *i*. *i* < *length ps* $\Longrightarrow$ *ps*!*i* /⊆ *qs*!*i*) $\Longrightarrow$ /× *ps* /⊆ /× *qs*
by (*smt* (*verit*, *ccfv-threshold*) *qequal-tuple-def qequals-in qsubset-strong-def tuple-quotient-in tuple-quotient-mod*)

lemma *tuple-quotient-bishop-subset-intro*:
length ps = *length qs* $\Longrightarrow$ ($\bigwedge$ *i*. *i* < *length ps* $\Longrightarrow$ *ps*!*i* /⊑ *qs*!*i*) $\Longrightarrow$ /× *ps* /⊑ /× *qs*
apply (*auto simp add*: *qsubset-bishop-def*)
by (*metis* (*no-types*, *lifting*) *qequal-tuple-def qin-mod tuple-quotient-mod*)+

end
theory *Algebra* **imports** *Signature Quotients*
begin

5 Abstraction Algebra

We will abbreviate *Abstraction Algebra* by leaving the prefix *Algebra* implicit, and just saying *Algebra* instead.

5.1 Operations and Operators as Quotients

type-synonym *'a operation* = *'a list* ⇒ *'a*
type-synonym *'a operator* = *'a operation list* ⇒ *'a*

definition *operations* :: *'a quotient* ⇒ *nat* ⇒ (*'a operation*) *quotient* **where**
operations U n = *U* /^ *n* /⇒ *U*

definition *operators* :: *'a quotient* ⇒ *shape* ⇒ (*'a operator*) *quotient* **where**
operators U s = /× (*map* (λ *deps*. *operations U* (*card deps*)) (*Preshape s*)) /⇒ *U*

definition *value-op* :: *'a* ⇒ *'a operation* **where**
value-op u = (λ -. *u*)

lemma *operators-appeq-intro*:
assumes *FG*: *F* = *G* (*mod operators* $\mathcal{U}$ *s*)
assumes *lenfs*: *length fs* = §*ar s*
assumes *lengs*: *length gs* = §*ar s*
assumes *fsgs*: ($\bigwedge$ *i*. *i* < §*ar s* $\Longrightarrow$ *fs*!*i* = *gs*!*i* (*mod operations* $\mathcal{U}$ (*s*.#*i*)))
shows *F fs* = *G gs* (*mod* $\mathcal{U}$)
proof −
have *appeq*: $\bigwedge$ *x y*. *x* = *y* (*mod* /× (*map* (λ*deps*. *operations* $\mathcal{U}$ (*card deps*)) (*Preshape s*))) $\Longrightarrow$ *F x* = *G y* (*mod* $\mathcal{U}$)

using *FG*[*simplified operators-def fun-quotient-mod qequal-fun-def*]
by *blast*
show *?thesis*
apply (*rule appeq*)
apply (*simp add*: *tuple-quotient-mod qequal-tuple-def*)
apply *auto*
using *fsgs*
apply (*simp add*: *lenfs shape-arity-def*)
apply (*simp add*: *lengs shape-arity-def*)
using *fsgs shape-arity-def shape-deps-def* **by** *auto*
qed

lemma *operator-appeq-intro*:
assumes *F*: F /$\in$ *operators* $\mathcal{U}$ s
assumes *lenfs*: *length fs* = §*ar s*
assumes *lengs*: *length gs* = §*ar s*
assumes *fsgs*: ($\bigwedge$ *i*. *i* < §*ar s* $\Longrightarrow$ *fs*!*i* = *gs*!*i* (*mod operations* $\mathcal{U}$ (*s*.#*i*)))
shows *F fs* = *F gs* (*mod* $\mathcal{U}$)
by (*meson F fsgs lenfs lengs operators-appeq-intro qin-mod*)

lemma *operations-eq-intro*:
assumes $\bigwedge$ *us vs*. *us* = *vs* (*mod* $\mathcal{U}$ /^ *n*) $\Longrightarrow$ *f us* = *g vs* (*mod* $\mathcal{U}$)
shows *f* = *g* (*mod operations* $\mathcal{U}$ *n*)
by (*simp add*: *assms fun-quotient-mod operations-def qequal-fun-def*)

lemma *operations-mod*:
(*f* = *g* (*mod operations* $\mathcal{U}$ *n*)) = ($\forall$ *us vs*. *us* = *vs* (*mod* $\mathcal{U}$/^*n*) $\longrightarrow$ *f us* = *g vs* (*mod* $\mathcal{U}$))
by (*metis fun-quotient-app-mod operations-def operations-eq-intro*)

5.2 Compatibility of Shape and Operator

definition *shape-compatible* :: ′*a quotient* $\Rightarrow$ *shape* $\Rightarrow$ ′*a operator* $\Rightarrow$ *bool* **where**
shape-compatible U s op = (*op* /$\in$ *operators U s*)

definition *shape-compatible-opt* :: ′*a quotient* $\Rightarrow$ *shape option* $\Rightarrow$ ′*a operator option* $\Rightarrow$ *bool* **where**
shape-compatible-opt U s op = ((*s* = *None* $\wedge$ *op* = *None*) $\vee$ (*s* $\neq$ *None* $\wedge$ *op* $\neq$ *None* $\wedge$
shape-compatible U (*the s*) (*the op*)))

5.3 Abstraction Algebras

type-synonym ′*a operators* = (*abstraction*, ′*a operator*) *map*

type-synonym ′*a prealgebra* = ′*a quotient* $\times$ *signature* $\times$ ′*a operators*

definition *is-algebra* :: ′*a prealgebra* $\Rightarrow$ *bool* **where**
is-algebra paa =
(*let U* = *fst paa in*

let sig = *fst* (*snd paa*) *in*
let ops = *snd* (*snd paa*) *in*
$U \neq /\emptyset \wedge (\forall\ a.\ \textit{shape-compatible-opt}\ U\ (\textit{sig}\ a)\ (\textit{ops}\ a)))$

definition *trivial-prealgebra* :: *'a prealgebra* **where**
trivial-prealgebra = $(/\mathcal{U}$, *Map.empty*, *Map.empty*)

lemma *trivial-prealgebra*: *is-algebra trivial-prealgebra*
by (*metis* (*no-types*, *lifting*) *empty-quotient-in fst-conv is-algebra-def shape-compatible-opt-def snd-conv trivial-prealgebra-def univ-quotient-in*)

typedef *'a algebra* = { *aa* :: *'a prealgebra* . *is-algebra aa* } **morphisms** *Prealgebra Algebra*
using *trivial-prealgebra* **by** *blast*

definition *Univ* :: *'a algebra* ⇒ *'a quotient* **where**
Univ aa = *fst* (*Prealgebra aa*)

definition *Sig* :: *'a algebra* ⇒ *signature* **where**
Sig aa = *fst* (*snd* (*Prealgebra aa*))

definition *Ops* :: *'a algebra* ⇒ *'a operators* **where**
Ops aa = *snd* (*snd* (*Prealgebra aa*))

