The Hereditarily Finite Sets

Lawrence C. Paulson

July 22, 2014

Abstract

The theory of hereditarily finite sets is formalised, following the development of Świerczkowski [2]. An HF set is a finite collection of other HF sets; they enjoy an induction principle and satisfy all the axioms of ZF set theory apart from the axiom of infinity, which is negated. All constructions that are possible in ZF set theory (Cartesian products, disjoint sums, natural numbers, functions) without using infinite sets are possible here. The definition of addition for the HF sets follows Kirby [1].

This development forms the foundation for the Isabelle proof of Gödel's incompleteness theorems, which has been formalised separately.

Contents

Chapter 1

The Hereditarily Finite Sets

theory *HF* **imports** ~~/*src*/*HOL*/*Library*/*Nat-Bijection*
begin

From "Finite sets and Gdel's Incompleteness Theorems" by S. Swierczkowski. Thanks for Brian Huffman for this development, up to the cases and induct rules.

1.1 Basic Definitions and Lemmas

typedef *hf* = *UNIV* :: *nat set* **..**

definition *hfset* :: *hf* ⇒ *hf set*
  where *hfset a* = *Abs-hf* ‘ *set-decode* (*Rep-hf a*)

definition *HF* :: *hf set* ⇒ *hf*
  where *HF A* = *Abs-hf* (*set-encode* (*Rep-hf* ‘ *A*))

definition *hinsert* :: *hf* ⇒ *hf* ⇒ *hf*
  where *hinsert a b* = *HF* (*insert a* (*hfset b*))

definition *hmem* :: *hf* ⇒ *hf* ⇒ *bool*   (**infixl** <: *50*)
  where *hmem a b* ⟷ *a* ∈ *hfset b*

instantiation *hf* :: *zero*
begin

definition
  Zero-hf-def: *0* = *HF* {}

instance ..

end

HF Set enumerations

syntax
-HFinset :: *args* ⇒ *hf* ({|(-)|})

syntax (*xsymbols*)
-HFinset :: *args* ⇒ *hf* (⦃-⦄)
-inserthf :: *hf* ⇒ *hf* ⇒ *hf* (**infixl** ◁ *60*)

notation (*xsymbols*)
hmem (**infixl** ∈ *50*)

translations
y ◁ *x* == *CONST hinsert x y*
{|*x*, *y*|} == ⦃*y*⦄ ◁ *x*
{|*x*|} == *0* ◁ *x*

lemma *finite-hfset* [*simp*]: *finite* (*hfset a*)
unfolding *hfset-def* **by** *simp*

lemma *HF-hfset* [*simp*]: *HF* (*hfset a*) = *a*
unfolding *HF-def hfset-def*
by (*simp add*: *image-image Abs-hf-inverse Rep-hf-inverse*)

lemma *hfset-HF* [*simp*]: *finite A* ⟹ *hfset* (*HF A*) = *A*
unfolding *HF-def hfset-def*
by (*simp add*: *image-image Abs-hf-inverse Rep-hf-inverse*)

lemma *hmem-hempty* [*simp*]: ¬ *a* ∈ *0*
unfolding *hmem-def Zero-hf-def* **by** *simp*

lemmas *hemptyE* [*elim*!] = *hmem-hempty* [*THEN notE*]

lemma *hmem-hinsert* [*iff*]:
hmem a (*c* ◁ *b*) ⟷ *a* = *b* ∨ *a* ∈ *c*
unfolding *hmem-def hinsert-def* **by** *simp*

lemma *hf-ext*: *a* = *b* ⟷ (∀ *x*. *x* ∈ *a* ⟷ *x* ∈ *b*)
unfolding *hmem-def set-eq-iff* [*symmetric*]
by (*metis HF-hfset*)

lemma *finite-cases* [*consumes 1*, *case-names empty insert*]:
⟦*finite F*; *F* = {} ⟹ *P*; ⋀*A x*. ⟦*F* = *insert x A*; *x* ∉ *A*; *finite A*⟧ ⟹ *P*⟧ ⟹ *P*
by (*induct F rule*: *finite-induct*, *simp-all*)

lemma *hf-cases* [*cases type*: *hf*, *case-names 0 hinsert*]:
obtains *y* = *0* | *a b* **where** *y* = *b* ◁ *a* **and** ¬ *a* ∈ *b*
proof −
have *finite* (*hfset y*) **by** (*rule finite-hfset*)
thus *thesis*

by (*metis Zero-hf-def finite-cases hf-ext hfset-HF hinsert-def hmem-def that*)
qed

lemma *Rep-hf-hinsert*:
$\neg\ a \in b \Longrightarrow$ *Rep-hf* (*hinsert a b*) $=$ *2* ^ (*Rep-hf a*) $+$ *Rep-hf b*
unfolding *hinsert-def HF-def hfset-def*
apply (*simp add*: *image-image Abs-hf-inverse Rep-hf-inverse*)
apply (*subst set-encode-insert*, *simp*)
apply (*clarsimp simp add*: *hmem-def hfset-def image-def*
Rep-hf-inject [*symmetric*] *Abs-hf-inverse*, *simp*)
done

lemma *less-two-power*: $n <$ *2* ^ n
by (*induct n*, *auto*)

1.2 Verifying the Axioms of HF

HF1

lemma *hempty-iff*: $z{=}0 \longleftrightarrow (\forall x.\ \neg\ x \in z)$
by (*simp add*: *hf-ext*)

HF2

lemma *hinsert-iff*: $z = y \triangleleft x \longleftrightarrow (\forall u.\ u \in z \longleftrightarrow u \in y \mid u{=}x)$
by (*auto simp*: *hf-ext*)

HF induction

lemma *hf-induct* [*induct type*: *hf*, *case-names 0 hinsert*]:
assumes [*simp*]: *P 0*
$\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ \neg\ x \in y]\!] \Longrightarrow P\ (y \triangleleft x)$
shows *P z*
proof (*induct z rule*: *wf-induct* [**where** *r=measure Rep-hf*, *OF wf-measure*])
case (*1 x*) **show** *?case*
proof (*cases x rule*: *hf-cases*)
case *0* **thus** *?thesis* **by** *simp*
next
case (*hinsert a b*)
thus *?thesis* **using** *1*
by (*simp add*: *Rep-hf-hinsert*
less-le-trans [*OF less-two-power le-add1*])
qed
qed

HF3

lemma *hf-induct-ax*: $[\![P\ 0;\ \forall x.\ P\ x \longrightarrow (\forall y.\ P\ y \longrightarrow P\ (x \triangleleft y))]\!] \Longrightarrow P\ x$
by (*induct x*, *auto*)

lemma *hf-equalityI* [*intro*]: $(\bigwedge x.\ x \in a \longleftrightarrow x \in b) \Longrightarrow a = b$
by (*simp add*: *hf-ext*)

lemma *hinsert-nonempty* [*simp*]: $A \triangleleft a \neq 0$
  by (*auto simp*: *hf-ext*)

lemma *hinsert-commute*: $(z \triangleleft y) \triangleleft x = (z \triangleleft x) \triangleleft y$
  by (*auto simp*: *hf-ext*)

lemma *singleton-eq-iff* [*iff*]: $\{\!|a|\!\} = \{\!|b|\!\} \longleftrightarrow a{=}b$
  by (*metis hmem-hempty hmem-hinsert*)

lemma *doubleton-eq-iff*: $\{\!|a,b|\!\} = \{\!|c,d|\!\} \longleftrightarrow (a{=}c\ \&\ b{=}d) \mid (a{=}d\ \&\ b{=}c)$
  by (*metis* (*hide-lams*, *no-types*) *hinsert-commute hmem-hempty hmem-hinsert*)

