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 $nats :: nat \Rightarrow nat\ set$ **where**
 $nats\ n = \{..< n\}$

lemma *finite-nats*[*iff*]: $finite\ (nats\ n)$
 ⟨*proof*⟩

lemma *nats-elem*[*simp*]: $(d \in nats\ n) = (d < n)$
 ⟨*proof*⟩

lemma *nats-0*[*simp*]: $nats\ 0 = \{\}$
 ⟨*proof*⟩

lemma *card-nats*[*simp*]: $card\ (nats\ n) = n$
 ⟨*proof*⟩

lemma *nats-eq-nats*[*simp*]: $(nats\ n = nats\ m) = (n = m)$
 ⟨*proof*⟩

lemma *Max-nats*: $n > 0 \Longrightarrow 1 + Max\ (nats\ n) = n$
 ⟨*proof*⟩

1.2 Lists

1.2.1 Tools for Indices

lemma *nats-length-nths*:
 assumes $A \subseteq nats\ (length\ xs)$
 shows $length\ (nths\ xs\ A) = card\ A$
⟨*proof*⟩

fun $index\text{-}of :: 'a \Rightarrow 'a\ list \Rightarrow nat\ option$ **where**
 $index\text{-}of\ x\ [] = None$
| $index\text{-}of\ x\ (a\#as) = (if\ x = a\ then\ Some\ 0\ else$
 $(case\ index\text{-}of\ x\ as\ of$
 $None \Rightarrow None$
 $|\ Some\ i \Rightarrow Some\ (Suc\ i)))$

lemma *index-of-head*: $index\text{-}of\ x\ (x\ \#\ xs) = Some\ 0$
 ⟨*proof*⟩

lemma *index-of-exists*: $x \in set\ xs \Longrightarrow \exists\ i.\ index\text{-}of\ x\ xs = Some\ i$
⟨*proof*⟩

lemma *index-of-is-None*: $index\text{-}of\ x\ xs = None \Longrightarrow x \notin set\ xs$
⟨*proof*⟩

lemma *index-of-is-Some*: $index\text{-}of\ x\ xs = Some\ i \Longrightarrow i < length\ xs \wedge xs!i = x$
⟨*proof*⟩

definition *shift-index* $:: nat \Rightarrow (nat \Rightarrow 'a) => (nat => 'a)$ **where**
$shift\text{-}index\ d\ f\ x = f\ (x + d)$

lemma *shift-index-0*[*simp*]: $shift\text{-}index\ 0 = id$
⟨*proof*⟩

lemma *shift-index-acc-append*[*simp*]:
$shift\text{-}index\ d\ (\lambda i\ acc\ x.\ acc\ @\ [f\ i\ x]) = (\lambda\ i\ acc\ x.\ acc\ @\ [shift\text{-}index\ d\ f\ i\ x])$
⟨*proof*⟩

lemma *shift-index-gather*:
$shift\text{-}index\ d\ (\lambda i\ acc\ x.\ g\ (f\ i\ x)\ acc) = (\lambda i\ acc\ x.\ g\ (shift\text{-}index\ d\ f\ i\ x)\ acc)$
⟨*proof*⟩

lemma *shift-index-applied-twice*[*simp*]:
$shift\text{-}index\ a\ (shift\text{-}index\ b\ f) = shift\text{-}index\ (a+b)\ f$
⟨*proof*⟩

lemma *shift-index-unindexed*[*simp*]: $shift\text{-}index\ d\ (\lambda\ i.\ F) = (\lambda\ i.\ F)$
⟨*proof*⟩

definition *sorted-list* $:: nat\ set \Rightarrow nat\ list$
where $sorted\text{-}list\ js = (THE\ l.\ sorted\ l \wedge distinct\ l \wedge set\ l = js)$

lemma *set-sorted-list*: $finite\ js \Longrightarrow set\ (sorted\text{-}list\ js) = js$
⟨*proof*⟩

lemma *sorted-sorted-list*: $finite\ js \Longrightarrow sorted\ (sorted\text{-}list\ js)$
⟨*proof*⟩

lemma *distinct-sorted-list*: $finite\ js \Longrightarrow distinct\ (sorted\text{-}list\ js)$
⟨*proof*⟩

lemma *sorted-list-intro*: $sorted\ l \wedge distinct\ l \wedge set\ l = js \Longrightarrow sorted\text{-}list\ js = l$
⟨*proof*⟩

lemma *sorted-list-nats*: $sorted\text{-}list\ (nats\ n) = [0\ ..< n]$
⟨*proof*⟩

lemma *no-index-sorted-list*:
assumes *finite*: $finite\ js$
assumes *j*: $j \notin js$

shows *index-of j* (*sorted-list js*) = *None*
⟨*proof*⟩

lemma *index-sorted-list*:
assumes *finite*: *finite js*
assumes *j*: $j \in js$
shows $\exists$ *i*. *index-of j* (*sorted-list js*) = *Some i*
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⟹ *x* = *ys*!*i* ⟹ *P i x* = *Q i x*
shows (∀ *x* = *xs*!*i*. *P i x*) = (∀ *y* = *ys*!*i*. *Q i y*)
⟨*proof*⟩

lemma *size-nth*[*termination-simp*]: *i* < *length ts* ⟹ *size* (*ts* ! *i*) < *Suc* (*size-list size ts*)
⟨*proof*⟩

lemma *list-indexed-forall-empty*[*simp*]: *list-indexed-forall* [] *f* = *True*
⟨*proof*⟩

lemma *list-indexed-forall-cons*[*simp*]:
list-indexed-forall (*x*#*xs*) *f* = (*f 0 x* ∧ *list-indexed-forall xs* (*shift-index 1 f*))
⟨*proof*⟩

1.2.3 Indexed Fold

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

syntax

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

translations
§*fold a* = *a0*, *x* = *xs*!*i*. *F* ⇌ *CONST list-indexed-fold* (λ *i x a*. *F*) *xs a0*

lemma *list-indexed-fold-empty*[*simp*]: *list-indexed-fold f* [] *y* = *y*
⟨*proof*⟩

lemma *list-indexed-fold-cong*[*fundef-cong*]:
assumes *xs* = *ys*
assumes ⋀*i a x*. *i* < *length ys* ⟹ *x* = *ys*!*i* ⟹ *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*)
⟨*proof*⟩

lemma *list-indexed-fold-eq*:
assumes ⋀*i a x*. *i* < *length xs* ⟹ *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*)
⟨*proof*⟩

lemma *list-unindexed-forall*[*simp*]: (∀ *x* = *xs*!*i*. *P x*) = (∀ *x* ∈ *set xs*. *P x*)
⟨*proof*⟩

lemma *fold-zip-interval-shift*:
i + *length xs* = *j* ⟹
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*⟩

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*⟩

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*⟩

lemma *list-unindexed-fold*:
(§*fold a* = *a0*, *x* = *xs*!*i*. *F x a*) = *fold F xs a0*
⟨*proof*⟩

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 $\bigwedge$*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*)
⟨*proof*⟩

lemma [*9*, *49*] = (§*map x* = [*3* :: *nat*, *7*]!*i*. *x* ∗ *x*)
⟨*proof*⟩

lemma *list-indexed-map-empty*[*simp*]: *list-indexed-map F* [] = []
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *list-indexed-map-single*[*simp*]: *list-indexed-map F* [*a*] = [*F 0 a*]
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *map-cons*: *map f* (*a*#*as*) = *f a* # (*map f as*)
⟨*proof*⟩

lemma *map-snoc*: *map f* (*as*@[*a*]) = (*map f as*) @ [*f a*]
⟨*proof*⟩

lemma *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*)]
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *list-indexed-map-alt*:
(§*map x* = *xs*!*i*. *F i x*) = *map* (λ *i*. *F i* (*xs*!*i*)) [*0* ..< *length xs*]
⟨*proof*⟩

lemma *list-unindexed-map*: (§*map x* = *xs*!*i*. *F x*) = *map F xs*
⟨*proof*⟩

lemma *list-indexed-map-length*[*simp*]: *length* (§*map x* = *xs*!*i*. *F i x*) = *length xs*
⟨*proof*⟩