lemma *Prealgebra-components*: *Prealgebra aa* = (*Univ aa*, *Sig aa*, *Ops aa*)
by (*simp add*: *Ops-def Sig-def Univ-def*)

lemma *Univ-nonempty*: *Univ aa* $\neq /\emptyset$
by (*metis Prealgebra Univ-def is-algebra-def mem-Collect-eq*)

lemma *algebra-compatibility*: *shape-compatible-opt* (*Univ aa*) (*Sig aa a*) (*Ops aa a*)
by (*metis Ops-def Prealgebra Sig-def Univ-def is-algebra-def mem-Collect-eq*)

end
theory *NTerm* **imports** *Algebra*
begin

6 Term

6.1 Variables

type-synonym *variable* = *string*

type-synonym *variables* = (*variable* × *nat*) *set*

definition *binders-as-vars* :: *variable list* ⇒ *variables* (-‘,0 [*1000*] *1000*) **where**
xs‘,0 = { (*x*, *0*) | *x*. *x* ∈ *set xs* }

lemma *binders-as-vars-empty*[*simp*]: []‘,*0* = {}
  by (*simp add*: *binders-as-vars-def*)

lemma *deduction-forall-deps-0*[*iff*]: 𝔇!!*abstr-forall*.@*0*([*x*]) = [*x*]
  apply (*auto*)
  apply (*subst has-shape-get*)
  apply *blast*
  apply (*subst operator-shape-deps-0*)
  by *blast*

6.2 Terms

datatype *nterm* =
  VarApp variable nterm list
| *AbsApp abstraction variable list nterm list*

definition *xvar* :: *variable* (‘*x*) **where** ‘*x* = ''*x*''
definition *xvar0* :: *nterm* (§*x*) **where** §*x* = *VarApp* ‘*x* []

definition *yvar* :: *variable* (‘*y*) **where** ‘*y* = ''*y*''
definition *yvar0* :: *nterm* (§*y*) **where** §*y* = *VarApp* ‘*y* []

definition *Avar* :: *variable* (‘*A*) **where** ‘*A* = ''*A*''
definition *Avar0* :: *nterm* (§*A*) **where** §*A* = *VarApp* ‘*A* []
definition *Avar1* :: *nterm* ⇒ *nterm* (§*A*[-]) **where** §*A*[*t*] = *VarApp* ‘*A* [*t*]

definition *Bvar* :: *variable* (‘*B*) **where** ‘*B* = ''*B*''
definition *Bvar0* :: *nterm* (§*B*) **where** §*B* = *VarApp* ‘*B* []
definition *Bvar1* :: *nterm* ⇒ *nterm* (§*B*[-]) **where** §*B*[*t*] = *VarApp* ‘*B* [*t*]

definition *Cvar* :: *variable* (‘*C*) **where** ‘*C* = ''*C*''
definition *Cvar0* :: *nterm* (§*C*) **where** §*C* = *VarApp* ‘*C* []
definition *Cvar1* :: *nterm* ⇒ *nterm* (§*C*[-]) **where** §*C*[*t*] = *VarApp* ‘*C* [*t*]

definition *implies-app* :: *nterm* ⇒ *nterm* ⇒ *nterm* (**infix** ‘⇒ *225*) **where**
  A ‘⇒ *B* = *AbsApp abstr-implies* [] [*A*, *B*]

definition *true-app* :: *nterm* (‘⊤) **where** ‘⊤ = *AbsApp abstr-true* [] []

definition *false-app* :: *nterm* (‘⊥) **where** ‘⊥ = *AbsApp abstr-false* [] []

definition *forall-app* :: *variable* ⇒ *nterm* ⇒ *nterm* ((*3*‘∀ -. -) [*1000*, *210*] *210*)
where
  forall-app x P = *AbsApp abstr-forall* [*x*] [*P*]

6.3 Wellformedness

fun *nt-wf* :: *signature* ⇒ *nterm* ⇒ *bool* **where**
  nt-wf sig (*VarApp x ts*) = (∀ *t* = *ts*!-. *nt-wf sig t*)
| *nt-wf sig* (*AbsApp a xs ts*) =

($sig\text{-}contains\ sig\ a\ (length\ xs)\ (length\ ts) \wedge$
$distinct\ xs \wedge$
$(\forall\ t = ts!\text{-}.\ nt\text{-}wf\ sig\ t))$

lemma $nt\text{-}wf\text{-}x0[iff]$: $nt\text{-}wf\ sig\ \S x$ **by** ($simp\ add$: $xvar0\text{-}def$)
lemma $nt\text{-}wf\text{-}y0[iff]$: $nt\text{-}wf\ sig\ \S y$ **by** ($simp\ add$: $yvar0\text{-}def$)
lemma $nt\text{-}wf\text{-}A0[iff]$: $nt\text{-}wf\ sig\ \S A$ **by** ($simp\ add$: $Avar0\text{-}def$)
lemma $nt\text{-}wf\text{-}A1[iff]$: $nt\text{-}wf\ sig\ \S A[t] = nt\text{-}wf\ sig\ t$
by ($simp\ add$: $Avar1\text{-}def$)
lemma $nt\text{-}wf\text{-}B0[iff]$: $nt\text{-}wf\ sig\ \S B$ **by** ($simp\ add$: $Bvar0\text{-}def$)
lemma $nt\text{-}wf\text{-}B1[iff]$: $nt\text{-}wf\ sig\ \S B[t] = nt\text{-}wf\ sig\ t$
by ($simp\ add$: $Bvar1\text{-}def$)
lemma $nt\text{-}wf\text{-}C0[iff]$: $nt\text{-}wf\ sig\ \S C$ **by** ($simp\ add$: $Cvar0\text{-}def$)
lemma $nt\text{-}wf\text{-}C1[iff]$: $nt\text{-}wf\ sig\ \S C[t] = nt\text{-}wf\ sig\ t$
by ($simp\ add$: $Cvar1\text{-}def$)

lemma $nt\text{-}wf\text{-}true[simp]$: $nt\text{-}wf\ \mathfrak{D}\ `\top$
by ($simp\ add$: $true\text{-}app\text{-}def$)

lemma $nt\text{-}wf\text{-}implies[simp]$: $nt\text{-}wf\ \mathfrak{D}\ (A\ `\Rightarrow\ B) = (nt\text{-}wf\ \mathfrak{D}\ A \wedge nt\text{-}wf\ \mathfrak{D}\ B)$
by ($auto\ simp\ add$: $implies\text{-}app\text{-}def\ shift\text{-}index\text{-}def$)

lemma $nt\text{-}wf\text{-}forall[simp]$: $nt\text{-}wf\ \mathfrak{D}\ (`\forall\ `x.\ t) = nt\text{-}wf\ \mathfrak{D}\ t$
by ($auto\ simp\ add$: $forall\text{-}app\text{-}def$)

lemma $sig\text{-}extends\text{-}nt\text{-}wf$: $V \succeq U \Longrightarrow nt\text{-}wf\ U\ t \Longrightarrow nt\text{-}wf\ V\ t$
proof ($induct\ t$)
case ($VarApp\ x\ ts$)
then show $?case$ **by** $simp$
next
case ($AbsApp\ a\ xs\ ts$)
then show $?case$
by ($auto\ simp\ add$: $extends\text{-}sig\text{-}contains$)
qed