1.3 Ordered Pairs, from ZF/ZF.thy

definition *hpair* :: $hf \Rightarrow hf \Rightarrow hf$
  where $hpair\ a\ b = \{\!|\{\!|a|\!\},\{\!|a,b|\!\}|\!\}$

definition *hfst* :: $hf \Rightarrow hf$
  where $hfst\ p \equiv THE\ x.\ \exists\, y.\ p = hpair\ x\ y$

definition *hsnd* :: $hf \Rightarrow hf$
  where $hsnd\ p \equiv THE\ y.\ \exists\, x.\ p = hpair\ x\ y$

definition *hsplit* :: $[[hf, hf] \Rightarrow 'a, hf] \Rightarrow 'a{::}\{\}$ — for pattern-matching
  where $hsplit\ c \equiv \%p.\ c\ (hfst\ p)\ (hsnd\ p)$

Ordered Pairs, from ZF/ZF.thy

nonterminal *hfs*
syntax
      :: $hf \Rightarrow hfs$ (-)
  -Enum   :: $[hf, hfs] \Rightarrow hfs$ (-,/ -)
  -Tuple   :: $[hf, hfs] \Rightarrow hf$ (<(-,/ -)>)
  -hpattern :: $[pttrn, patterns] \Rightarrow pttrn$ (<-,/ ->)
syntax (*xsymbols*)
  -Tuple   :: $[hf, hfs] \Rightarrow hf$ (⟨(-,/ -)⟩)
  -hpattern :: $[pttrn, patterns] \Rightarrow pttrn$ (⟨-,/ -⟩)
syntax (*HTML* **output**)
  -Tuple   :: $[hf, hfs] \Rightarrow hf$ (⟨(-,/ -)⟩)
  -hpattern :: $[pttrn, patterns] \Rightarrow pttrn$ (⟨-,/ -⟩)

translations
  $<x, y, z>\ \ == <x, <y, z>>$
  $<x, y>\ \ == CONST\ hpair\ x\ y$
  $<x, y, z>\ \ == <x, <y, z>>$
  $\%<x,y,zs>.\ b == CONST\ hsplit(\%x\ <y,zs>.\ b)$
  $\%<x,y>.\ b\ \ == CONST\ hsplit(\%x\ y.\ b)$

lemma *hpair-def'*: *hpair a b* = $\{\!|\{\!|a,a|\!\},\{\!|a,b|\!\}|\!\}$
by (*auto simp*: *hf-ext hpair-def*)

lemma *hpair-iff* [*simp*]: *hpair a b* = *hpair a' b'* $\longleftrightarrow$ *a=a'* & *b=b'*
by (*auto simp*: *hpair-def' doubleton-eq-iff*)

lemmas *hpair-inject* = *hpair-iff* [*THEN iffD1*, *THEN conjE*, *elim*!]

lemma *hfst-conv* [*simp*]: *hfst* $\langle a,b\rangle$ = *a*
by (*simp add*: *hfst-def*)

lemma *hsnd-conv* [*simp*]: *hsnd* $\langle a,b\rangle$ = *b*
by (*simp add*: *hsnd-def*)

lemma *hsplit* [*simp*]: *hsplit c* $\langle a,b\rangle$ = *c a b*
by (*simp add*: *hsplit-def*)

1.4 Unions, Comprehensions, Intersections

1.4.1 Unions

Theorem 1.5 (Existence of the union of two sets).

lemma *binary-union*: $\exists z.\ \forall u.\ u \in z \longleftrightarrow u \in x \mid u \in y$
proof (*induct x rule*: *hf-induct*)
case *0* **thus** *?case* **by** *auto*
next
case (*hinsert a b*) **thus** *?case* **by** (*metis hmem-hinsert*)
qed

Theorem 1.6 (Existence of the union of a set of sets).

lemma *union-of-set*: $\exists z.\ \forall u.\ u \in z \longleftrightarrow (\exists y.\ y \in x\ \&\ u \in y)$
proof (*induct x rule*: *hf-induct*)
case *0* **thus** *?case* **by** (*metis hmem-hempty*)
next
case (*hinsert a b*)
then show *?case*
by (*metis hmem-hinsert binary-union* [*of a*])
qed

1.4.2 Set comprehensions

Theorem 1.7, comprehension scheme

lemma *comprehension*: $\exists z.\ \forall u.\ u \in z \longleftrightarrow u \in x\ \&\ P\ u$
proof (*induct x rule*: *hf-induct*)
case *0* **thus** *?case* **by** (*metis hmem-hempty*)
next
case (*hinsert a b*) **thus** *?case* **by** (*metis hmem-hinsert*)
qed

definition *HCollect* :: $(hf \Rightarrow bool) \Rightarrow hf \Rightarrow hf$ — comprehension
  where $HCollect\ P\ A = (THE\ z.\ \forall u.\ u \in z = (P\ u\ \&\ u \in A))$

syntax
  $\text{-}HCollect :: idt \Rightarrow hf \Rightarrow bool \Rightarrow hf \quad ((1\{\!|\text{-} <:/ \text{-}./ \text{-}|\!\}))$
syntax (*xsymbols*)
  $\text{-}HCollect :: idt \Rightarrow hf \Rightarrow bool \Rightarrow hf \quad ((1\{\!|\text{-} \in/ \text{-}./ \text{-}|\!\}))$
translations
  $\{\!|x <: A.\ P|\!\} == CONST\ HCollect\ (\%x.\ P)\ A$

lemma *HCollect-iff* [*iff*]: $hmem\ x\ (HCollect\ P\ A) \longleftrightarrow P\ x\ \&\ x \in A$
apply (*insert comprehension* [*of A P*], *clarify*)
apply (*simp add*: *HCollect-def*)
apply (*rule theI2*, *blast*)
apply (*auto simp*: *hf-ext*)
done

lemma *HCollectI*: $a \in A \Longrightarrow P\ a \Longrightarrow hmem\ a\ \{\!|x \in A.\ P\ x|\!\}$
  by *simp*

lemma *HCollectE*:
  assumes $a \in \{\!|x \in A.\ P\ x|\!\}$ **obtains** $a \in A$ $P\ a$
  using *assms* **by** *auto*

lemma *HCollect-hempty* [*simp*]: $HCollect\ P\ 0 = 0$
  by (*simp add*: *hf-ext*)

1.4.3 Union operators

instantiation *hf* :: *sup*
  begin
  definition *sup-hf* :: $hf \Rightarrow hf \Rightarrow hf$
    where $sup\text{-}hf\ a\ b = (THE\ z.\ \forall u.\ u \in z \longleftrightarrow u \in a \mid u \in b)$
  instance ..
  end

abbreviation *hunion* :: $hf \Rightarrow hf \Rightarrow hf$ (**infixl** $\sqcup$ *65*) **where**
  $hunion \equiv sup$

lemma *hunion-iff* [*iff*]: $hmem\ x\ (a \sqcup b) \longleftrightarrow x \in a \mid x \in b$
apply (*insert binary-union* [*of a b*], *clarify*)
apply (*simp add*: *sup-hf-def*)
apply (*rule theI2*)
apply (*auto simp*: *hf-ext*)
done

definition *HUnion* :: $hf \Rightarrow hf \qquad (\bigsqcup\text{-}\ [900]\ 900)$
  where $HUnion\ A = (THE\ z.\ \forall u.\ u \in z \longleftrightarrow (\exists y.\ y \in A\ \&\ u \in y))$

lemma *HUnion-iff* [*iff*]: *hmem* x ($\bigsqcup$ A) $\longleftrightarrow$ ($\exists\, y.\ y \in A$ & $x \in y$)
apply (*insert union-of-set* [*of* A], *clarify*)
apply (*simp add*: *HUnion-def*)
apply (*rule theI2*)
apply (*auto simp*: *hf-ext*)
done

lemma *HUnion-hempty* [*simp*]: $\bigsqcup\ 0 = 0$
by (*simp add*: *hf-ext*)

lemma *HUnion-hinsert* [*simp*]: $\bigsqcup(A \triangleleft a) = a \sqcup \bigsqcup A$
by (*auto simp*: *hf-ext*)

lemma *HUnion-hunion* [*simp*]: $\bigsqcup(A \sqcup B) = \bigsqcup A \sqcup \bigsqcup B$
by *blast*