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

1.2.5 Fold over Indexed Map

lemma *fold-indexed-map*: (§*fold acc* = *a*, *x* = *xs*!*i*. *g* (*F i x*) *acc*) = *fold g* (§*map x*=*xs*!*i*. *F i x*) *a*
⟨*proof*⟩

lemma *fold-union*: *fold* (λ*a b*. *b* $\cup$ *a*) *xs a0* = *a0* $\cup$ $\bigcup$ (*set xs*)
⟨*proof*⟩

lemma *Un-indexed-nats*: ($\bigcup$*i*$\in${*0*..<*n*::*nat*}. *F i*) = $\bigcup$ { *F i* | *i*. *i* < *n* }
⟨*proof*⟩

lemma *union-indexed-fold*:
(§*fold X* = *X0*, *x* = *xs*!*i*. *X* $\cup$ *F i x*) = *X0* $\cup$ $\bigcup$ { *F i* (*xs*!*i*) | *i*. *i* < *length xs* }
⟨*proof*⟩

lemma *union-unindexed-fold*:
(§*fold X* = *X0*, *x* = *xs*!-. *X* $\cup$ *F x*) = *X0* $\cup$ $\bigcup$ { *F x* | *x*. *x* $\in$ *set xs* }
⟨*proof*⟩

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* []
⟨*proof*⟩

2.2 Shapes are Wellformed Preshapes

typedef *shape* = {*s* . *wellformed-preshape s*} **morphisms** *Preshape Shape*
⟨*proof*⟩

lemma *wellformed-Preshape*[*iff*]: *wellformed-preshape* (*Preshape s*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *preshape-valence*: *preshape-alldeps* (*Preshape s*) = *nats* (*shape-valence s*)
⟨*proof*⟩

lemma *empty-deps-Shape-valence*:
preshape-alldeps s = {} ⟹ (*shape-valence* (*Shape s*) = *0*)
⟨*proof*⟩

lemma *nonempty-deps-Shape-valence*:
assumes *wf*: *wellformed-preshape s*

assumes *nonemtpy*: *preshape-alldeps* $s \neq \{\}$
shows *shape-valence* (*Shape* s) = *1* + *Max* (*preshape-alldeps* s)
⟨*proof*⟩

lemma *Shape-arity*[*intro*]: *wellformed-preshape* $s \Longrightarrow$ *shape-arity* (*Shape* s) = *length* s
⟨*proof*⟩

2.4 Dependencies

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

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

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

lemma *shape-deps-in-alldeps*:
$i <$ *shape-arity* $s \Longrightarrow$ *shape-deps* s $i \subseteq$ *preshape-alldeps* (*Preshape* s)
⟨*proof*⟩

lemma $i <$ *shape-arity* $s \Longrightarrow$ *shape-deps* s $i \subseteq$ *nats* (*shape-valence* s)
⟨*proof*⟩

lemma *shape-valence-deps*:
assumes d: $d <$ *shape-valence* s
shows $\exists$ $i <$ *shape-arity* s. $d \in$ *shape-deps* s i
⟨*proof*⟩

lemma *shape-deps-valence*:
assumes i: $i <$ *shape-arity* $s \wedge d \in$ *shape-deps* s i
shows $d <$ *shape-valence* s
⟨*proof*⟩

lemma *nats-shape-valence-is-union*:
nats (*shape-valence* s) = $\bigcup$ { *shape-deps* s i | i . $i <$ *shape-arity* s }
⟨*proof*⟩

lemma *zero-arity-valence*: *shape-arity* s = *0* $\Longrightarrow$ *shape-valence* s = *0*
⟨*proof*⟩

lemma *zero-valence-deps*: $i <$ *shape-arity* $s \Longrightarrow$ *shape-valence* s = *0* $\Longrightarrow$ *shape-deps* s i = {}
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Preshape-Shape*[*intro*]: *wellformed-preshape s* $\Longrightarrow$ *Preshape* (*Shape s*) = *s*
⟨*proof*⟩

lemma *value-Preshape*[*simp*]: *Preshape value-shape* = []
⟨*proof*⟩

lemma *value-shape-arity*[*simp*]: §*ar value-shape* = *0*
⟨*proof*⟩

2.5.2 *unop-shape*

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

lemma *wf-unop-preshape*: *wellformed-preshape* [{}]
⟨*proof*⟩

lemma *unop-Preshape*[*simp*]: *Preshape* (*unop-shape*) = [{}]
⟨*proof*⟩

lemma *unop-shape-arity*[*simp*]: §*ar unop-shape* = *1*
⟨*proof*⟩

lemma *unop-shape-valence*[*simp*]: §*val unop-shape* = *0*
⟨*proof*⟩

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

2.5.3 *binop-shape*

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

lemma *wf-binop-preshape*: *wellformed-preshape* [{}, {}]
⟨*proof*⟩

lemma *binop-Preshape*[*simp*]: *Preshape* (*binop-shape*) = [{}, {}]

⟨*proof*⟩

lemma *binop-shape-arity*[*simp*]: §*ar binop-shape* = *Suc* (*Suc 0*)
⟨*proof*⟩

lemma *binop-shape-valence*[*simp*]: §*val binop-shape* = *0*
⟨*proof*⟩

lemma *binop-shape-deps-0*[*simp*]: *binop-shape.*♮*0* = {}
⟨*proof*⟩

lemma *binop-shape-deps-1*[*simp*]: *binop-shape.*♮*1* = {}
⟨*proof*⟩

2.5.4 *operator-shape*

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

lemma *wf-operator-preshape*: *wellformed-preshape* [{*0*}]
⟨*proof*⟩

lemma *operator-Preshape*[*simp*]: *Preshape* (*operator-shape*) = [{*0*}]
⟨*proof*⟩

lemma *operator-shape-arity*[*simp*]: §*ar operator-shape* = *Suc 0*
⟨*proof*⟩

lemma *operator-shape-valence*[*simp*]: §*val operator-shape* = *Suc 0*
⟨*proof*⟩

lemma *operator-shape-deps-0*[*iff*]: *operator-shape.*♮*0* = {*0*}
⟨*proof*⟩

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* $\neq$ *abstr-implies*
⟨*proof*⟩

lemma *noteq-abstr-implies-forall*[*simp*]: *abstr-implies* $\neq$ *abstr-forall*
⟨*proof*⟩

lemma *noteq-abstr-true-forall*[*simp*]: *abstr-true* $\neq$ *abstr-forall*
⟨*proof*⟩

3.2 Signatures

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

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

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

definition *extends-sig* :: *signature* $\Rightarrow$ *signature* $\Rightarrow$ *bool* (**infix** $\succeq$ *50*) **where**
extends-sig T S = ($\forall$ *a*. *S a* = *None* $\vee$ *T a* = *S a*)

lemma *has-shape-extends*: *T* $\succeq$ *S* $\Longrightarrow$ *has-shape S a s* $\Longrightarrow$ *has-shape T a s*
⟨*proof*⟩