6.4 Free Variables

fun $nt\text{-}free :: signature \Rightarrow nterm \Rightarrow variables$ **where**
$nt\text{-}free\ sig\ (VarApp\ x\ ts) =$
$(\S fold\ X = \{(x,\ length\ ts)\},\ t = ts!\text{-}.\ X \cup nt\text{-}free\ sig\ t)$
$|\ nt\text{-}free\ sig\ (AbsApp\ a\ xs\ ts) =$
$(\S fold\ X = \{\},\ t = ts!i.\ X \cup (nt\text{-}free\ sig\ t - (sig!!a.@i(xs))`,0))$

lemma $nt\text{-}free\text{-}x0$: $nt\text{-}free\ sig\ \S x = \{(`x,\ 0)\}$ **by** ($simp\ add$: $xvar0\text{-}def$)

lemma $nt\text{-}free\text{-}y0$: $nt\text{-}free\ sig\ \S y = \{(`y,\ 0)\}$ **by** ($simp\ add$: $yvar0\text{-}def$)

lemma $nt\text{-}free\text{-}A0$: $nt\text{-}free\ sig\ \S A = \{(`A,\ 0)\}$ **by** ($simp\ add$: $Avar0\text{-}def$)
lemma $nt\text{-}free\text{-}A1$: $nt\text{-}free\ sig\ \S A[t] = \{(`A,\ Suc\ 0)\} \cup nt\text{-}free\ sig\ t$

by (*simp add*: *Avar1-def*)

lemma *nt-free-B0*: *nt-free sig* §*B* = {(‘*B*, *0*)} **by** (*simp add*: *Bvar0-def*)
lemma *nt-free-B1*: *nt-free sig* §*B*[*t*] = {(‘*B*, *Suc 0*)} ∪ *nt-free sig t*
by (*simp add*: *Bvar1-def*)

lemma *nt-free-C0*: *nt-free sig* §*C* = {(‘*C*, *0*)} **by** (*simp add*: *Cvar0-def*)
lemma *nt-free-C1*: *nt-free sig* §*C*[*t*] = {(‘*C*, *Suc 0*)} ∪ *nt-free sig t*
by (*simp add*: *Cvar1-def*)

lemma *nt-free-true*: *nt-free* $\mathfrak{D}$ ‘⊤ = {}
by (*simp add*: *true-app-def*)

lemma *nt-free-implies*: *nt-free* $\mathfrak{D}$ (*s* ‘⇒ *t*) = *nt-free* $\mathfrak{D}$ *s* ∪ *nt-free* $\mathfrak{D}$ *t*
by (*auto simp add*: *implies-app-def shift-index-def*)

lemma *nt-free-forall*: *nt-free* $\mathfrak{D}$ (‘∀ *x*. *t*) = *nt-free* $\mathfrak{D}$ *t* − {(*x*, *0*)}
thm *forall-app-def binders-as-vars-def*
apply (*subst forall-app-def*)
apply *auto*
apply (*auto simp add*: *binders-as-vars-def*)
using *deduction-sig-forall has-shape-get* **by** *auto*

lemma *sig-extends-nt-free*: *V* ⪰ *U* ⟹ *nt-wf U t* ⟹ *nt-free V t* = *nt-free U t*
proof(*induct t*)
case (*VarApp x ts*)
then show *?case*
apply *simp*
apply (*subst list-indexed-fold-cong*)
using *VarApp*
by *auto*
next
case (*AbsApp a xs ts*)
let *?F* = λ *i X t*. *X* ∪ (*nt-free V t* − (*V*!!*a*.@*i*(*xs*))‘,*0*)
let *?G* = λ *i X t*. *X* ∪ (*nt-free U t* − (*U*!!*a*.@*i*(*xs*))‘,*0*)
show *?case*
apply *simp*
thm *list-indexed-fold-eq*[**where** *?F* = *?F* **and** *?G* = *?G*]
apply (*subst list-indexed-fold-eq*[**where** *?F* = *?F* **and** *?G* = *?G*])
using *AbsApp*
apply *auto*
apply (*metis* (*no-types*, *lifting*) *extends-sig-def map-forced-get-def option.case-eq-if*
sig-contains-def)
by (*metis* (*no-types*, *lifting*) *extends-sig-def map-forced-get-def option.case-eq-if*
sig-contains-def)
qed

lemma *nt-free-VarApp*: *nt-free sig* (*VarApp x ts*) =
  $\{(x,\ length\ ts)\} \cup \bigcup\ \{\ nt\text{-}free\ sig\ t \mid t.\ t \in set\ ts\ \}$
proof –
  have *nt-free sig* (*VarApp x ts*) = (§*fold X* = {(*x*, *length ts*)}, *t* = *ts*!-. *X* ∪ *nt-free sig t*)
    by *simp*
  moreover have (§*fold X* = {(*x*, *length ts*)}, *t* = *ts*!-. *X* ∪ *nt-free sig t*) =
      $\{(x,\ length\ ts)\} \cup \bigcup\ \{\ nt\text{-}free\ sig\ t \mid t.\ t \in set\ ts\ \}$
    by (*subst union-unindexed-fold*, *simp*)
  ultimately show *?thesis* **by** *simp*
qed

lemma *nt-free-VarApp-arg-subset*:
  assumes *nt-free sig* (*VarApp x ts*) ⊆ *X*
  assumes *i* < *length ts*
  shows *nt-free sig* (*ts* ! *i*) ⊆ *X*
  by (*smt* (*verit*, *best*) *CollectI assms*(*1*) *assms*(*2*) *le-supE mem-simps*(*9*) *nt-free-VarApp nth-mem subset-iff*)

lemma *nt-free-ConsApp*:
  shows *nt-free sig* (*AbsApp a xs ts*) =
    $\bigcup\ \{\ nt\text{-}free\ sig\ (ts!i) - (sig!!a.@i(xs))`,0 \mid i.\ i < length\ ts\ \}$
  by (*simp add*: *union-indexed-fold*)

lemma *nt-free-ConsApp-arg-subset*:
  assumes *nt-free sig* (*AbsApp a xs ts*) ⊆ *X*
  assumes *i* < *length ts*
  shows *nt-free sig* (*ts*!*i*) ⊆ *X* ∪ (*sig*!!*a*.@*i*(*xs*))‘,*0*
proof –
  have *nt-free sig* (*ts*!*i*) − (*sig*!!*a*.@*i*(*xs*))‘,*0* ⊆ *X*
   by (*smt* (*verit*, *del-insts*) *CollectI assms*(*1*) *assms*(*2*) *mem-simps*(*9*) *nt-free-ConsApp subset-eq*)
  then show *?thesis* **by** *blast*
qed