1.4.4 Definition 1.8, Intersections

instantiation *hf* :: *inf*
begin

definition *inf-hf* :: *hf* $\Rightarrow$ *hf* $\Rightarrow$ *hf*
where *inf-hf* a b = $\{\!|x \in a.\ x \in b|\!\}$

instance ..

end

abbreviation *hinter* :: *hf* $\Rightarrow$ *hf* $\Rightarrow$ *hf* (**infixl** $\sqcap$ *70*) **where**
hinter $\equiv$ *inf*

lemma *hinter-iff* [*iff*]: *hmem* u ($x \sqcap y$) $\longleftrightarrow$ $u \in x$ & $u \in y$
by (*metis HCollect-iff inf-hf-def*)

definition *HInter* :: *hf* $\Rightarrow$ *hf* ($\bigsqcap$- [*900*] *900*)
where *HInter*(A) = $\{\!|x \in$ *HUnion*(A). $\forall\, y.\ y \in A \longrightarrow x \in y|\!\}$

lemma *HInter-hempty* [*iff*]: $\bigsqcap\ 0 = 0$
by (*metis HCollect-hempty HUnion-hempty HInter-def*)

lemma *HInter-iff* [*simp*]: $A \neq 0 \Longrightarrow$ *hmem* x ($\bigsqcap$ A) $\longleftrightarrow$ ($\forall\, y.\ y \in A \longrightarrow x \in y$)
by (*auto simp*: *HInter-def*)

lemma *HInter-hinsert* [*simp*]: $A \neq 0 \Longrightarrow \bigsqcap(A \triangleleft a) = a \sqcap \bigsqcap A$
by (*auto simp*: *hf-ext HInter-iff* [*OF hinsert-nonempty*])

1.4.5 Set Difference

instantiation *hf* :: *minus*

begin
definition *minus-hf* **where** $minus\ A\ B = \{\!|x \in A.\ \neg\ x \in B|\!\}$
instance proof qed
end

lemma *hdiff-iff* [*iff*]: $hmem\ u\ (x - y) \longleftrightarrow u \in x\ \&\ \neg\ u \in y$
by (*auto simp*: *minus-hf-def*)

lemma *hdiff-zero* [*simp*]: **fixes** $x :: hf$ **shows** $(x - 0) = x$
by *blast*

lemma *zero-hdiff* [*simp*]: **fixes** $x :: hf$ **shows** $(0 - x) = 0$
by *blast*

lemma *hdiff-insert*: $A - (B \triangleleft a) = A - B - \{\!|a|\!\}$
by *blast*

lemma *hinsert-hdiff-if*:
$(A \triangleleft x) - B = (\mathit{if}\ x \in B\ \mathit{then}\ A - B\ \mathit{else}\ (A - B) \triangleleft x)$
by *auto*

1.5 Replacement

Theorem 1.9 (Replacement Scheme).

lemma *replacement*:
$(\forall u\ v\ v'.\ u \in x \longrightarrow R\ u\ v \longrightarrow R\ u\ v' \longrightarrow v'=v) \Longrightarrow \exists z.\ \forall v.\ v \in z \longleftrightarrow (\exists u.\ u \in x\ \&\ R\ u\ v)$
proof (*induct x rule*: *hf-induct*)
case *0* **thus** *?case*
by (*metis hmem-hempty*)
next
case (*hinsert a b*) **thus** *?case*
by *simp* (*metis hmem-hinsert*)
qed

lemma *replacement-fun*: $\exists z.\ \forall v.\ v \in z \longleftrightarrow (\exists u.\ u \in x\ \&\ v = f\ u)$
by (*rule replacement* [**where** $R = \lambda u\ v.\ v = f\ u$]) *auto*

definition *PrimReplace* $:: hf \Rightarrow (hf \Rightarrow hf \Rightarrow bool) \Rightarrow hf$
where $PrimReplace\ A\ R = (THE\ z.\ \forall v.\ v \in z \longleftrightarrow (\exists u.\ u \in A\ \&\ R\ u\ v))$

definition *Replace* $:: hf \Rightarrow (hf \Rightarrow hf \Rightarrow bool) \Rightarrow hf$
where $Replace\ A\ R = PrimReplace\ A\ (\lambda x\ y.\ (\exists!z.\ R\ x\ z)\ \&\ R\ x\ y)$

definition *RepFun* $:: hf \Rightarrow (hf \Rightarrow hf) \Rightarrow hf$
where $RepFun\ A\ f = Replace\ A\ (\lambda x\ y.\ y = f\ x)$

syntax
  -HReplace :: [*pttrn*, *pttrn*, *hf*, *bool*] ⇒ *hf* ((*1*{|- ./ -<: -, -|}))
  -HRepFun :: [*hf*, *pttrn*, *hf*] ⇒ *hf* ((*1*{|- ./ -<: -|}) [*51*,*0*,*51*])
  -HINTER :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3INT* -<:-./ -) *10*)
  -HUNION :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3UN* -<:-./ -) *10*)

syntax (*xsymbols*)
  -HReplace :: [*pttrn*, *pttrn*, *hf*, *bool*] ⇒ *hf* ((*1*⦃- ./ - ∈ -, -⦄))
  -HRepFun :: [*hf*, *pttrn*, *hf*] ⇒ *hf* ((*1*⦃- ./ - ∈ -⦄) [*51*,*0*,*51*])
  -HUNION :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3*⨆-∈-./ -) *10*)
  -HINTER :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3*⨅-∈-./ -) *10*)

syntax (*HTML* **output**)
  -HReplace :: [*pttrn*, *pttrn*, *hf*, *bool*] ⇒ *hf* ((*1*⦃- ./ - ∈ -, -⦄))
  -HRepFun :: [*hf*, *pttrn*, *hf*] ⇒ *hf* ((*1*⦃- ./ - ∈ -⦄) [*51*,*0*,*51*])
  -HUNION :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3*⨆-∈-./ -) *10*)
  -HINTER :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3*⨅-∈-./ -) *10*)

translations
  {|*y*. *x*<:*A*, *Q*|} == *CONST Replace A* (%*x y*. *Q*)
  {|*b*. *x*<:*A*|} == *CONST RepFun A* (%*x*. *b*)
  INT x<:*A*. *B* == *CONST HInter*(*CONST RepFun A* (%*x*. *B*))
  UN x<:*A*. *B* == *CONST HUnion*(*CONST RepFun A* (%*x*. *B*))

lemma *PrimReplace-iff*:
  assumes *sv*: ∀ *u v v*′. *u* ∈ *A* ⟶ *R u v* ⟶ *R u v*′ ⟶ *v*′=*v*
  shows *v* ∈ (*PrimReplace A R*) ⟷ (∃ *u*. *u* ∈ *A* & *R u v*)
apply (*insert replacement* [*OF sv*], *clarify*)
apply (*simp add*: *PrimReplace-def*)
apply (*rule theI2*)
apply (*auto simp*: *hf-ext*)
done

lemma *Replace-iff* [*iff*]:
  v ∈ *Replace A R* ⟷ (∃ *u*. *u* ∈ *A* & *R u v* & (∀ *y*. *R u y* ⟶ *y*=*v*))
apply (*simp add*: *Replace-def*)
apply (*subst PrimReplace-iff*, *auto*)
done