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

lemma *has-shape-sig-contains*: *has-shape sig a s* $\Longrightarrow$ *sig-contains sig a* (§*val s*) (§*ar s*)
⟨*proof*⟩

lemma *has-shape-get*: *has-shape sig a s* $\Longrightarrow$ *sig* !! *a* = *s*
⟨*proof*⟩

lemma *extends-sig-contains*: *V* $\succeq$ *U* $\Longrightarrow$ *sig-contains U a val ar* $\Longrightarrow$ *sig-contains V a val ar*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *deduction-sig-implies*[*iff*]: *has-shape* $\mathfrak{D}$ *abstr-implies binop-shape*
⟨*proof*⟩

lemma *deduction-sig-forall*[*iff*]: *has-shape* $\mathfrak{D}$ *abstr-forall operator-shape*
⟨*proof*⟩

lemma *deduction-sig-contains-true*[*iff*]: *sig-contains* $\mathfrak{D}$ *abstr-true 0 0*
⟨*proof*⟩

lemma *deduction-sig-contains-implies*[*iff*]: *sig-contains* $\mathfrak{D}$ *abstr-implies 0* (*Suc* (*Suc 0*))
⟨*proof*⟩

lemma *deduction-sig-contains-forall*[*iff*]: *sig-contains* $\mathfrak{D}$ *abstr-forall* (*Suc 0*) (*Suc 0*)
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *equiv-Field*:
assumes *equiv A r*
shows *Field r* = *A*

$\langle proof \rangle$

lemma *equiv-QField-Rel*: $equiv\ (QField\ q)\ (Rel\ q)$
 $\langle proof \rangle$

definition *qin* :: $'a \Rightarrow 'a\ quotient \Rightarrow bool$ (**infix** $'/\in\ 50$) **where**
 $(a\ /\in\ q) = (a \in QField\ q)$

abbreviation *qnin* :: $'a \Rightarrow 'a\ quotient \Rightarrow bool$ (**infix** $'/\notin\ 50$) **where**
 $(a\ /\notin\ q) \equiv (a \in QField\ q)$

4.2 Equality Modulo

definition *qequals* :: $'a \Rightarrow 'a \Rightarrow 'a\ quotient \Rightarrow bool$ $(\text{-} = \text{-}\ '(mod\ \text{-}')\ [51,\ 51,\ 0]\ 50)$ **where**
 $(a = b\ (mod\ q)) = ((a,\ b) \in Rel\ q)$

abbreviation *qnequals* :: $'a \Rightarrow 'a \Rightarrow 'a\ quotient \Rightarrow bool$ $(\text{-} \neq \text{-}\ '(mod\ \text{-}')\ [51,\ 51,\ 0]\ 50)$ **where**
 $(a \neq b\ (mod\ q)) \equiv \neg\ (a = b\ (mod\ q))$

lemma *qin-mod*: $(a\ /\in\ q) = (a = a\ (mod\ q))$
 $\langle proof \rangle$

lemma *qequals-in*: $a = b\ (mod\ q) \Longrightarrow a\ /\in\ q \wedge b\ /\in\ q$
 $\langle proof \rangle$

lemma *qequals-sym*: $a = b\ (mod\ q) \Longrightarrow b = a\ (mod\ q)$
 $\langle proof \rangle$

lemma *qequals-trans*: $a = b\ (mod\ q) \Longrightarrow b = c\ (mod\ q) \Longrightarrow a = c\ (mod\ q)$
 $\langle proof \rangle$

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** $'/\leq\ 50$) **where**
 $(p\ /\leq\ q) = (\forall\ x\ y.\ x = y\ (mod\ p) \longrightarrow x = y\ (mod\ q))$

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*: $p\ /\subseteq\ q \Longrightarrow p\ /\sqsubseteq\ q$
 $\langle proof \rangle$

lemma *qsubset-strong-implies-weak*: $p\ /\subseteq\ q \Longrightarrow p\ /\leq\ q$
⟨*proof*⟩

lemma *qsubset-bishop-implies-weak*: $p\ /\sqsubseteq\ q \Longrightarrow p\ /\leq\ q$
⟨*proof*⟩

lemma *qsubset-QField-strong*: $p\ /\subseteq\ q \Longrightarrow \mathit{QField}\ p \subseteq \mathit{QField}\ q$
⟨*proof*⟩

lemma *qsubset-QField-weak*: $p\ /\leq\ q \Longrightarrow \mathit{QField}\ p \subseteq \mathit{QField}\ q$
⟨*proof*⟩

lemma *qsubset-QField-bishop*: $p\ /\sqsubseteq\ q \Longrightarrow \mathit{QField}\ p \subseteq \mathit{QField}\ q$
⟨*proof*⟩

lemma *qubseteq-refl-strong*[*iff*]: $q\ /\subseteq\ q$
⟨*proof*⟩

lemma *qubseteq-refl-bishop*[*iff*]: $q\ /\sqsubseteq\ q$
⟨*proof*⟩

lemma *qubseteq-refl-weak*[*iff*]: $q\ /\leq\ q$
⟨*proof*⟩

lemma *qsubset-trans-strong*: $p\ /\subseteq\ q \Longrightarrow q\ /\subseteq\ r \Longrightarrow p\ /\subseteq\ r$
⟨*proof*⟩

lemma *qsubset-trans-bishop*: $p\ /\sqsubseteq\ q \Longrightarrow q\ /\sqsubseteq\ r \Longrightarrow p\ /\sqsubseteq\ r$
⟨*proof*⟩

lemma *qsubset-trans-weak*: $p\ /\leq\ q \Longrightarrow q\ /\leq\ r \Longrightarrow p\ /\leq\ r$
⟨*proof*⟩

lemma *qsubset-antisym-weak*: $p\ /\leq\ q \Longrightarrow q\ /\leq\ p \Longrightarrow p = q$
⟨*proof*⟩

lemma *qsubset-antisym-bishop*: $p\ /\sqsubseteq\ q \Longrightarrow q\ /\sqsubseteq\ p \Longrightarrow p = q$
⟨*proof*⟩

lemma *qsubset-antisym-strong*: $p\ /\subseteq\ q \Longrightarrow q\ /\subseteq\ p \Longrightarrow p = q$
⟨*proof*⟩

lemma *qsubset-mod-weak*: $x = y\ (\mathit{mod}\ q) \Longrightarrow q\ /\leq\ p \Longrightarrow x = y\ (\mathit{mod}\ p)$
⟨*proof*⟩

lemma *qsubset-mod-bishop*: $x = y\ (\mathit{mod}\ q) \Longrightarrow q\ /\sqsubseteq\ p \Longrightarrow x = y\ (\mathit{mod}\ p)$
⟨*proof*⟩

lemma *qsubset-mod-strong*: $x = y\ (\mathit{mod}\ q) \Longrightarrow q\ /\subseteq\ p \Longrightarrow x = y\ (\mathit{mod}\ p)$

⟨*proof*⟩

4.4 Equivalence Classes

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

lemma *qequals-implies-equal-qclasses*: $a = b$ (*mod* q) $\Longrightarrow a$ /% $q = b$ /% q
⟨*proof*⟩

lemma *empty-qclass*: (a /% q = {}) = (¬ (a /∈ q))
⟨*proof*⟩

lemma *qclass-elems*: ($b \in a$ /% q) = ($a = b$ (*mod* q))
⟨*proof*⟩

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* ∧ *eq y y*
⟨*proof*⟩

lemma *QuotientP-equiv*:
assumes *symp eq*
assumes *transp eq*
shows *equiv* { x . *eq x x*} { (x, y) . *eq x y* }
⟨*proof*⟩

lemma *QuotientP-Rel*: *symp eq* $\Longrightarrow$ *transp eq* $\Longrightarrow$ *Rel* (*QuotientP eq*) = { (x, y) . *eq x y* }
⟨*proof*⟩

lemma *QuotientP-mod*: *symp eq* $\Longrightarrow$ *transp eq* $\Longrightarrow$ ($x = y$ (*mod QuotientP eq*)) = (*eq x y*)
⟨*proof*⟩

lemma *QuotientP-in*: *symp eq* $\Longrightarrow$ *transp eq* $\Longrightarrow$ (x /∈ *QuotientP eq*) = *eq x x*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *qequal-set-trans*: *transp* (*qequal-set I*)