end
theory *Locales* **imports** *NTerm*
begin

6.5 Signature Locale

locale *sigloc* =
  fixes *Signature* :: *signature* ($\mathcal{S}$)

context *sigloc*
begin

abbreviation
  Deps :: *abstraction* ⇒ *nat* ⇒ *nat set* (**infixl** !♮ *100*)

where $a\ !\natural\ i \equiv \mathcal{S}!!a.\natural i$

abbreviation
CardDeps :: *abstraction* ⇒ *nat* ⇒ *nat* (**infixl** !# *100*)
where $a\ !\#\ i \equiv \mathcal{S}!!a.\#i$

abbreviation
SelDeps :: *abstraction* ⇒ *nat* ⇒ *'b list* ⇒ *'b list* (-!@-'(-') [*100*, *101*, *0*] *100*)
where $a!@i(xs) \equiv \mathcal{S}!!a.@i(xs)$

abbreviation *wf* :: *nterm* ⇒ *bool*
where *wf t* ≡ *nt-wf* $\mathcal{S}$ *t*

abbreviation *frees* :: *nterm* ⇒ *variables*
where *frees t* ≡ *nt-free* $\mathcal{S}$ *t*

abbreviation *is-valid-abstraction* :: *abstraction* ⇒ *bool* (✓)
where ✓*a* ≡ (($\mathcal{S}$ *a*) ≠ *None*)

abbreviation *valence-of-abstraction* :: *abstraction* ⇒ *nat* (§*v*)
where §*v a* ≡ §*val* ($\mathcal{S}$!!*a*)

abbreviation *arity-of-abstraction* :: *abstraction* ⇒ *nat* (§*a*)
where §*a a* ≡ §*ar* ($\mathcal{S}$!!*a*)

lemma *wf-implies-valid-abs*:
assumes *wf*: *wf* (*AbsApp a xs ts*)
shows ✓*a*
proof –
have *sig-contains* $\mathcal{S}$ *a* (*length xs*) (*length ts*)
using *local.wf nt-wf.simps*(*2*) **by** *blast*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *option.case-eq-if sig-contains-def*)
qed

lemma *wf-VarApp*: *wf* (*VarApp x ts*) = (∀ *t* ∈ *set ts*. *wf t*)
by *simp*

lemma *wf-AbsApp-valence*: **assumes** *wf*: *wf* (*AbsApp a xs ts*) **shows** *length xs* =
§*v a*
by (*smt* (*z3*) *local.wf map-forced-get-def nt-wf.simps*(*2*) *option.case-eq-if sig-contains-def*)

lemma *shape-deps-upper-bound*: ✓*a* ⟹ *i* < §*a a* ⟹ *a*!♮*i* ⊆ *nats* (§*v a*)
using *preshape-valence shape-deps-in-alldeps* **by** *auto*

lemma *finite-shape-deps*: ✓*a* ⟹ *i* < §*a a* ⟹ *finite*(*a*!♮*i*)
by (*meson finite-nats finite-subset sigloc.shape-deps-upper-bound*)

lemma *length-boundvars-at*:

assumes *wf*: *wf* (*AbsApp a xs ts*)
assumes *i*: *i* < *length ts*
shows *length* (*a*!@*i*(*xs*)) = *a* !# *i*
proof −
have *valid*: ✓*a*
using *wf wf-implies-valid-abs* **by** *blast*
have *val*: §*v a* = *length xs*
by (*metis wf-AbsApp-valence wf*)
have *deps*: (*a*!♮*i*) ⊆ *nats* (§*v a*)
by (*smt* (*z3*) *shape-deps-upper-bound i local.wf map-forced-get-def nt-wf.simps*(*2*)

option.case-eq-if sig-contains-def)
then show *?thesis*
by (*simp add*: *nats-length-nths val*)
qed

definition *closed* :: *nterm* ⇒ *bool*
where *closed t* = (*frees t* = {})

end

6.6 Abstraction Algebra Locale

locale *algloc* = *sigloc Sig* 𝔄 **for** *AA* :: ′*a algebra* (𝔄)
begin

abbreviation
Universe :: ′*a quotient* (𝒰)
where 𝒰 ≡ *Univ* 𝔄

abbreviation
Operators :: ′*a operators* (𝒪)
where 𝒪 ≡ *Ops* 𝔄

abbreviation
Signature :: *signature* (𝒮)
where 𝒮 ≡ *Sig* 𝔄

notation
Deps (**infixl** !♮ *100*) **and**
CardDeps (**infixl** !# *100*) **and**
SelDeps (-!@-′(-′) [*100*, *101*, *0*] *100*) **and**
is-valid-abstraction (✓) **and**
valence-of-abstraction (§*v*) **and**
arity-of-abstraction (§*a*)

end

context *algloc* **begin**

lemma *valid-in-operators*: ✓$a \Longrightarrow (\mathcal{O}!!a)$ /∈ *operators* $\mathcal{U}$ $(\mathcal{S}!!a)$
by (*metis algebra-compatibility map-forced-get-def shape-compatible-def shape-compatible-opt-def*)

end

end
theory *Valuation* **imports** *NTerm Algebra Locales*
begin

7 Valuation

7.1 Valuations

type-synonym *'a valuation* = (*variable* × *nat*) ⇒ *'a operation*

definition *update-valuation* :: *'a valuation* ⇒ *variable list* ⇒ *'a list* ⇒ *'a valuation*

(-{- := -} [*1000*, *51*, *51*] *1000*)
where
υ\{*xs* := *us*\} = (λ (*x*, *n*).
(*if n* = *0 then*
(*case index-of x xs of*
Some i ⇒ *value-op* (*us*!*i*)
| *None* ⇒ υ (*x*, *0*))
else υ (*x*, *n*)))

definition *qequal-valuation* :: *variables* ⇒ *'a quotient* ⇒ *'a valuation* ⇒ *'a valuation* ⇒ *bool*
where
qequal-valuation X $\mathcal{U}$ τ υ = (∀ (*x*, *n*) ∈ *X*. τ (*x*, *n*) = υ (*x*, *n*) (*mod operations* $\mathcal{U}$ *n*))

lemma *qequal-valuation-sym*: *symp* (*qequal-valuation* X $\mathcal{U}$)
by (*metis* (*no-types*, *lifting*) *case-prodD case-prodI2 qequal-valuation-def qequals-sym sympI*)

lemma *qequal-valuation-trans*: *transp* (*qequal-valuation* X $\mathcal{U}$)
by (*smt* (*verit*, *best*) *case-prodD case-prodI2 qequal-valuation-def qequals-trans transpI*)

definition *valuation-quotient* :: *variables* ⇒ *'a quotient* ⇒ *'a valuation quotient* (**infix** ↣ *90*)
where
$X \rightarrowtail \mathcal{U}$ = *QuotientP* (*qequal-valuation* X $\mathcal{U}$)