lemma *Replace-0* [*simp*]: *Replace 0 R* = *0*
  by *blast*

lemma *Replace-hunion* [*simp*]: *Replace* (*A* ⊔ *B*) *R* = *Replace A R* ⊔ *Replace B R*
  by *blast*

lemma *Replace-cong* [*cong*]:
  ⟦ *A*=*B*; !!*x y*. *x* ∈ *B* ⟹ *P x y* ⟷ *Q x y* ⟧ ⟹ *Replace A P* = *Replace B Q*
  by (*simp add*: *hf-ext cong*: *conj-cong*)

lemma *RepFun-iff* [*iff*]: $v \in (\textit{RepFun } A\ f) \longleftrightarrow (\exists\, u.\ u \in A\ \&\ v = f\ u)$
by (*auto simp*: *RepFun-def*)

lemma *RepFun-cong* [*cong*]:
$[\![A{=}B;\ !!x.\ x \in B \Longrightarrow f(x){=}g(x)]\!] \Longrightarrow \textit{RepFun } A\ f = \textit{RepFun } B\ g$
by (*simp add*: *RepFun-def*)

lemma *triv-RepFun* [*simp*]: $\textit{RepFun } A\ (\lambda x.\ x) = A$
by *blast*

lemma *RepFun-0* [*simp*]: $\textit{RepFun } 0\ f = 0$
by *blast*

lemma *RepFun-hinsert* [*simp*]: $\textit{RepFun } (\textit{hinsert } a\ b)\ f = \textit{hinsert } (f\ a)\ (\textit{RepFun } b\ f)$
by *blast*

lemma *RepFun-hunion* [*simp*]:
$\textit{RepFun } (A \sqcup B)\ f = \textit{RepFun } A\ f\ \sqcup\ \textit{RepFun } B\ f$
by *blast*

1.6 Subset relation and the Lattice Properties

Definition 1.10 (Subset relation).

instantiation *hf* :: *order*
begin
definition *less-eq-hf* **where** $A \le B \longleftrightarrow (\forall\, x.\ x \in A \longrightarrow x \in B)$

definition *less-hf* **where** $A < B \longleftrightarrow A \le B\ \&\ A \ne (B{::}hf)$

instance proof qed (*auto simp*: *less-eq-hf-def less-hf-def*)
end

1.6.1 Rules for subsets

lemma *hsubsetI* [*intro*!]:
$(!!x.\ x{\in}A \Longrightarrow x{\in}B) \Longrightarrow A \le B$
by (*simp add*: *less-eq-hf-def*)

Classical elimination rule

lemma *hsubsetCE* [*elim*]: $[\![A \le B;\ {\sim}(c{\in}A) \Longrightarrow P;\ c{\in}B \Longrightarrow P]\!] \Longrightarrow P$
by (*auto simp*: *less-eq-hf-def*)

Rule in Modus Ponens style

lemma *hsubsetD* [*elim*]: $[\![A \le B;\ c{\in}A]\!] \Longrightarrow c{\in}B$
by (*simp add*: *less-eq-hf-def*)

Sometimes useful with premises in this order

lemma *rev-hsubsetD*: $[\![c{\in}A;\ A{\le}B]\!] \Longrightarrow c{\in}B$
  by *blast*

lemma *contra-hsubsetD*: $[\![A \le B;\ c \notin B]\!] \Longrightarrow c \notin A$
  by *blast*

lemma *rev-contra-hsubsetD*: $[\![c \notin B;\ A \le B]\!] \Longrightarrow c \notin A$
  by *blast*

lemma *hf-equalityE*:
  fixes $A :: hf$ **shows** $A = B \Longrightarrow (A \le B \Longrightarrow B \le A \Longrightarrow P) \Longrightarrow P$
  by (*metis order-refl*)

1.6.2 Lattice properties

instantiation *hf* :: *distrib-lattice*
  begin
  instance proof qed (*auto simp*: *less-eq-hf-def less-hf-def inf-hf-def*)
  end

instantiation *hf* :: *bounded-lattice-bot*
  begin
  definition *bot-hf* **where** *bot-hf* = (*0*::*hf*)
  instance proof qed (*auto simp*: *less-eq-hf-def bot-hf-def*)
  end

lemma *hinter-hempty-left* [*simp*]: $0 \sqcap A = 0$
  by (*metis bot-hf-def inf-bot-left*)

lemma *hinter-hempty-right* [*simp*]: $A \sqcap 0 = 0$
  by (*metis bot-hf-def inf-bot-right*)

lemma *hunion-hempty-left* [*simp*]: $0 \sqcup A = A$
  by (*metis bot-hf-def sup-bot-left*)

lemma *hunion-hempty-right* [*simp*]: $A \sqcup 0 = A$
  by (*metis bot-hf-def sup-bot-right*)

lemma *less-eq-hempty* [*simp*]: $u \le 0 \longleftrightarrow u = (0{::}hf)$
  by (*metis hempty-iff less-eq-hf-def*)

lemma *less-eq-insert1-iff* [*iff*]: $(hinsert\ x\ y) \le z \longleftrightarrow x \in z\ \&\ y \le z$
  by (*auto simp*: *less-eq-hf-def*)

lemma *less-eq-insert2-iff*:
  $z \le (hinsert\ x\ y) \longleftrightarrow z \le y \vee (\exists u.\ hinsert\ x\ u = z \wedge {\sim}\ x \in u \wedge u \le y)$
proof ($cases\ x \in z$)
  case *True*
  hence *u*: $hinsert\ x\ (z - \{\!|x|\!\}) = z$ **by** *auto*

show *?thesis*
proof
assume $z \leq (hinsert\ x\ y)$
thus $z \leq y \vee (\exists u.\ hinsert\ x\ u = z \wedge \neg\ x \in u \wedge u \leq y)$
by (*simp add*: *less-eq-hf-def*) (*metis u hdiff-iff hmem-hinsert*)
next
assume $z \leq y \vee (\exists u.\ hinsert\ x\ u = z \wedge \neg\ x \in u \wedge u \leq y)$
thus $z \leq (hinsert\ x\ y)$
by (*auto simp*: *less-eq-hf-def*)
qed
next
case *False* **thus** *?thesis*
by (*metis hmem-hinsert less-eq-hf-def*)
qed

lemma *zero-le* [*simp*]: $0 \leq (x{::}hf)$
by *blast*

lemma *hinsert-eq-sup*: $b \triangleleft a = b \sqcup \{\!|a|\!\}$
by *blast*

lemma *hunion-hinsert-left*: $hinsert\ x\ A \sqcup B = hinsert\ x\ (A \sqcup B)$
by *blast*

lemma *hunion-hinsert-right*: $B \sqcup hinsert\ x\ A = hinsert\ x\ (B \sqcup A)$
by *blast*

lemma *hinter-hinsert-left*: $hinsert\ x\ A \sqcap B = (if\ x \in B\ then\ hinsert\ x\ (A \sqcap B)$ $else\ A \sqcap B)$
by *auto*

lemma *hinter-hinsert-right*: $B \sqcap hinsert\ x\ A = (if\ x \in B\ then\ hinsert\ x\ (B \sqcap A)$ $else\ B \sqcap A)$
by *auto*

1.7 Foundation, Cardinality, Powersets

1.7.1 Foundation

Theorem 1.13: Foundation (Regularity) Property.