$\langle proof \rangle$

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

lemma *set-quotient-Rel*: $Rel(/\equiv U) = \{\ (x,\ y)\ .\ x \in U \wedge x = y\ \}$
$\langle proof \rangle$

lemma *set-quotient-mod*: $(x = y\ (mod\ /\equiv U)) = (x \in U \wedge x = y)$
$\langle proof \rangle$

lemma *set-quotient-in*: $(x\ /\in\ /\equiv U) = (x \in U)$
$\langle proof \rangle$

lemma *set-quotient-subset-strong*: $(/\equiv U\ /\subseteq\ /\equiv V) = (U \subseteq V)$
$\langle proof \rangle$

lemma *set-quotient-subset-weak*: $(/\equiv U\ /\leq\ /\equiv V) = (U \subseteq V)$
$\langle proof \rangle$

lemma *set-quotient-subset-bishop*: $(/\equiv U\ /\sqsubseteq\ /\equiv V) = (U \subseteq V)$
$\langle proof \rangle$

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*: $Rel\ /\emptyset = \{\}$
$\langle proof \rangle$

lemma *empty-quotient-mod*: $\neg\ (x = y\ (mod\ /\emptyset))$
$\langle proof \rangle$

lemma *empty-quotient-in*: $\neg\ (x\ /\in\ /\emptyset)$
$\langle proof \rangle$

lemma *univ-quotient-Rel*: $Rel\ /\mathcal{U} = Id$
$\langle proof \rangle$

lemma *univ-quotient-in*: $x\ /\in\ /\mathcal{U}$
$\langle proof \rangle$

lemma *univ-quotient-mod*: $(x = y\ (mod\ /\mathcal{U})) = (x = y)$
$\langle proof \rangle$

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* $U\ y\ x$
⟨*proof*⟩

lemma *qequal-singleton-trans*:
qequal-singleton $U\ x\ y \Longrightarrow$ *qequal-singleton* $U\ y\ z \Longrightarrow$ *qequal-singleton* $U\ x\ z$
⟨*proof*⟩

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

lemma *singleton-quotient-Rel*: $Rel\ (/\mathbf{1}U) = \{\ (x,\ y).\ qequal\text{-}singleton\ U\ x\ y\ \}$
⟨*proof*⟩

lemma *singleton-quotient-mod*[*simp*]: $(x = y\ (mod\ /\mathbf{1}U)) = (x \in U \wedge y \in U)$
⟨*proof*⟩

lemma *singleton-quotient-in*: $(x\ /{\in}\ /\mathbf{1}U) = (x \in U)$
⟨*proof*⟩

lemma *empty-singleton-quotient*[*iff*]: $/\mathbf{1}\{\} = /\emptyset$
⟨*proof*⟩

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*: $/\emptyset\ /{\leq}\ q$
⟨*proof*⟩

lemma *empty-subset-singleton-quotient-bishop*: $/\emptyset\ /{\sqsubseteq}\ q$
⟨*proof*⟩

lemma *empty-subset-singleton-quotient-strong*: $/\emptyset\ /{\subseteq}\ q$
⟨*proof*⟩

lemma *same-QField-bishop*: $QField\ p = QField\ q \Longrightarrow p\ /{\sqsubseteq}\ q \Longrightarrow p = q$
⟨*proof*⟩

lemma *same-QField-strong*: $QField\ p = QField\ q \Longrightarrow p\ /{\subseteq}\ q \Longrightarrow p = q$
⟨*proof*⟩

lemma *singleton-quotient-subset-weak*: $(/\mathbf{1}U\ /{\leq}\ /\mathbf{1}V) = (U \subseteq V)$
⟨*proof*⟩

lemma *singleton-quotient-subset-bishop*: $(/\mathbf{1}\, U\ /\sqsubseteq\ /\mathbf{1}\, V) = (U \subseteq V)$
$\langle proof \rangle$

lemma *singleton-quotient-subset-strong*: $(/\mathbf{1}\, U\ /\subseteq\ /\mathbf{1}\, V) = (U = V \vee U = \{\})$
$\langle proof \rangle$

lemma *subset-universal-singleton-weak*: $q\ /\leq\ /\mathbf{1}\mathcal{U}$
$\langle proof \rangle$

lemma *subset-universal-singleton-bishop*: $(q\ /\sqsubseteq\ /\mathbf{1}\mathcal{U}) = (q = /\mathbf{1}(QField\ q))$
$\langle proof \rangle$

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

lemma *identity-QField-subset-weak*: $/\!\equiv\!(QField\ q)\ /\leq\ q$
$\langle proof \rangle$

lemma *identity-QField-subset-bishop*: $(/\!\equiv\!(QField\ q)\ /\sqsubseteq\ q) = (q = /\!\equiv\!(QField\ q))$
$\langle proof \rangle$

lemma *identity-QField-subset-strong*: $(/\!\equiv\!(QField\ q)\ /\subseteq\ q) = (q = /\!\equiv\!(QField\ q))$
$\langle proof \rangle$

lemma *qsubset-weak-neq-bishop*:
assumes *xy*: $(x{::}'a) \neq y$
shows $((/\leq) :: 'a\ quotient \Rightarrow 'a\ quotient \Rightarrow bool) \neq (/\sqsubseteq)$
$\langle proof \rangle$

lemma *qsubset-bishop-neq-strong*:
assumes *xy*: $(x{::}'a) \neq y$
shows $((/\sqsubseteq) :: 'a\ quotient \Rightarrow 'a\ quotient \Rightarrow bool) \neq (/\subseteq)$
$\langle proof \rangle$

lemma *qsubset-weak-neq-strong*:
assumes *xy*: $(x{::}'a) \neq y$
shows $((/\leq) :: 'a\ quotient \Rightarrow 'a\ quotient \Rightarrow bool) \neq (/\subseteq)$
$\langle proof \rangle$

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*: $symp\ (qequal\text{-}fun\ p\ q)\ \langle proof \rangle$

lemma *qequal-fun-trans*: $transp\ (qequal\text{-}fun\ p\ q)$

$\langle proof \rangle$

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*: $Rel\ (p\ /\Rightarrow\ q) = \{\ (f,\ g)\ .\ qequal\text{-}fun\ p\ q\ f\ g\ \}$
$\langle proof \rangle$

lemma *fun-quotient-mod*: $(f = g\ (mod\ p\ /\Rightarrow\ q)) = (qequal\text{-}fun\ p\ q\ f\ g)$
$\langle proof \rangle$

lemma *fun-quotient-in*: $(f\ /\in\ p\ /\Rightarrow\ q) = (qequal\text{-}fun\ p\ q\ f\ f)$
$\langle proof \rangle$

lemma *fun-quotient-app-in*: $f\ /\in\ p\ /\Rightarrow\ q \Longrightarrow x\ /\in\ p \Longrightarrow f\ x\ /\in\ q$
$\langle proof \rangle$

lemma *fun-quotient-app-mod*: $f = g\ (mod\ p\ /\Rightarrow\ q) \Longrightarrow x = y\ (mod\ p) \Longrightarrow f\ x = g\ y\ (mod\ q)$
$\langle proof \rangle$

lemma *fun-quotient-app-in-mod*: $f\ /\in\ p\ /\Rightarrow\ q \Longrightarrow x = y\ (mod\ p) \Longrightarrow f\ x = f\ y\ (mod\ q)$
$\langle proof \rangle$