lemma *valuation-quotient-Rel*:
Rel ($X \rightarrowtail \mathcal{U}$) = \{ ($\tau$, υ). *qequal-valuation* X $\mathcal{U}$ τ υ \}

by (*simp add*: *QuotientP-Rel qequal-valuation-sym qequal-valuation-trans valuation-quotient-def*)

lemma *valuation-quotient-mod*:
$(\tau = \upsilon\ (mod\ X \rightarrowtail \mathcal{U})) = qequal\text{-}valuation\ X\ \mathcal{U}\ \tau\ \upsilon$
by (*simp add*: *QuotientP-mod qequal-valuation-sym qequal-valuation-trans valuation-quotient-def*)

lemma *valuation-quotient-in*:
$(\upsilon\ /\!\in\ X \rightarrowtail \mathcal{U}) = qequal\text{-}valuation\ X\ \mathcal{U}\ \upsilon\ \upsilon$
by (*simp add*: *qin-mod valuation-quotient-mod*)

lemma *valuation-quotient-app*:
$\tau = \upsilon\ (mod\ X \rightarrowtail \mathcal{U}) \Longrightarrow (x,\ n) \in X \Longrightarrow us = vs\ (mod\ \mathcal{U}\ /\widehat{\ }\ n) \Longrightarrow \tau\ (x,\ n)\ us = \upsilon\ (x,\ n)\ vs\ (mod\ \mathcal{U})$
by (*metis* (*no-types*, *lifting*) *QuotientP-mod case-prodD fun-quotient-app-mod operations-def*
qequal-valuation-def qequal-valuation-sym qequal-valuation-trans valuation-quotient-def)

lemma *valuation-mod-subdomain*:
assumes *mod*: $\tau = \upsilon\ (mod\ X \rightarrowtail \mathcal{U})$
assumes *sub*: $Y \subseteq X$
shows $\tau = \upsilon\ (mod\ Y \rightarrowtail \mathcal{U})$
proof –
have $\forall\ (x,\ n) \in X.\ \tau\ (x,\ n) = \upsilon\ (x,\ n)\ (mod\ operations\ \mathcal{U}\ n)$
using *mod*
by (*simp add*: *qequal-valuation-def valuation-quotient-mod*)
then show *?thesis*
by (*meson mod qequal-valuation-def sub subset-iff valuation-quotient-mod*)
qed

lemma *update-valuation-skipvar*:
assumes *x*: $x \notin set\ xs$
shows $\upsilon\{xs := us\}(x,\ n) = \upsilon(x,\ n)$
proof –
have *index-of x xs = None*
by (*metis index-of-is-Some nth-mem option.exhaust x*)
then show *?thesis*
by (*simp add*: *update-valuation-def*)
qed

lemma *subtracted-bound-vars*:
assumes *x*: $(x,\ n) \in X - xs`,0$
shows $n > 0 \lor x \notin set\ xs$
using *x*
by (*simp add*: *binders-as-vars-def*)

lemma *update-valuation-eq-intro*:
assumes $\tau = \upsilon\ (mod\ X \rightarrowtail \mathcal{U})$

assumes $us = vs$ (*mod* $\mathcal{U}/\widehat{\ }n$)
assumes *length* $xs = n$
shows $\tau\{xs := us\} = \upsilon\{xs := vs\}$ (*mod* $(X \cup ((xs)`,0)) \rightarrowtail \mathcal{U}$)
proof −
{
fix x :: *variable*
assume $x \in$ *set xs*
then have $\exists\ i.$ *index-of x xs* = *Some i*
by (*simp add*: *index-of-exists*)
then obtain i **where** i: *index-of x xs* = *Some i* **by** *blast*
then have $i < n$
using *assms*(*3*) *index-of-is-Some* **by** *fastforce*
then have *us-vs-eq-at-i*: *us* ! i = *vs* ! i (*mod* $\mathcal{U}$)
using *assms*(*2*) *vector-quotient-nth-mod* **by** *blast*
have $\tau\{xs := us\}(x,\ 0) = \upsilon\{xs := vs\}(x,\ 0)$ (*mod operations* $\mathcal{U}$ *0*)
apply (*auto simp add*: *update-valuation-def i*)
using *us-vs-eq-at-i*
by (*simp add*: *operations-eq-intro value-op-def*)
}
note *main* = *this*
{
fix x :: *variable*
fix n :: *nat*
assume *xn*: $(x,\ n) \in X - xs`,0$
then have *x-or-n*: $x \notin$ *set xs* $\vee\ n > 0$
using *subtracted-bound-vars* **by** *blast*
have $\tau\{xs := us\}\ (x,\ n) = \upsilon\{xs := vs\}\ (x,\ n)$ (*mod operations* $\mathcal{U}$ *n*)
by (*smt* (*verit*, *ccfv-threshold*) *DiffE assms*(*1*) *not-gr0 operations-mod prod.simps*(*2*)

update-valuation-def update-valuation-skipvar valuation-quotient-app x-or-n
xn)
}
note *simple* = *this*
show *?thesis*
apply (*simp add*: *valuation-quotient-mod qequal-valuation-def*)
using *main simple binders-as-vars-def* **by** *fastforce*
qed

lemma *valuations-empty-domain*[*simp*]: $\{\} \rightarrowtail \mathcal{U} = /\mathbf{1}\mathcal{U}$
by (*meson equals0D qequal-valuation-def qsubset-antisym-weak qsubset-weak-def*
subset-universal-singleton-weak valuation-quotient-mod)

7.2 Evaluation

context *algloc*
begin

abbreviation *Valuations* :: *variables* ⇒ *'a valuation quotient* ($\mathbb{V}$)
where $\mathbb{V}\ X \equiv (X \rightarrowtail \mathcal{U})$

```
fun eval :: nterm ⇒ 'a valuation ⇒ 'a (⟨-; -⟩) where
  eval (VarApp x ts) υ = υ (x, length ts) (§map t = ts!-. eval t υ)
| eval (AbsApp a xs ts) υ = (𝒪 !! a) (§map t = ts!i.
      (λ us. eval t (υ { a!@i(xs) := us })))

lemma eval-modulo:
  wf t ⟹
   frees t ⊆ X ⟹
   τ = υ (mod 𝕍 X) ⟹
   eval t τ = eval t υ (mod 𝒰)
proof (induct t arbitrary: τ υ X)
  case (VarApp x ts)
  have τ-eq-υ: τ = υ (mod 𝕍 X) using VarApp by auto
  thm valuation-quotient-app[OF τ-eq-υ]
  have frees (VarApp x ts) ⊆ X using VarApp by force
  then have xInX: (x, length ts) ∈ X
    using nt-free-VarApp by force
  have frees-sub: ⋀ i. i < length ts ⟹ frees (ts ! i) ⊆ X
    using VarApp.prems(2) nt-free-VarApp-arg-subset by presburger
  show ?case
    apply (auto simp add: list-unindexed-map)
    apply (rule valuation-quotient-app[where X=X])
    apply (rule τ-eq-υ)
    apply (rule xInX)
    apply (subst vector-quotient-mod)
    apply (auto simp add: qequal-vector-def)
    by (metis VarApp.hyps VarApp.prems(1) VarApp.prems(3) frees-sub nth-mem
wf-VarApp)
next
  case (AbsApp a xs ts)
  have valid: ✓a using AbsApp wf-implies-valid-abs by blast
  have len-ts: length ts = §ar (𝒮 !! a)
   by (smt (z3) AbsApp.prems(1) map-forced-get-def nt-wf.simps(2) option.case-eq-if
sig-contains-def)
  have frees: ⋀ i. i < length ts ⟹ frees (ts!i) ⊆ X ∪ (a!@i(xs))‘,0
    using AbsApp.prems(2) nt-free-ConsApp-arg-subset by auto
  show ?case
    apply simp
    apply (rule operator-appeq-intro)
    apply (rule valid-in-operators)
    using valid apply blast
    apply (simp add: len-ts)+
    apply (simp add: len-ts[symmetric])
    apply (rule operations-eq-intro)
    apply (rule AbsApp(1))
    apply force
    using AbsApp.prems(1) apply auto[1]
    apply (rule frees, simp)
```