lemma *foundation*:
assumes z: $z \neq 0$ **shows** $\exists w.\ w \in z\ \&\ w \sqcap z = 0$
proof –
{ **fix** x
assume z: $(\forall w.\ w \in z \longrightarrow w \sqcap z \neq 0)$
have $\sim\ x \in z \wedge x \sqcap z = 0$
proof (*induction x rule*: *hf-induct*)
case *0* **thus** *?case*
by (*metis hinter-hempty-left z*)

  next
    case (*hinsert x y*) **thus** *?case*
      by (*metis hinter-hinsert-left z*)
  qed
 }
 thus *?thesis* **using** *z*
  by (*metis z hempty-iff*)
qed

lemma *hmem-not-refl*: ~ $(x \in x)$
 using *foundation* [*of* $\{\!|x|\!\}$]
 by (*metis hinter-iff hmem-hempty hmem-hinsert*)

lemma *hmem-not-sym*: ~ $(x \in y \wedge y \in x)$
 using *foundation* [*of* $\{\!|x,y|\!\}$]
 by (*metis hinter-iff hmem-hempty hmem-hinsert*)

lemma *hmem-ne*: $x \in y \Longrightarrow x \neq y$
 by (*metis hmem-not-refl*)

lemma *hmem-Sup-ne*: $x <: y \Longrightarrow \bigsqcup x \neq y$
 by (*metis HUnion-iff hmem-not-sym*)

lemma *hpair-neq-fst*: $\langle a,b\rangle \neq a$
 by (*metis hpair-def hinsert-iff hmem-not-sym*)

lemma *hpair-neq-snd*: $\langle a,b\rangle \neq b$
 by (*metis hpair-def hinsert-iff hmem-not-sym*)

lemma *hpair-nonzero* [*simp*]: $\langle x,y\rangle \neq 0$
 by (*auto simp*: *hpair-def*)

lemma *zero-notin-hpair*: ~ $0 \in \langle x,y\rangle$
 by (*auto simp*: *hpair-def*)

1.7.2 Cardinality

First we need to hack the underlying representation

lemma *hfset-0*: *hfset 0* = {}
 by (*metis Zero-hf-def finite.emptyI hfset-HF*)

lemma *hfset-hinsert*: *hfset* $(b \triangleleft a)$ = *insert a* (*hfset b*)
 by (*metis finite-insert hinsert-def HF.finite-hfset hfset-HF*)

lemma *hfset-hdiff*: *hfset* $(x - y)$ = *hfset x* − *hfset y*
proof (*induct x arbitrary*: *y rule*: *hf-induct*)
 case *0* **thus** *?case*
  by (*simp add*: *hfset-0*)
next

case (*hinsert a b*) **thus** *?case*
  by (*simp add*: *hfset-hinsert Set.insert-Diff-if hinsert-hdiff-if hmem-def*)
qed

definition *hcard* :: *hf* ⇒ *nat*
  where *hcard x* = *card* (*hfset x*)

lemma *hcard-0* [*simp*]: *hcard 0* = *0*
  by (*simp add*: *hcard-def hfset-0*)

lemma *hcard-hinsert-if*: *hcard* (*hinsert x y*) = (*if x* ∈ *y then hcard y else Suc* (*hcard y*))
  by (*simp add*: *hcard-def hfset-hinsert card-insert-if hmem-def*)

lemma *hcard-union-inter*: *hcard* (*x* ⊔ *y*) + *hcard* (*x* ⊓ *y*) = *hcard x* + *hcard y*
  apply (*induct x arbitrary*: *y rule*: *hf-induct*)
  apply (*auto simp*: *hcard-hinsert-if hunion-hinsert-left hinter-hinsert-left*)
  done

lemma *hcard-hdiff1-less*: *x* ∈ *z* ⟹ *hcard* (*z* − ⦃*x*⦄) < *hcard z*
  by (*simp add*: *hcard-def hfset-hdiff hfset-hinsert hfset-0*)
    (*metis card-Diff1-less finite-hfset hmem-def*)

1.7.3 Powerset Operator

Theorem 1.11 (Existence of the power set).

lemma *powerset*: ∃ *z*. ∀ *u*. *u* ∈ *z* ⟷ *u* ≤ *x*
proof (*induction x rule*: *hf-induct*)
 case *0* **thus** *?case*
  by (*metis hmem-hempty hmem-hinsert less-eq-hempty*)
next
  case (*hinsert a b*)
  then obtain *Pb* **where** *Pb*: ∀ *u*. *u* ∈ *Pb* ⟷ *u* ≤ *b*
    by *auto*
  obtain *RPb* **where** *RPb*: ∀ *v*. *v* ∈ *RPb* ⟷ (∃ *u*. *u* ∈ *Pb* & *v* = *hinsert a u*)
    using *replacement-fun* **..**
  thus *?case* **using** *Pb binary-union* [*of Pb RPb*]
    apply (*simp add*: *less-eq-insert2-iff*, *clarify*)
    apply (*rule-tac x*=*z* **in** *exI*)
    apply (*metis hinsert.hyps less-eq-hf-def*)
    done
qed

definition *HPow* :: *hf* ⇒ *hf*
  where *HPow x* = (*THE z*. ∀ *u*. *u* ∈ *z* ⟷ *u* ≤ *x*)

lemma *HPow-iff* [*iff*]: *u* ∈ *HPow x* ⟷ *u* ≤ *x*
apply (*insert powerset* [*of x*], *clarify*)
apply (*simp add*: *HPow-def*)

apply (*rule theI2*)
apply (*auto simp*: *hf-ext*)
done

lemma *HPow-mono*: $x \le y \Longrightarrow HPow\ x \le HPow\ y$
  by (*metis HPow-iff less-eq-hf-def order-trans*)

lemma *HPow-mono-strict*: $x < y \Longrightarrow HPow\ x < HPow\ y$
  by (*metis HPow-iff HPow-mono less-le-not-le order-eq-iff*)

lemma *HPow-mono-iff* [*simp*]: $HPow\ x \le HPow\ y \longleftrightarrow x \le y$
  by (*metis HPow-iff HPow-mono hsubsetCE order-refl*)

lemma *HPow-mono-strict-iff* [*simp*]: $HPow\ x < HPow\ y \longleftrightarrow x < y$
  by (*metis HPow-mono-iff less-le-not-le*)

1.8 Bounded Quantifiers

definition *HBall* :: $hf \Rightarrow (hf \Rightarrow bool) \Rightarrow bool$ **where**
  $HBall\ A\ P \longleftrightarrow (\forall x.\ x <: A \longrightarrow P\ x)$ — bounded universal quantifiers

definition *HBex* :: $hf \Rightarrow (hf \Rightarrow bool) \Rightarrow bool$ **where**
  $HBex\ A\ P \longleftrightarrow (\exists x.\ x <: A \wedge P\ x)$ — bounded existential quantifiers

syntax
  -HBall :: *pttrn* ⇒ *hf* ⇒ *bool* ⇒ *bool* ((*3ALL* -<:-./ -) [*0*, *0*, *10*] *10*)
  -HBex :: *pttrn* ⇒ *hf* ⇒ *bool* ⇒ *bool* ((*3EX* -<:-./ -) [*0*, *0*, *10*] *10*)
  -HBex1 :: *pttrn* ⇒ *hf* ⇒ *bool* ⇒ *bool* ((*3EX*! -<:-./ -) [*0*, *0*, *10*] *10*)

syntax (*xsymbols*)
  -HBall :: *pttrn* ⇒ *hf* ⇒ *bool* ⇒ *bool* ((*3*∀ -∈-./ -) [*0*, *0*, *10*] *10*)
  -HBex :: *pttrn* ⇒ *hf* ⇒ *bool* ⇒ *bool* ((*3*∃ -∈-./ -) [*0*, *0*, *10*] *10*)
  -HBex1 :: *pttrn* ⇒ *hf* ⇒ *bool* ⇒ *bool* ((*3*∃!-∈-./ -) [*0*, *0*, *10*] *10*)