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

lemma *fun-quotient-empty-domain*: $(/\emptyset\ /\Rightarrow\ q) = /\mathbf{1}\mathcal{U}$
$\langle proof \rangle$

lemma *fun-quotient-empty-range*: $q \neq /\emptyset \Longrightarrow (q\ /\Rightarrow\ /\emptyset) = /\emptyset$
$\langle proof \rangle$

lemma *fun-quotient-subset-weak-intro*:
assumes $p2\ /\leq\ p1 \wedge q1\ /\leq\ q2$
shows $p1\ /\Rightarrow\ q1\ /\leq\ p2\ /\Rightarrow\ q2$
$\langle proof \rangle$

lemma *fun-quotient-subset-weakdef*:
$(p1\ /\Rightarrow\ q1\ /\leq\ p2\ /\Rightarrow\ q2) =$
$(\forall\ f\ g.\ (\forall\,x\ y.\ x = y\ (mod\ p1) \longrightarrow f\ x = g\ y\ (mod\ q1)) \longrightarrow$
$(\forall\,x\ y.\ x = y\ (mod\ p2) \longrightarrow f\ x = g\ y\ (mod\ q2)))$
$\langle proof \rangle$

lemma *fun-quotient-range-subset-weak*:
assumes *sub*: $((p1 :: 'a\ quotient)\ /\Rightarrow\ q1\ /\leq\ p2\ /\Rightarrow\ q2)$
assumes *nonempty*: $p2 \neq /\emptyset$

shows $q1$ $/\le$ $q2$
⟨*proof*⟩

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

lemma *fun-quotient-domain-subset-weak*:
assumes *sub*: $((p1 :: 'a\ quotient)\ /\Rightarrow\ q1\ /\le\ p2\ /\Rightarrow\ q2)$
assumes *nontrivial*: $\neg\ (/\mathbf{1}(QField\ q1)\ /\le\ q2)$
shows $p2\ /\le\ p1$
⟨*proof*⟩

4.11 Vectors as Quotients

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

lemma *qequal-vector-sym*: $symp\ (qequal\text{-}vector\ q\ n)$
⟨*proof*⟩

lemma *qequal-vector-trans*: $transp\ (qequal\text{-}vector\ q\ n)$
⟨*proof*⟩

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

lemma *vector-quotient-Rel*: $Rel\ (q\ /\hat{}\ n) = \{\ (u,\ v).\ qequal\text{-}vector\ q\ n\ u\ v\ \}$
⟨*proof*⟩

lemma *vector-quotient-in*: $(u\ /\in\ q\ /\hat{}\ n) = (qequal\text{-}vector\ q\ n\ u\ u)$
⟨*proof*⟩

lemma *vector-quotient-mod*: $(u = v\ (mod\ q\ /\hat{}\ n)) = (qequal\text{-}vector\ q\ n\ u\ v)$
⟨*proof*⟩

lemma *vector-quotient-nth*: $i < n \Longrightarrow (\lambda\ u.\ u\ !\ i)\ /\in\ q\ /\hat{}\ n\ /\Rightarrow\ q$
⟨*proof*⟩

lemma *vector-quotient-nth-in*: $i < n \Longrightarrow u\ /\in\ q\ /\hat{}\ n \Longrightarrow u\ !\ i\ /\in\ q$
⟨*proof*⟩

lemma *vector-quotient-nth-mod*: $i < n \Longrightarrow u = v\ (mod\ q\ /\hat{}\ n) \Longrightarrow u\ !\ i = v\ !\ i\ (mod\ q)$
⟨*proof*⟩

lemma *vector-quotient-append*: $(@)\ /\in\ q\ /\hat{}\ n\ /\Rightarrow\ q\ /\hat{}\ m\ /\Rightarrow\ q\ /\hat{}(n+m)$

$\langle proof \rangle$

lemma *vector-quotient-append-in*: x /$\in$ q /$\frown$ $n \Longrightarrow y$ /$\in$ q /$\frown$ $m \Longrightarrow x$ @ y /$\in$ q /$\frown$ $(n+m)$
$\langle proof \rangle$

lemma *vector-quotient-append-mod*:
$x = x'$ (*mod* q /$\frown$ n) $\Longrightarrow y = y'$ (*mod* q /$\frown$ m) $\Longrightarrow x@y = x'@y'$ (*mod* q /$\frown$ $(n+m)$)
$\langle proof \rangle$

lemma *vector-quotient-weak-subset-intro*: p /$\leq$ $q \Longrightarrow p$/$\frown n$ /$\leq$ q/$\frown n$
$\langle proof \rangle$

lemma *vector-quotient-strong-subset-intro*: p /$\subseteq$ $q \Longrightarrow p$/$\frown n$ /$\subseteq$ q/$\frown n$
$\langle proof \rangle$

lemma *vector-quotient-bishop-subset-intro*: p /$\sqsubseteq$ $q \Longrightarrow p$/$\frown n$ /$\sqsubseteq$ q/$\frown n$
$\langle proof \rangle$

4.12 Tuples as Quotients

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

lemma *qequal-tuple-sym*: *symp* (*qequal-tuple qs*)
$\langle proof \rangle$

lemma *qequal-tuple-trans*: *transp* (*qequal-tuple qs*)
$\langle proof \rangle$

definition *tuple-quotient* :: *'a quotient list* $\Rightarrow$ *'a list quotient* ('/$\times$) **where**
/$\times$ *qs* = *QuotientP* (*qequal-tuple qs*)

lemma *tuple-quotient-rel*: *Rel* (/$\times$ *qs*) = { (u, v). *qequal-tuple qs u v* }
$\langle proof \rangle$

lemma *tuple-quotient-in*: (u /$\in$ (/$\times$ *qs*)) = (*qequal-tuple qs u u*)
$\langle proof \rangle$

lemma *tuple-quotient-mod*: (u = v (*mod* /$\times$ *qs*)) = (*qequal-tuple qs u v*)
$\langle proof \rangle$

lemma *tuple-quotient-nth*: i < *length qs* $\Longrightarrow$ (λ u. u ! i) /$\in$ /$\times$ *qs* /$\Rightarrow$ *qs* ! i
$\langle proof \rangle$

lemma *tuple-quotient-append*: (@) /$\in$ /$\times$ *ps* /$\Rightarrow$ /$\times$ *qs* /$\Rightarrow$ /$\times$ (*ps*@*qs*)
$\langle proof \rangle$

lemma *vectors-are-tuples*: $q\ /\widehat{}\ n = /\times\ (replicate\ n\ q)$
⟨*proof*⟩

lemma *tuple-quotient-strong-subset-intro*:
$length\ ps = length\ qs \Longrightarrow (\bigwedge i.\ i < length\ ps \Longrightarrow ps!i\ /\subseteq\ qs!i) \Longrightarrow /\times\ ps\ /\subseteq\ /\times\ qs$
⟨*proof*⟩

lemma *tuple-quotient-bishop-subset-intro*:
$length\ ps = length\ qs \Longrightarrow (\bigwedge i.\ i < length\ ps \Longrightarrow ps!i\ /\sqsubseteq\ qs!i) \Longrightarrow /\times\ ps\ /\sqsubseteq\ /\times\ qs$
⟨*proof*⟩