apply (*rule update-valuation-eq-intro*)
using *AbsApp*
apply *simp*
apply *simp*
using *valid*
apply *auto*
apply (*rule length-boundvars-at*[*OF AbsApp*(*2*)])
by *simp*
qed

lemma *eval-is-fun-modulo*:
assumes *wf*: *wf t*
shows *eval t* $/\in$ $\mathbb{V}$ (*frees t*) $/\Rightarrow$ $\mathcal{U}$
using *eval-modulo*[**where** *X*=*frees t*, *OF wf*, *simplified*]
by (*simp add*: *fun-quotient-in qequal-fun-def*)

lemma *eval-closed*:
assumes *wf*: *wf t*
assumes *cl*: *closed t*
shows *eval t* τ = *eval t* υ (*mod* $\mathcal{U}$)
proof −
have *frees*: *frees t* $\subseteq$ {} **using** *cl*
by (*simp add*: *closed-def*)
show *?thesis* **by** (*rule eval-modulo*[*OF wf frees*, *simplified*])
qed

7.3 Semantical Equivalence

Two terms are semantically equivalent if for all abstraction algebras, and all valuations, they evaluate to the same value. We cannot really define this as a closed notion in HOL, as quantifying over all abstraction algebras requires quantifying over type variables, which is not possible in HOL. So we first define semantical equivalence just relative to a fixed abstraction algebra, and then relative to the base type of the abstraction algebra.

definition *sem-equiv* :: *nterm* $\Rightarrow$ *nterm* $\Rightarrow$ *bool*
where *sem-equiv s t* = ($\forall$ υ. υ $/\in$ $\mathbb{V}$ *UNIV* $\longrightarrow$ *eval s* υ = *eval t* υ (*mod* $\mathcal{U}$))

end

HOL can be extended with quantification over type variables [1], and then the notion of semantical equivalence of two terms could be defined via

semantically-equivalent s t = $\forall\,\alpha.\ \forall\ \mathfrak{A} :: \alpha$ *algebra. algloc.sem-equiv* $\mathfrak{A}$ *s t*

But all we can do here is to define semantical equivalence relative to α:

definition *semantically-equivalent* :: *'a* $\Rightarrow$ *nterm* $\Rightarrow$ *nterm* $\Rightarrow$ *bool*
where *semantically-equivalent* α *s t* = ($\forall$ $\mathfrak{A}$ :: *'a algebra. algloc.sem-equiv* $\mathfrak{A}$ *s t*)

lemma *semantically-equivalent* (α_1::*'a*) *s t* = *semantically-equivalent* (α_2::*'a*) *s t*

by (*simp add*: *semantically-equivalent-def*)

end
theory *BTerm* **imports** *Valuation*
begin

8 De Bruijn Term

8.1 De Bruijn Terms

datatype *bterm* =
FreeVar variable ‹*bterm list*›
| *BoundVar nat*
| *Abstr abstraction* ‹*bterm list*›

8.2 Unbound and Free Variables

context *sigloc* **begin**

definition *raise-indices* :: *nat set* ⇒ *nat* ⇒ *nat set* (**infixl** .↑ *80*)
where *raise-indices I m* = $\{ i + m \mid i.\ i \in I \}$

definition *lower-indices* :: *nat set* ⇒ *nat* ⇒ *nat set* (**infixl** .↓ *80*)
where I .↓ $m = \{ i - m \mid i.\ i \in I \wedge i \geq m \}$

lemma *raise-indices-0*[*simp*]: I .↑ $0 = I$
by (*simp add*: *raise-indices-def*)

lemma *lower-indices-0*[*simp*]: I .↓ $0 = I$
by (*simp add*: *lower-indices-def*)

lemma *raise-indices-mono*: $A \subseteq B \Longrightarrow A$.↑ $m \subseteq B$.↑ m
using *raise-indices-def* **by** *auto*

lemma *lower-indices-mono*: $A \subseteq B \Longrightarrow A$.↓ $m \subseteq B$.↓ m
using *lower-indices-def* **by** *auto*

lemma *raise-lower-indices-le*[*simp*]: $n \leq m \Longrightarrow I$.↑ m .↓ $n = I$.↑ $(m - n)$
by (*auto simp add*: *lower-indices-def raise-indices-def*)

lemma *raise-lower-indices-ge*[*simp*]: $n \geq m \Longrightarrow I$.↑ m .↓ $n = I$.↓ $(n - m)$
by (*auto simp add*: *lower-indices-def raise-indices-def*)

lemma *erase-bottom-indices*: $i \in I$.↓ m .↑ $m \Longrightarrow i \geq m \wedge i \in I$
using *lower-indices-def raise-indices-def* **by** *force*

lemma *undo-raise-indices*: $i \in I$.↑ $m \Longrightarrow i - m \in I$
using *raise-indices-def* **by** *fastforce*

fun *unbounds* :: *bterm* ⇒ *nat set* **where**
unbounds (*FreeVar x ts*) = (§*fold I* = {}, *t*=*ts*!-. *I* ∪ *unbounds t*)
| *unbounds* (*BoundVar i*) = {*i*}
| *unbounds* (*Abstr a ts*) = (§*fold I* = {}, *t*=*ts*!-. *I* ∪ *unbounds t*) .↓ §*v a*

lemma *unbounds-freeVar-arg*: ⋀ *i*. *i* < *length ts* ⟹ *unbounds* (*ts* ! *i*) ⊆ *unbounds* (*FreeVar x ts*)
by (*auto simp add*: *union-indexed-fold*)