syntax (*HTML* **output**)
  -HBall :: *pttrn* ⇒ *hf* ⇒ *bool* ⇒ *bool* ((*3*∀ -∈-./ -) [*0*, *0*, *10*] *10*)
  -HBex :: *pttrn* ⇒ *hf* ⇒ *bool* ⇒ *bool* ((*3*∃ -∈-./ -) [*0*, *0*, *10*] *10*)
  -HBex1 :: *pttrn* ⇒ *hf* ⇒ *bool* ⇒ *bool* ((*3*∃!-∈-./ -) [*0*, *0*, *10*] *10*)

translations
  ALL x<:*A*. *P* == *CONST HBall A* (%*x*. *P*)
  EX x<:*A*. *P* == *CONST HBex A* (%*x*. *P*)
  EX! *x*<:*A*. *P* => *EX*! *x*. *x*:*A* & *P*

lemma *hball-cong* [*cong*]:
  $[\![A=A';\ !!x.\ x \in A' \Longrightarrow P(x) \longleftrightarrow P'(x)]\!] \Longrightarrow (\forall x{\in}A.\ P(x)) \longleftrightarrow (\forall x{\in}A'.\ P'(x))$
  by (*simp add*: *HBall-def*)

lemma *hballI* [*intro*!]: $(!!x.\ x{<}{:}A \Longrightarrow P\ x) \Longrightarrow ALL\ x{<}{:}A.\ P\ x$
  by (*simp add*: *HBall-def*)

lemma *hbspec* [*dest?*]: $ALL\ x{<}{:}A.\ P\ x \Longrightarrow x{<}{:}A \Longrightarrow P\ x$
  by (*simp add*: *HBall-def*)

lemma *hballE* [*elim*]: $ALL\ x{<}{:}A.\ P\ x \Longrightarrow (P\ x \Longrightarrow Q) \Longrightarrow ({}^{\sim}\ x\ {<}{:}\ A \Longrightarrow Q) \Longrightarrow Q$
  by (*unfold HBall-def*) *blast*

lemma *hbex-cong* [*cong*]:
  $[\![\ A{=}A';\ \ !!x.\ x \in A' \Longrightarrow P(x) \longleftrightarrow P'(x)\ ]\!]\ \Longrightarrow (\exists x{\in}A.\ P(x)) \longleftrightarrow (\exists x{\in}A'.\ P'(x))$
  by (*simp add*: *HBex-def cong*: *conj-cong*)

lemma *hbexI* [*intro*]: $P\ x \Longrightarrow x{<}{:}A \Longrightarrow EX\ x{<}{:}A.\ P\ x$
  by (*unfold HBex-def*) *blast*

lemma *rev-hbexI* [*intro?*]: $x{<}{:}A \Longrightarrow P\ x \Longrightarrow EX\ x{<}{:}A.\ P\ x$
  by (*unfold HBex-def*) *blast*

lemma *bexCI*: $(ALL\ x{<}{:}A.\ {}^{\sim}P\ x \Longrightarrow P\ a) \Longrightarrow a{<}{:}A \Longrightarrow EX\ x{<}{:}A.\ P\ x$
  by (*unfold HBex-def*) *blast*

lemma *hbexE* [*elim*!]: $EX\ x{<}{:}A.\ P\ x \Longrightarrow (!!x.\ x{<}{:}A \Longrightarrow P\ x \Longrightarrow Q) \Longrightarrow Q$
  by (*unfold HBex-def*) *blast*

lemma *hball-triv* [*simp*]: $(ALL\ x{<}{:}A.\ P) = ((EX\ x.\ x{<}{:}A)\ {-}{-}{>}\ P)$
  — Trival rewrite rule.
  by (*simp add*: *HBall-def*)

lemma *hbex-triv* [*simp*]: $(EX\ x{<}{:}A.\ P) = ((EX\ x.\ x{<}{:}A)\ \&\ P)$
  — Dual form for existentials.
  by (*simp add*: *HBex-def*)

lemma *hbex-triv-one-point1* [*simp*]: $(EX\ x{<}{:}A.\ x = a) = (a{<}{:}A)$
  by *blast*

lemma *hbex-triv-one-point2* [*simp*]: $(EX\ x{<}{:}A.\ a = x) = (a{<}{:}A)$
  by *blast*

lemma *hbex-one-point1* [*simp*]: $(EX\ x{<}{:}A.\ x = a\ \&\ P\ x) = (a{<}{:}A\ \&\ P\ a)$
  by *blast*

lemma *hbex-one-point2* [*simp*]: $(EX\ x{<}{:}A.\ a = x\ \&\ P\ x) = (a{<}{:}A\ \&\ P\ a)$
  by *blast*

lemma *hball-one-point1* [*simp*]: $(ALL\ x{<}{:}A.\ x = a\ {-}{-}{>}\ P\ x) = (a{<}{:}A\ {-}{-}{>}\ P\ a)$

by *blast*

lemma *hball-one-point2* [*simp*]: $(ALL\ x{<}{:}A.\ a = x\ {-}{-}{>}\ P\ x) = (a{<}{:}A\ {-}{-}{>}\ P\ a)$
by *blast*

lemma *hball-conj-distrib*:
$(\forall x{\in}A.\ P\ x \wedge Q\ x) \longleftrightarrow ((\forall x{\in}A.\ P\ x) \wedge (\forall x{\in}A.\ Q\ x))$
by *blast*

lemma *hbex-disj-distrib*:
$(\exists x{\in}A.\ P\ x \vee Q\ x) \longleftrightarrow ((\exists x{\in}A.\ P\ x) \vee (\exists x{\in}A.\ Q\ x))$
by *blast*

lemma *hb-all-simps* [*simp*, *no-atp*]:
$\bigwedge A\ P\ Q.\ (\forall x \in A.\ P\ x \vee Q) \longleftrightarrow ((\forall x \in A.\ P\ x) \vee Q)$
$\bigwedge A\ P\ Q.\ (\forall x \in A.\ P \vee Q\ x) \longleftrightarrow (P \vee (\forall x \in A.\ Q\ x))$
$\bigwedge A\ P\ Q.\ (\forall x \in A.\ P \longrightarrow Q\ x) \longleftrightarrow (P \longrightarrow (\forall x \in A.\ Q\ x))$
$\bigwedge A\ P\ Q.\ (\forall x \in A.\ P\ x \longrightarrow Q) \longleftrightarrow ((\exists x \in A.\ P\ x) \longrightarrow Q)$
$\bigwedge P.\ (\forall x \in \mathit{0}.\ P\ x) \longleftrightarrow \mathit{True}$
$\bigwedge a\ B\ P.\ (\forall x \in B \triangleleft a.\ P\ x) \longleftrightarrow (P\ a \wedge (\forall x \in B.\ P\ x))$
$\bigwedge P\ Q.\ (\forall x \in \mathit{HCollect}\ Q\ A.\ P\ x) \longleftrightarrow (\forall x \in A.\ Q\ x \longrightarrow P\ x)$
$\bigwedge A\ P.\ (\neg\ (\forall x \in A.\ P\ x)) \longleftrightarrow (\exists x \in A.\ \neg\ P\ x)$
by *auto*

lemma *hb-ex-simps* [*simp*, *no-atp*]:
$\bigwedge A\ P\ Q.\ (\exists x \in A.\ P\ x \wedge Q) \longleftrightarrow ((\exists x \in A.\ P\ x) \wedge Q)$
$\bigwedge A\ P\ Q.\ (\exists x \in A.\ P \wedge Q\ x) \longleftrightarrow (P \wedge (\exists x \in A.\ Q\ x))$
$\bigwedge P.\ (\exists x \in \mathit{0}.\ P\ x) \longleftrightarrow \mathit{False}$
$\bigwedge a\ B\ P.\ (\exists x \in B \triangleleft a.\ P\ x) \longleftrightarrow (P\ a \mid (\exists x \in B.\ P\ x))$
$\bigwedge P\ Q.\ (\exists x \in \mathit{HCollect}\ Q\ A.\ P\ x) \longleftrightarrow (\exists x \in A.\ Q\ x \wedge P\ x)$
$\bigwedge A\ P.\ (\neg(\exists x \in A.\ P\ x)) \longleftrightarrow (\forall x \in A.\ \neg\ P\ x)$
by *auto*