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 $'a\ operation = 'a\ list \Rightarrow 'a$
type-synonym $'a\ operator = 'a\ operation\ list \Rightarrow 'a$

definition $operations :: 'a\ quotient \Rightarrow nat \Rightarrow ('a\ operation)\ quotient$ **where**
$operations\ U\ n = U\ /\widehat{}\ n\ /\!\Rightarrow U$

definition $operators :: 'a\ quotient \Rightarrow shape \Rightarrow ('a\ operator)\ quotient$ **where**
$operators\ U\ s = /\times\ (map\ (\lambda\ deps.\ operations\ U\ (card\ deps))\ (Preshape\ s))\ /\!\Rightarrow U$

definition $value\text{-}op :: 'a \Rightarrow 'a\ operation$ **where**
$value\text{-}op\ u = (\lambda\ \text{-}.\ u)$

lemma *operators-appeq-intro*:
assumes *FG*: $F = G\ (mod\ operators\ \mathcal{U}\ s)$
assumes *lenfs*: $length\ fs = \S ar\ s$
assumes *lengs*: $length\ gs = \S ar\ s$
assumes *fsgs*: $(\bigwedge i.\ i < \S ar\ s \Longrightarrow fs!i = gs!i\ (mod\ operations\ \mathcal{U}\ (s.\#i)))$
shows $F\ fs = G\ gs\ (mod\ \mathcal{U})$
⟨*proof*⟩

lemma *operator-appeq-intro*:
assumes *F*: $F\ /\!\in\ operators\ \mathcal{U}\ s$
assumes *lenfs*: $length\ fs = \S ar\ s$

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 $F\ fs = F\ gs\ (mod\ \mathcal{U})$
⟨*proof*⟩

lemma *operations-eq-intro*:
assumes $\bigwedge\ us\ vs.\ us = vs\ (mod\ \mathcal{U}\ /\widehat{}\ n) \Longrightarrow f\ us = g\ vs\ (mod\ \mathcal{U})$
shows $f = g\ (mod\ operations\ \mathcal{U}\ n)$
⟨*proof*⟩

lemma *operations-mod*:
$(f = g\ (mod\ operations\ \mathcal{U}\ n)) = (\forall\ us\ vs.\ us = vs\ (mod\ \mathcal{U}/\widehat{}n) \longrightarrow f\ us = g\ vs\ (mod\ \mathcal{U}))$
⟨*proof*⟩

5.2 Compatibility of Shape and Operator

definition *shape-compatible* :: *'a quotient* ⇒ *shape* ⇒ *'a operator* ⇒ *bool* **where**
shape-compatible U s op = (*op* /∈ *operators U s*)

definition *shape-compatible-opt* :: *'a quotient* ⇒ *shape option* ⇒ *'a operator option* ⇒ *bool* **where**
shape-compatible-opt U s op = ((*s* = *None* ∧ *op* = *None*) ∨ (*s* ≠ *None* ∧ *op* ≠ *None* ∧
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* × *signature* × *'a operators*

definition *is-algebra* :: *'a prealgebra* ⇒ *bool* **where**
is-algebra paa =
(*let U* = *fst paa in*
let sig = *fst* (*snd paa*) *in*
let ops = *snd* (*snd paa*) *in*
U ≠ /∅ ∧ (∀ *a*. *shape-compatible-opt U* (*sig a*) (*ops a*)))

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

lemma *trivial-prealgebra*: *is-algebra trivial-prealgebra*
⟨*proof*⟩

typedef *'a algebra* = { *aa* :: *'a prealgebra* . *is-algebra aa* } **morphisms** *Prealgebra Algebra*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Univ-nonempty*: *Univ aa* ≠ /∅
⟨*proof*⟩

lemma *algebra-compatibility*: *shape-compatible-opt* (*Univ aa*) (*Sig aa a*) (*Ops aa a*)
⟨*proof*⟩

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* = {}
⟨*proof*⟩

lemma *deduction-forall-deps-0*[*iff*]: 𝔇!!*abstr-forall*.@*0*([*x*]) = [*x*]
⟨*proof*⟩

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-contains sig* a (*length xs*) (*length ts*) ∧
distinct xs ∧
(∀ t = *ts*!-. *nt-wf sig* t))

lemma *nt-wf-x0*[*iff*]: *nt-wf sig* §x ⟨*proof*⟩
lemma *nt-wf-y0*[*iff*]: *nt-wf sig* §y ⟨*proof*⟩
lemma *nt-wf-A0*[*iff*]: *nt-wf sig* §A ⟨*proof*⟩
lemma *nt-wf-A1*[*iff*]: *nt-wf sig* §A[t] = *nt-wf sig* t
⟨*proof*⟩
lemma *nt-wf-B0*[*iff*]: *nt-wf sig* §B ⟨*proof*⟩
lemma *nt-wf-B1*[*iff*]: *nt-wf sig* §B[t] = *nt-wf sig* t
⟨*proof*⟩
lemma *nt-wf-C0*[*iff*]: *nt-wf sig* §C ⟨*proof*⟩
lemma *nt-wf-C1*[*iff*]: *nt-wf sig* §C[t] = *nt-wf sig* t
⟨*proof*⟩

lemma *nt-wf-true*[*simp*]: *nt-wf* $\mathfrak{D}$ ‘⊤
⟨*proof*⟩

lemma *nt-wf-implies*[*simp*]: *nt-wf* $\mathfrak{D}$ (A ‘$\Rightarrow$ B) = (*nt-wf* $\mathfrak{D}$ A $\wedge$ *nt-wf* $\mathfrak{D}$ B)
⟨*proof*⟩

lemma *nt-wf-forall*[*simp*]: *nt-wf* $\mathfrak{D}$ (‘$\forall$ ‘x. t) = *nt-wf* $\mathfrak{D}$ t
⟨*proof*⟩

lemma *sig-extends-nt-wf*: $V \succeq U \Longrightarrow$ *nt-wf* U t $\Longrightarrow$ *nt-wf* V t
⟨*proof*⟩

6.4 Free Variables

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

lemma *nt-free-x0*: *nt-free sig* §x = {(‘x, *0*)} ⟨*proof*⟩

lemma *nt-free-y0*: *nt-free sig* §y = {(‘y, *0*)} ⟨*proof*⟩

lemma *nt-free-A0*: *nt-free sig* §A = {(‘A, *0*)} ⟨*proof*⟩
lemma *nt-free-A1*: *nt-free sig* §A[t] = {(‘A, *Suc 0*)} $\cup$ *nt-free sig t*
⟨*proof*⟩

lemma *nt-free-B0*: *nt-free sig* §B = {(‘B, *0*)} ⟨*proof*⟩
lemma *nt-free-B1*: *nt-free sig* §B[t] = {(‘B, *Suc 0*)} $\cup$ *nt-free sig t*
⟨*proof*⟩

lemma *nt-free-C0*: *nt-free sig* §C = {(‘C, *0*)} ⟨*proof*⟩
lemma *nt-free-C1*: *nt-free sig* §C[t] = {(‘C, *Suc 0*)} $\cup$ *nt-free sig t*
⟨*proof*⟩

lemma *nt-free-true*: *nt-free* $\mathfrak{D}$ ‘$\top$ = {}
⟨*proof*⟩

lemma *nt-free-implies*: *nt-free* $\mathfrak{D}$ (s ‘$\Rightarrow$ t) = *nt-free* $\mathfrak{D}$ s $\cup$ *nt-free* $\mathfrak{D}$ t
⟨*proof*⟩