lemma *unbounds-abstr*:
assumes *i*: *k* < *length ts*
shows *unbounds* (*ts* ! *k*) ⊆ *unbounds* (*Abstr a ts*) .↑ §*v a* ∪ *nats* (§*v a*)
proof −
{
fix *i* :: *nat*
assume *i*: *i* ∈ *unbounds* (*ts* ! *k*) ∧ *i* ≥ §*v a*
have *i* − §*v a* ∈ *unbounds* (*Abstr a ts*)
apply (*auto simp add*: *union-indexed-fold*)
using *assms*(*1*) *i lower-indices-def* **by** *auto*
have *i* ∈ *unbounds* (*Abstr a ts*) .↑ §*v a*
apply (*auto simp add*: *union-indexed-fold*)
by (*smt* (*verit*, *ccfv-SIG*) *CollectI UnionI assms*(*1*) *i le-add-diff-inverse2 lower-indices-def*
raise-indices-def)
}
then show *?thesis* **by** *force*
qed

fun *bfrees* :: *bterm* ⇒ *variables* **where**
bfrees (*FreeVar x ts*) =
(§*fold X* = {(*x*, *length ts*)}, *t*=*ts*!-. *X* ∪ *bfrees t*)
| *bfrees* (*BoundVar i*) = {}
| *bfrees* (*Abstr a ts*) = (§*fold X* = {}, *t*=*ts*!-. *X* ∪ *bfrees t*)

lemma *bfrees-freeVar-arg*: ⋀ *i*. *i* < *length ts* ⟹ *bfrees* (*ts* ! *i*) ⊆ *bfrees* (*FreeVar x ts*)
by (*auto simp add*: *union-indexed-fold*)

lemma *bfrees-abstr-arg*: ⋀ *i*. *i* < *length ts* ⟹ *bfrees* (*ts* ! *i*) ⊆ *bfrees* (*Abstr a ts*)
by (*auto simp add*: *union-indexed-fold*)

8.3 Wellformedness

fun *bwf* :: *bterm* ⇒ *bool* **where**
bwf (*FreeVar x ts*) = (∀ *t*=*ts*!-. *bwf t*)
| *bwf* (*BoundVar i*) = *True*
| *bwf* (*Abstr a ts*) = (✓*a* ∧ §*a a* = *length ts* ∧
(∀ *t*=*ts*!*i*. *bwf t* ∧ *unbounds t* ∩ *nats* (§*v a*) ⊆ *a*!♮*i*))

lemma *bwf-freeVar-arg*: *bwf* (*FreeVar x ts*) $\Longrightarrow$ *i* < *length ts* $\Longrightarrow$ *bwf* (*ts*!*i*)
  by *auto*

lemma *bwf-abstr-arg*: *bwf* (*Abstr a ts*) $\Longrightarrow$ *i* < *length ts* $\Longrightarrow$ *bwf* (*ts*!*i*)
  by (*simp add*: *list-indexed-forall-def*)

lemma *unbounds-bwf-abstr*:
  assumes *i*: *k* < *length ts*
  assumes *wf*: *bwf* (*Abstr a ts*)
  shows *unbounds* (*ts* ! *k*) ⊆ *unbounds* (*Abstr a ts*) .↑ §*v a* ∪ *a*!♮*k*
proof –
  {
    fix *i* :: *nat*
    assume *i*: *i* ∈ *unbounds* (*ts* ! *k*) ∧ *i* ≥ §*v a*
    have *i* − §*v a* ∈ *unbounds* (*Abstr a ts*)
      apply (*auto simp add*: *union-indexed-fold*)
      using *assms*(*1*) *i lower-indices-def* **by** *auto*
    have *i* ∈ *unbounds* (*Abstr a ts*) .↑ §*v a*
      apply (*auto simp add*: *union-indexed-fold*)
        by (*smt* (*verit*, *ccfv-SIG*) *CollectI UnionI assms*(*1*) *i le-add-diff-inverse2*
lower-indices-def
          raise-indices-def)
  }
  note *left* = *this*
  {
    fix *i* :: *nat*
    assume *i*: *i* ∈ *unbounds* (*ts* ! *k*) ∧ *i* < §*v a*
    have *unbounds* (*ts*!*k*) ∩ *nats* (§*v a*) ⊆ *a*!♮*k* **using** *wf*
      by (*simp add*: *assms*(*1*) *list-indexed-forall-def*)
    with *i* **have** *i'*: *i* ∈ *a*!♮*k* **by** (*simp add*: *in-mono*)
  }
  note *right* = *this*
  show *?thesis* **using** *left right* **by** *fastforce*
qed

lemma *upper-bound-unbounds-abstr-arg*:
  assumes *i*:*i* ∈ ⋃ { *unbounds t* | *t*. *t* ∈ *set ts* }
  shows *i* ∈ *unbounds* (*Abstr a ts*) .↑ §*v a* ∪ (*nats* (§*v a*))
proof –
  have *conv*: ⋃ { *unbounds t* | *t*. *t* ∈ *set ts* } = ⋃ { *unbounds* (*ts*!*k*) | *k*. *k* <
length ts }
    by (*metis in-set-conv-nth*)
  with *i* **have** *i* ∈ ⋃ { *unbounds* (*ts*!*k*) | *k*. *k* < *length ts* }
    by *auto*
  then have *i* ∈ ⋃ { *unbounds* (*Abstr a ts*).↑ §*v a* ∪ (*nats* (§*v a*)) | *k*. *k* < *length*
ts }
    using *unbounds-abstr* **by** *fastforce*
  then show *?thesis* **by** *blast*
qed

lemma *upper-bound-erased-unbounds-abstr-arg*:
  assumes $i{:}i \in \bigcup$ { *unbounds t* | *t*. *t* ∈ *set ts* } .↓ §*v a* .↑ §*v a*
  shows *i* ∈ *unbounds* (*Abstr a ts*) .↑ §*v a*
proof –
  have *i1*: *i* ∈ *unbounds* (*Abstr a ts*) .↑ §*v a* ∪ (*nats* (§*v a*))
    by (*meson erase-bottom-indices i upper-bound-unbounds-abstr-arg*)
  have *i2*: *i* ≥ §*v a*
    using *erase-bottom-indices i* **by** *blast*
  from *i1 i2* **show** *?thesis* **by** *fastforce*
qed

end

8.4 Environments

type-synonym *'a env* = *nat* ⇒ *'a*

definition *update-env* :: *'a env* ⇒ *nat* ⇒ *nat set* ⇒ *'a list* ⇒ *'a env*
  (-↑-{- := -} [*1000*, *51*, *51*, *51*] *1000*)
where
  update-env env m js xs = (λ *j*.
    (*case index-of j* (*sorted-list js*) *of*
      Some i ⇒ *xs*!*i*
    | *None* ⇒ *env* (*j* − *m*)))

abbreviation *raise-env* :: *'a env* ⇒ *nat* ⇒ *'a env*
  (-↑-{} [*1000*, *51*] *1000*)
where
  env ↑ *m* {} ≡ *env* ↑ *m* { {} := [] }

lemma *env-app-noupdate*: *finite js* ⟹ *j* ∉ *js* ⟹ *env* ↑ *m* { *js* := *xs* } *j* = *env* ↑ *m* {} *j*
  by (*simp add*: *update-env-def no-index-sorted-list*)