lemma *le-HCollect-iff*: $A \le \{\!|x \in B.\ P\ x|\!\} \longleftrightarrow A \le B \wedge (\forall x \in A.\ P\ x)$
by *blast*

end

Chapter 2

Relations, Families, Ordinals

theory *Ordinal* **imports** *HF*
begin

2.1 Relations and Functions

definition *is-hpair* :: $hf \Rightarrow bool$
 where *is-hpair* $z = (\exists\, x\ y.\ z = \langle x,y\rangle)$

definition *hconverse* :: $hf \Rightarrow hf$
 where *hconverse*$(r) = \{\!|z.\ w \in r,\ \exists\, x\ y.\ w = \langle x,y\rangle\ \&\ z = \langle y,x\rangle|\!\}$

definition *hdomain* :: $hf \Rightarrow hf$
 where *hdomain*$(r) = \{\!|x.\ w \in r,\ \exists\, y.\ w = \langle x,y\rangle|\!\}$

definition *hrange* :: $hf \Rightarrow hf$
 where *hrange*$(r) =$ *hdomain*(*hconverse*(r))

definition *hrelation* :: $hf \Rightarrow bool$
 where *hrelation*$(r) = (\forall\, z.\ z \in r \longrightarrow$ *is-hpair* $z)$

definition *hrestrict* :: $hf \Rightarrow hf \Rightarrow hf$
 — Restrict the relation r to the domain A
 where *hrestrict* $r\ A = \{\!|z \in r.\ \exists\, x \in A.\ \exists\, y.\ z = \langle x,y\rangle|\!\}$

definition *nonrestrict* :: $hf \Rightarrow hf \Rightarrow hf$
 where *nonrestrict* $r\ A = \{\!|z \in r.\ \forall\, x \in A.\ \forall\, y.\ z \neq \langle x,y\rangle|\!\}$

definition *hfunction* :: $hf \Rightarrow bool$
 where *hfunction*$(r) = (\forall\, x\ y.\ \langle x,y\rangle \in r \longrightarrow (\forall\, y'.\ \langle x,y'\rangle \in r \longrightarrow y{=}y'))$

definition *app* :: $hf \Rightarrow hf \Rightarrow hf$
 where *app* $f\ x = (THE\ y.\ \langle x,\ y\rangle \in f)$

lemma *hrestrict-iff* [*iff*]:

$z \in$ *hrestrict r A* $\longleftrightarrow z \in r$ & $(\exists\ x\ y.\ z = \langle x,\ y\rangle$ & $x \in A)$
by (*auto simp*: *hrestrict-def*)

lemma *hrelation-0* [*simp*]: *hrelation 0*
by (*force simp add*: *hrelation-def*)

lemma *hrelation-restr* [*iff*]: *hrelation* (*hrestrict r x*)
by (*metis hrelation-def hrestrict-iff is-hpair-def*)

lemma *hrelation-hunion* [*simp*]: *hrelation* $(f \sqcup g) \longleftrightarrow$ *hrelation f* $\wedge$ *hrelation g*
by (*auto simp*: *hrelation-def*)

lemma *hfunction-restr*: *hfunction r* $\Longrightarrow$ *hfunction* (*hrestrict r x*)
by (*auto simp*: *hfunction-def hrestrict-def*)

lemma *hdomain-restr* [*simp*]: *hdomain* (*hrestrict r x*) = *hdomain r* $\sqcap$ *x*
by (*force simp add*: *hdomain-def hrestrict-def*)

lemma *hdomain-0* [*simp*]: *hdomain 0* = *0*
by (*force simp add*: *hdomain-def*)

lemma *hdomain-ins* [*simp*]: *hdomain* $(r \triangleleft \langle x,\ y\rangle)$ = *hdomain r* $\triangleleft$ *x*
by (*force simp add*: *hdomain-def*)

lemma *hdomain-hunion* [*simp*]: *hdomain* $(f \sqcup g)$ = *hdomain f* $\sqcup$ *hdomain g*
by (*simp add*: *hdomain-def*)

lemma *hdomain-not-mem* [*iff*]: $\neg$ $\langle$*hdomain r*, $a\rangle \in r$
by (*metis hdomain-ins hinter-hinsert-right hmem-hinsert hmem-not-refl*
hunion-hinsert-right sup-inf-absorb)

lemma *app-singleton* [*simp*]: *app* $\{\!|\langle x,\ y\rangle|\!\}$ $x = y$
by (*simp add*: *app-def*)

lemma *app-equality*: *hfunction f* $\Longrightarrow \langle x,\ y\rangle$ <: *f* $\Longrightarrow$ *app f x* = *y*
by (*auto simp*: *app-def hfunction-def intro*: *the1I2*)

lemma *app-ins2*: $x' \neq x \Longrightarrow$ *app* $(f \triangleleft \langle x,\ y\rangle)\ x'$ = *app f* x'
by (*simp add*: *app-def*)

lemma *hfunction-0* [*simp*]: *hfunction 0*
by (*force simp add*: *hfunction-def*)

lemma *hfunction-ins*: *hfunction f* $\Longrightarrow$ ~ *x* <: *hdomain f* $\Longrightarrow$ *hfunction* $(f \triangleleft \langle x,$
$y\rangle)$
by (*auto simp*: *hfunction-def hdomain-def*)

lemma *hdomainI*: $\langle x,\ y\rangle \in f \Longrightarrow x \in$ *hdomain f*
by (*auto simp*: *hdomain-def*)

lemma *hfunction-hunion*: $hdomain\ f \sqcap hdomain\ g = 0$
$\quad\Longrightarrow hfunction\ (f \sqcup g) \longleftrightarrow hfunction\ f \wedge hfunction\ g$
by (*auto simp*: *hfunction-def*) (*metis hdomainI hinter-iff hmem-hempty*)+

lemma *app-hrestrict* [*simp*]: $x \in A \Longrightarrow app\ (hrestrict\ f\ A)\ x = app\ f\ x$
by (*simp add*: *hrestrict-def app-def*)

2.2 Operations on families of sets

definition *HLambda* :: $hf \Rightarrow (hf \Rightarrow hf) \Rightarrow hf$
where $HLambda\ A\ b = RepFun\ A\ (\lambda x.\ \langle x,\ b\ x\rangle)$

definition *HSigma* :: $hf \Rightarrow (hf \Rightarrow hf) \Rightarrow hf$
where $HSigma\ A\ B = (\bigsqcup x{\in}A.\ \bigsqcup y{\in}B(x).\ \{\!|\langle x,y\rangle|\!\})$

definition *HPi* :: $hf \Rightarrow (hf \Rightarrow hf) \Rightarrow hf$
where $HPi\ A\ B = \{\!|\ f \in HPow(HSigma\ A\ B).\ A \le hdomain(f)\ \&\ hfunction(f)|\!\}$

syntax
-PROD :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3PROD* -<:-./ -) *10*)
-SUM :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3SUM* -<:-./ -) *10*)
-lam :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3lam* -<:-./ -) *10*)

syntax (*xsymbols*)
-PROD :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3*Π-∈-./ -) *10*)
-SUM :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3*Σ-∈-./ -) *10*)
-lam :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3*λ-∈-./ -) *10*)

syntax (*HTML* **output**)
-PROD :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3*Π-∈-./ -) *10*)
-SUM :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3*Σ-∈-./ -) *10*)
-lam :: [*pttrn*, *hf*, *hf*] ⇒ *hf* ((*3*λ-∈-./ -) *10*)

translations
PROD x<:A. B == CONST HPi A (*%x. B*)
SUM x<:A. B == CONST HSigma A (*%x. B*)
lam x<:A. f == CONST HLambda A (*%x. f*)