lemma *nt-free-forall*: *nt-free* $\mathfrak{D}$ (‘$\forall$ x. t) = *nt-free* $\mathfrak{D}$ t − {(x, *0*)}
thm *forall-app-def binders-as-vars-def*
⟨*proof*⟩

lemma *sig-extends-nt-free*: $V \succeq U \Longrightarrow$ *nt-wf* U t $\Longrightarrow$ *nt-free* V t = *nt-free* U t
⟨*proof*⟩

lemma *nt-free-VarApp*: *nt-free sig* (*VarApp x ts*) =
{(x, *length ts*)} $\cup \bigcup$ { *nt-free sig t* | t. $t \in$ *set ts* }
⟨*proof*⟩

lemma *nt-free-VarApp-arg-subset*:
  assumes *nt-free sig* (*VarApp x ts*) $\subseteq X$
  assumes $i <$ *length ts*
  shows *nt-free sig* (*ts* ! *i*) $\subseteq X$
  ⟨*proof*⟩

lemma *nt-free-ConsApp*:
  shows *nt-free sig* (*AbsApp a xs ts*) =
    $\bigcup$ { *nt-free sig* (*ts*!*i*) − (*sig*!!*a*.@*i*(*xs*))‘,*0* | *i*. *i* < *length ts* }
  ⟨*proof*⟩

lemma *nt-free-ConsApp-arg-subset*:
  assumes *nt-free sig* (*AbsApp a xs ts*) $\subseteq X$
  assumes $i <$ *length ts*
  shows *nt-free sig* (*ts*!*i*) $\subseteq X \cup$ (*sig*!!*a*.@*i*(*xs*))‘,*0*
⟨*proof*⟩

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* !♮ *i* ≡ $\mathcal{S}$!!*a*.♮*i*

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

abbreviation
  SelDeps :: *abstraction* ⇒ *nat* ⇒ ′*b list* ⇒ ′*b list* (-!@-′(-′) [*100*, *101*, *0*] *100*)
  where *a*!@*i*(*xs*) ≡ $\mathcal{S}$!!*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 $\checkmark a \equiv ((\mathcal{S}\ a) \neq None)$

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

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

lemma *wf-implies-valid-abs*:
assumes *wf*: *wf* (*AbsApp a xs ts*)
shows $\checkmark a$
⟨*proof*⟩

lemma *wf-VarApp*: *wf* (*VarApp x ts*) = ($\forall$ $t \in$ *set ts*. *wf t*)
⟨*proof*⟩

lemma *wf-AbsApp-valence*: **assumes** *wf*: *wf* (*AbsApp a xs ts*) **shows** *length xs* = $\S v\ a$
⟨*proof*⟩

lemma *shape-deps-upper-bound*: $\checkmark a \Longrightarrow i < \S a\ a \Longrightarrow a!\natural i \subseteq$ *nats* ($\S v\ a$)
⟨*proof*⟩

lemma *finite-shape-deps*: $\checkmark a \Longrightarrow i < \S a\ a \Longrightarrow$ *finite*($a!\natural i$)
⟨*proof*⟩

lemma *length-boundvars-at*:
assumes *wf*: *wf* (*AbsApp a xs ts*)
assumes *i*: $i <$ *length ts*
shows *length* ($a!@i(xs)$) = $a\ !\#\ i$
⟨*proof*⟩

definition *closed* :: *nterm* $\Rightarrow$ *bool*
where *closed t* = (*frees t* = {})

end

6.6 Abstraction Algebra Locale

locale *algloc* = *sigloc Sig* $\mathfrak{A}$ **for** *AA* :: *'a algebra* ($\mathfrak{A}$)
begin

abbreviation
Universe :: *'a quotient* ($\mathcal{U}$)
where $\mathcal{U} \equiv$ *Univ* $\mathfrak{A}$

abbreviation
Operators :: *'a operators* ($\mathcal{O}$)
where $\mathcal{O} \equiv$ *Ops* $\mathfrak{A}$

abbreviation
  Signature :: *signature* ($\mathcal{S}$)
  where $\mathcal{S} \equiv$ *Sig* $\mathfrak{A}$

notation
  Deps (**infixl** !$\natural$ *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* $\mathcal{U}$ ($\mathcal{S}!!a$)
  ⟨*proof*⟩

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 $\mathcal{U}$)
  ⟨*proof*⟩

lemma *qequal-valuation-trans*: *transp* (*qequal-valuation* X $\mathcal{U}$)
  ⟨*proof*⟩

definition *valuation-quotient* :: *variables* ⇒ *'a quotient* ⇒ *'a valuation quotient* (**infix** $\rightarrowtail$ *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 }
  ⟨*proof*⟩

lemma *valuation-quotient-mod*:
  (τ = υ (*mod* $X \rightarrowtail \mathcal{U}$)) = *qequal-valuation* X $\mathcal{U}$ τ υ
  ⟨*proof*⟩

lemma *valuation-quotient-in*:
  (υ /∈ $X \rightarrowtail \mathcal{U}$) = *qequal-valuation* X $\mathcal{U}$ υ υ
  ⟨*proof*⟩

lemma *valuation-quotient-app*:
  τ = υ (*mod* $X \rightarrowtail \mathcal{U}$) $\Longrightarrow$ (x, n) ∈ X $\Longrightarrow$ *us* = *vs* (*mod* $\mathcal{U}$ /^ n) $\Longrightarrow$ τ (x, n) *us* = υ (x, n) *vs* (*mod* $\mathcal{U}$)
  ⟨*proof*⟩

lemma *valuation-mod-subdomain*:
  assumes *mod*: τ = υ (*mod* $X \rightarrowtail \mathcal{U}$)
  assumes *sub*: $Y \subseteq X$
  shows τ = υ (*mod* $Y \rightarrowtail \mathcal{U}$)
⟨*proof*⟩

lemma *update-valuation-skipvar*:
  assumes *x*: $x \notin$ *set xs*
  shows υ{*xs* := *us*}(x, n) = υ(x, n)
⟨*proof*⟩

lemma *subtracted-bound-vars*:
  assumes *x*: (x, n) ∈ X − *xs*‘,*0*
  shows n > *0* ∨ $x \notin$ *set xs*
  ⟨*proof*⟩

lemma *update-valuation-eq-intro*:
  assumes τ = υ (*mod* $X \rightarrowtail \mathcal{U}$)
  assumes *us* = *vs* (*mod* $\mathcal{U}$/^n)
  assumes *length xs* = n

shows $\tau\{xs := us\} = \upsilon\{xs := vs\}$ $(mod\ (X \cup ((xs)\text{`},0)) \rightarrowtail \mathcal{U})$
$\langle proof \rangle$

lemma *valuations-empty-domain*[*simp*]: $\{\} \rightarrowtail \mathcal{U} = /\mathbf{1}\mathcal{U}$
$\langle proof \rangle$

7.2 Evaluation

context *algloc*
begin

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

fun *eval* :: *nterm* $\Rightarrow$ *'a valuation* $\Rightarrow$ *'a* ($\langle$-; -$\rangle$) **where**
eval (*VarApp x ts*) $\upsilon = \upsilon$ (*x*, *length ts*) (§*map t* = *ts*!-. *eval t* υ)
| *eval* (*AbsApp a xs ts*) υ = ($\mathcal{O}$!! *a*) (§*map t* = *ts*!*i*.
(λ *us*. *eval t* (υ { *a*!@*i*(*xs*) := *us* })))

lemma *eval-modulo*:
wf t $\Longrightarrow$
frees t $\subseteq X \Longrightarrow$
$\tau = \upsilon$ $(mod\ \mathbb{V}\ X) \Longrightarrow$
eval t τ = *eval t* υ $(mod\ \mathcal{U})$
$\langle proof \rangle$

lemma *eval-is-fun-modulo*:
assumes *wf*: *wf t*
shows *eval t* $/\!\in \mathbb{V}$ (*frees t*) $/\!\Rightarrow \mathcal{U}$
$\langle proof \rangle$

lemma *eval-closed*:
assumes *wf*: *wf t*
assumes *cl*: *closed t*
shows *eval t* τ = *eval t* υ $(mod\ \mathcal{U})$
$\langle proof \rangle$