lemma *env-raised-app*: *env* ↑ *m* {} *j* = *env* (*j* − *m*)
  by (*simp add*: *no-index-sorted-list update-env-def*)

lemma *env-app-update*:
  finite js ⟹ *j* ∈ *js* ⟹ *env* ↑ *m* { *js* := *us* } *j* = *us* ! *the* (*index-of j* (*sorted-list js*))
  by (*metis index-of-is-None option.case-eq-if set-sorted-list update-env-def*)

context *algloc* **begin**

definition *env-quotient* :: *nat set* ⇒ (*'a env*) *quotient* (𝔼)
  where *env-quotient I* = (/≡*I* /⇒ 𝒰)

lemma *env-subset*: *A* ⊆ *B* ⟹ 𝔼 *B* /≤ 𝔼 *A*

by (*simp add*: *env-quotient-def fun-quotient-subset-weak-intro set-quotient-subset-weak*)

lemma *env-subset-mod*: $A \subseteq B \Longrightarrow env1 = env2\ (mod\ \mathbb{E}\ B) \Longrightarrow env1 = env2$ $(mod\ \mathbb{E}\ A)$
by (*meson env-subset qsubset-mod-weak*)

lemma *env-app*: $i \in I \Longrightarrow env1 = env2\ (mod\ \mathbb{E}\ I) \Longrightarrow env1\ i = env2\ i\ (mod\ \mathcal{U})$
by (*simp add*: *env-quotient-def fun-quotient-app-mod set-quotient-mod*)

lemma *env-mod*: $(env1 = env2\ (mod\ \mathbb{E}\ I)) = (\forall\ i \in I.\ env1\ i = env2\ i\ (mod\ \mathcal{U}))$
by (*smt* (*verit*, *best*) *env-quotient-def fun-quotient-mod qequal-fun-def set-quotient-mod*)

8.5 Evaluation

fun *beval* :: *bterm* $\Rightarrow$ *'a valuation* $\Rightarrow$ *'a env* $\Rightarrow$ *'a* $(\langle$-; -, -$\rangle)$ **where**
beval (*FreeVar x ts*) υ *env* = υ (*x*, *length ts*) (§*map t* = *ts*!-. *beval t* υ *env*)
| *beval* (*BoundVar i*) υ *env* = *env i*
| *beval* (*Abstr a ts*) υ *env* = ($\mathcal{O}$!!*a*) (§*map t* = *ts*!*i*.
(λ *us*. *beval t* υ (*env* $\uparrow$ §υ *a* {*a*!$\natural$*i* := *us*})))

lemma *beval-modulo*:
bwf t $\Longrightarrow$
bfrees t $\subseteq X \Longrightarrow$
$\tau = \upsilon\ (mod\ \mathbb{V}\ X) \Longrightarrow$
unbounds t $\subseteq I \Longrightarrow$
$env1 = env2\ (mod\ \mathbb{E}\ I) \Longrightarrow$
beval t τ *env1* = *beval t* υ *env2* (*mod* $\mathcal{U}$)
proof (*induct t arbitrary*: *env1 env2 I*)
case (*FreeVar x ts*)
have (*x*, *length ts*) $\in$ *bfrees* (*FreeVar x ts*)
by (*simp add*: *union-indexed-fold*)
then have *x*: (*x*, *length ts*) $\in X$
using *FreeVar.prems*(*2*) **by** *blast*
have *bfrees*: $\bigwedge$ *i*. *i* < *length ts* $\Longrightarrow$ *bfrees* (*ts* ! *i*) $\subseteq X$ **using** *bfrees-freeVar-arg*
using *FreeVar.prems*(*2*) **by** *blast*
show *?case* **apply** *simp*
apply (*rule valuation-quotient-app*[**where** *X*=*X*])
using *FreeVar* **apply** *simp*
apply (*simp add*: *x*)
apply (*auto simp add*: *vector-quotient-mod qequal-vector-def*)
apply (*rule FreeVar*(*1*))
apply *simp*
apply (*erule bwf-freeVar-arg*[*OF FreeVar.prems*(*1*)])
apply (*erule bfrees*)
using *FreeVar* **apply** *simp*
apply *force*
apply (*rule env-subset-mod*[**where** *B* = *I*])
apply (*meson FreeVar.prems*(*4*) *sigloc.unbounds-freeVar-arg subset-trans*)
using *FreeVar* **apply** *simp*

```
    done
next
  case (BoundVar i)
  have i: i ∈ I
    using BoundVar.prems(4) by auto
  show ?case
    apply simp
    using BoundVar i env-app
    by force
next
  case (Abstr a ts)
  have valid: ✓ a
    using Abstr.prems(1) by force
  have length-ts: §a a = length ts
    using Abstr.prems(1) by force
  show ?case
    apply simp
    apply (rule operator-appeq-intro[of 𝒪 !! a 𝒰 𝒮 !! a])
    using valid valid-in-operators apply force
    apply (simp add: length-ts)+
    apply (auto simp add: operations-mod)
    apply (rule Abstr(1))
    apply simp
    using Abstr.prems(1) sigloc.bwf-abstr-arg apply blast
    using Abstr.prems(2) bfrees-abstr-arg apply blast
    using Abstr.prems(3) apply blast
    apply (rule unbounds-bwf-abstr[where a=a], simp)
    using Abstr apply simp
    apply (auto simp add: env-mod union-unindexed-fold)
    apply (subst env-app-noupdate[where env=env1])
    using length-ts finite-shape-deps valid apply presburger
     apply (metis (no-types, lifting) erase-bottom-indices length-ts linorder-not-le
shape-deps-valence)
    apply (subst env-app-noupdate[where env=env2])
    using length-ts finite-shape-deps valid apply presburger
     apply (metis (no-types, lifting) erase-bottom-indices length-ts linorder-not-le
shape-deps-valence)
    apply (simp add: env-raised-app)
    apply (rule env-app[where I=I])
    using upper-bound-unbounds-abstr-arg erase-bottom-indices
    apply (metis (no-types, lifting) Abstr.prems(4) subset-eq undo-raise-indices
           upper-bound-erased-unbounds-abstr-arg)
    using Abstr apply simp
    apply (subst env-app-update[where env=env1])
    using finite-shape-deps length-ts valid apply presburger
    apply simp
    apply (subst env-app-update[where env=env2])
    using finite-shape-deps length-ts valid apply presburger
    apply simp
```

apply (*rule vector-quotient-nth-mod*)
apply *auto* **using** *valid*
using *length-ts sigloc.finite-shape-deps upper-bound-index-sorted-list* **by** *presburger*
qed

end

end

References

[1] T. F. Melham. The hol logic extended with quantification over type variables. https://doi.org/10.1007/BF01383982, 1993.

[2] S. Obua. Abstraction logic. https://doi.org/10.47757/abstraction.logic.2, November 2021.

[3] S. Obua. Philosophy of abstraction logic. https://doi.org/10.47757/pal.2, December 2021.