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* = ($\forall\ \upsilon.\ \upsilon\ /\!\in \mathbb{V}$ *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

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

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

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

lemma $\textit{semantically-equivalent}\ (\alpha_1{::}'a)\ s\ t = \textit{semantically-equivalent}\ (\alpha_2{::}'a)\ s\ t$
  ⟨*proof*⟩

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* :: $\textit{nat set} \Rightarrow \textit{nat} \Rightarrow \textit{nat set}$ (**infixl** $.\!\uparrow$ *80*)
  where $\textit{raise-indices}\ I\ m = \{\ i + m \mid i.\ i \in I\ \}$

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

lemma *raise-indices-0*[*simp*]: $I\ .\!\uparrow 0 = I$
  ⟨*proof*⟩

lemma *lower-indices-0*[*simp*]: $I\ .\!\downarrow 0 = I$
  ⟨*proof*⟩

lemma *raise-indices-mono*: $A \subseteq B \Longrightarrow A\ .\!\uparrow m \subseteq B\ .\!\uparrow m$
  ⟨*proof*⟩

lemma *lower-indices-mono*: $A \subseteq B \Longrightarrow A\ .\!\downarrow m \subseteq B\ .\!\downarrow m$
  ⟨*proof*⟩

lemma *raise-lower-indices-le*[*simp*]: $n \leq m \Longrightarrow I\ .\!\uparrow m\ .\!\downarrow n = I\ .\!\uparrow (m - n)$

$\langle proof \rangle$

lemma *raise-lower-indices-ge*[*simp*]: $n \geq m \Longrightarrow I\ .\!\uparrow m\ .\!\downarrow n = I\ .\!\downarrow (n - m)$
$\langle proof \rangle$

lemma *erase-bottom-indices*: $i \in I\ .\!\downarrow m\ .\!\uparrow m \Longrightarrow i \geq m \wedge i \in I$
$\langle proof \rangle$

lemma *undo-raise-indices*: $i \in I\ .\!\uparrow m \Longrightarrow i - m \in I$
$\langle proof \rangle$

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*: $\bigwedge i.\ i < length\ ts \Longrightarrow unbounds\ (ts\ !\ i) \subseteq unbounds\ (FreeVar\ x\ ts)$
$\langle proof \rangle$

lemma *unbounds-abstr*:
assumes *i*: $k < length\ ts$
shows *unbounds* (*ts* ! *k*) ⊆ *unbounds* (*Abstr a ts*) .↑ §*v a* ∪ *nats* (§*v a*)
$\langle proof \rangle$

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*: $\bigwedge i.\ i < length\ ts \Longrightarrow bfrees\ (ts\ !\ i) \subseteq bfrees\ (FreeVar\ x\ ts)$
$\langle proof \rangle$

lemma *bfrees-abstr-arg*: $\bigwedge i.\ i < length\ ts \Longrightarrow bfrees\ (ts\ !\ i) \subseteq bfrees\ (Abstr\ a\ ts)$
$\langle proof \rangle$

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*: $bwf\ (FreeVar\ x\ ts) \Longrightarrow i < length\ ts \Longrightarrow bwf\ (ts!i)$
$\langle proof \rangle$

lemma *bwf-abstr-arg*: $bwf\ (Abstr\ a\ ts) \Longrightarrow i < length\ ts \Longrightarrow bwf\ (ts!i)$
⟨*proof*⟩

lemma *unbounds-bwf-abstr*:
assumes *i*: $k < length\ ts$
assumes *wf*: $bwf\ (Abstr\ a\ ts)$
shows $unbounds\ (ts\ !\ k) \subseteq unbounds\ (Abstr\ a\ ts)\ .\!\uparrow\ \S v\ a \cup a!\natural k$
⟨*proof*⟩

lemma *upper-bound-unbounds-abstr-arg*:
assumes *i*:$i \in \bigcup\ \{\ unbounds\ t\ |\ t.\ t \in set\ ts\ \}$
shows $i \in unbounds\ (Abstr\ a\ ts)\ .\!\uparrow\ \S v\ a \cup (nats\ (\S v\ a))$
⟨*proof*⟩

lemma *upper-bound-erased-unbounds-abstr-arg*:
assumes *i*:$i \in \bigcup\ \{\ unbounds\ t\ |\ t.\ t \in set\ ts\ \}\ .\!\downarrow\ \S v\ a\ .\!\uparrow\ \S v\ a$
shows $i \in unbounds\ (Abstr\ a\ ts)\ .\!\uparrow\ \S v\ a$
⟨*proof*⟩

end

8.4 Environments

type-synonym $'a\ env = nat \Rightarrow 'a$

definition *update-env* :: $'a\ env \Rightarrow nat \Rightarrow nat\ set \Rightarrow 'a\ list \Rightarrow 'a\ env$
(-↑-{- := -} [*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* :: $'a\ env \Rightarrow nat \Rightarrow 'a\ env$
(-↑-{} [*1000*, *51*] *1000*)
where
$env \uparrow m\ \{\} \equiv env \uparrow m\ \{\ \{\} := []\ \}$

lemma *env-app-noupdate*: $finite\ js \Longrightarrow j \notin js \Longrightarrow env \uparrow m\ \{\ js := xs\ \}\ j = env \uparrow m\ \{\}\ j$
⟨*proof*⟩

lemma *env-raised-app*: $env \uparrow m\ \{\}\ j = env\ (j - m)$
⟨*proof*⟩

lemma *env-app-update*:
$finite\ js \Longrightarrow j \in js \Longrightarrow env \uparrow m\ \{\ js := us\ \}\ j = us\ !\ the\ (index\text{-}of\ j\ (sorted\text{-}list\ js))$
⟨*proof*⟩

context *algloc* **begin**

definition *env-quotient* :: *nat set* ⇒ (*'a env*) *quotient* ($\mathbb{E}$)
  where *env-quotient* I = ($/\equiv I$ $/\Rightarrow \mathcal{U}$)

lemma *env-subset*: $A \subseteq B \Longrightarrow \mathbb{E}\ B\ /\leq \mathbb{E}\ A$
  ⟨*proof*⟩

lemma *env-subset-mod*: $A \subseteq B \Longrightarrow$ *env1* = *env2* (*mod* $\mathbb{E}$ B) $\Longrightarrow$ *env1* = *env2* (*mod* $\mathbb{E}$ A)
  ⟨*proof*⟩

lemma *env-app*: $i \in I \Longrightarrow$ *env1* = *env2* (*mod* $\mathbb{E}$ I) $\Longrightarrow$ *env1* i = *env2* i (*mod* $\mathcal{U}$)
  ⟨*proof*⟩

lemma *env-mod*: (*env1* = *env2* (*mod* $\mathbb{E}$ I)) = ($\forall$ $i \in I$. *env1* i = *env2* i (*mod* $\mathcal{U}$))
  ⟨*proof*⟩

8.5 Evaluation

fun *beval* :: *bterm* ⇒ *'a valuation* ⇒ *'a env* ⇒ *'a* (⟨-; -, -⟩) **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* ↑ §υ a {a!♮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*⟩

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.