2.2.1 Rules for Unions and Intersections of families

lemma *HUN-iff* [*simp*]: $b \in (\bigsqcup x{\in}A.\ B(x)) \longleftrightarrow (\exists\, x{\in}A.\ b \in B(x))$
by *auto*

lemma *HUN-I*: $[\![\ a \in A;\ \ b \in B(a)\ ]\!] \Longrightarrow b \in (\bigsqcup x{\in}A.\ B(x))$
by *auto*

lemma *HUN-E* [*elim*!]: **assumes** $b \in (\bigsqcup x{\in}A.\ B(x))$ **obtains** x **where** $x \in A$ $b \in B(x)$
using *assms* **by** *blast*

lemma *HINT-iff*: $b \in (\bigsqcap x{\in}A.\ B(x)) \longleftrightarrow (\forall x{\in}A.\ b \in B(x))\ \&\ A{\neq}0$
by (*simp add*: *HInter-def HBall-def*) (*metis foundation hmem-hempty*)

lemma *HINT-I*: $[\![\ !!x.\ x \in A \Longrightarrow b \in B(x);\ \ A{\neq}0\ ]\!] \Longrightarrow b \in (\bigsqcap x{\in}A.\ B(x))$
by (*simp add*: *HINT-iff*)

lemma *HINT-E*: $[\![\ b \in (\bigsqcap x{\in}A.\ B(x));\ \ a \in A\ ]\!] \Longrightarrow b \in B(a)$
by (*auto simp*: *HINT-iff*)

2.2.2 Generalized Cartesian product

lemma *HSigma-iff* [*simp*]: $\langle a,b\rangle \in HSigma\ A\ B \longleftrightarrow a \in A\ \&\ b \in B(a)$
by (*force simp add*: *HSigma-def*)

lemma *HSigmaI* [*intro*!]: $[\![\ a \in A;\ \ b \in B(a)\ ]\!] \Longrightarrow \langle a,b\rangle \in HSigma\ A\ B$
by *simp*

lemmas *HSigmaD1* = *HSigma-iff* [*THEN iffD1*, *THEN conjunct1*]
lemmas *HSigmaD2* = *HSigma-iff* [*THEN iffD1*, *THEN conjunct2*]

The general elimination rule

lemma *HSigmaE* [*elim*!]:
assumes $c \in HSigma\ A\ B$
obtains $x\ y$ **where** $x \in A$ $y \in B(x)$ $c{=}\langle x,y\rangle$
using *assms* **by** (*force simp add*: *HSigma-def*)

lemma *HSigmaE2* [*elim*!]:
assumes $\langle a,b\rangle \in HSigma\ A\ B$ **obtains** $a \in A$ **and** $b \in B(a)$
using *assms* **by** *auto*

lemma *HSigma-empty1* [*simp*]: *HSigma 0 B = 0*
by *blast*

instantiation *hf* :: *times*
begin
definition *times-hf* **where**
times A B = HSigma A ($\lambda x.\ B$)
instance proof qed
end

lemma *times-iff* [*simp*]: $\langle a,b\rangle \in A * B \longleftrightarrow a \in A\ \&\ b \in B$
by (*simp add*: *times-hf-def*)

lemma *timesI* [*intro*!]: $[\![\ a \in A;\ \ b \in B\ ]\!] \Longrightarrow \langle a,b\rangle \in A * B$
by *simp*

lemmas *timesD1* = *times-iff* [*THEN iffD1*, *THEN conjunct1*]
lemmas *timesD2* = *times-iff* [*THEN iffD1*, *THEN conjunct2*]

The general elimination rule

lemma *timesE* [*elim*!]:
  assumes *c*: $c \in A * B$
  obtains *x y* **where** $x \in A$ $y \in B$ $c=\langle x,y\rangle$ **using** *c*
  by (*auto simp*: *times-hf-def*)

...and a specific one

lemma *timesE2* [*elim*!]:
  assumes $\langle a,b\rangle \in A * B$ **obtains** $a \in A$ **and** $b \in B$
using *assms*
  by *auto*

lemma *times-empty1* [*simp*]: $0 * B = (0::hf)$
  by *auto*

lemma *times-empty2* [*simp*]: $A*0 = (0::hf)$
  by *blast*

lemma *times-empty-iff*: $A*B=0 \longleftrightarrow A=0 \mid B=(0::hf)$
  by (*auto simp*: *times-hf-def hf-ext*)

instantiation *hf* :: *mult-zero*
begin
instance proof qed *auto*
end

2.3 Disjoint Sum

instantiation *hf* :: *zero-neq-one*
begin

definition
  One-hf-def: $1 = \{\!|0|\!\}$
instance proof
  qed (*auto simp*: *One-hf-def*)
end

instantiation *hf* :: *plus*
begin
definition *plus-hf* **where**
  $plus\ A\ B = (\{\!|0|\!\} * A) \sqcup (\{\!|1|\!\} * B)$
instance proof qed
end

definition *Inl* :: *hf=>hf* **where**
  $Inl(a) \equiv \langle 0,a\rangle$

definition *Inr* :: *hf=>hf* **where**
Inr$(b) \equiv \langle 1,b\rangle$

lemmas *sum-defs* = *plus-hf-def Inl-def Inr-def*

lemma *Inl-nonzero* [*simp*]:*Inl* $x \neq 0$
by (*metis Inl-def hpair-nonzero*)

lemma *Inr-nonzero* [*simp*]:*Inr* $x \neq 0$
by (*metis Inr-def hpair-nonzero*)

Introduction rules for the injections (as equivalences)

lemma *Inl-in-sum-iff* [*iff*]: *Inl*$(a) \in A+B \longleftrightarrow a \in A$
by (*auto simp*: *sum-defs*)

lemma *Inr-in-sum-iff* [*iff*]: *Inr*$(b) \in A+B \longleftrightarrow b \in B$
by (*auto simp*: *sum-defs*)

Elimination rule

lemma *sumE* [*elim*!]:
assumes *u*: $u \in A+B$
obtains *x* **where** $x \in A$ u=*Inl*(x) | *y* **where** $y \in B$ u=*Inr*(y) **using** *u*
by (*auto simp*: *sum-defs*)

Injection and freeness equivalences, for rewriting

lemma *Inl-iff* [*iff*]: *Inl*(a)=*Inl*$(b) \longleftrightarrow a=b$
by (*simp add*: *sum-defs*)

lemma *Inr-iff* [*iff*]: *Inr*(a)=*Inr*$(b) \longleftrightarrow a=b$
by (*simp add*: *sum-defs*)

lemma *Inl-Inr-iff* [*iff*]: *Inl*(a)=*Inr*$(b) \longleftrightarrow$ *False*
by (*simp add*: *sum-defs*)

lemma *Inr-Inl-iff* [*iff*]: *Inr*(b)=*Inl*$(a) \longleftrightarrow$ *False*
by (*simp add*: *sum-defs*)

lemma *sum-empty* [*simp*]: $0+0 = (0::hf)$
by (*auto simp*: *sum-defs*)

lemma *sum-iff*: $u \in A+B \longleftrightarrow (\exists x.\ x \in A\ \&\ u=$*Inl*$(x))\ |\ (\exists y.\ y \in B\ \&\ u=$*Inr*$(y))$
by *blast*

lemma *sum-subset-iff*:
fixes *A* :: *hf* **shows** $A+B \leq C+D \longleftrightarrow A \leq C\ \&\ B \leq D$
by *blast*

lemma *sum-equal-iff*:

fixes A :: *hf* **shows** $A+B = C+D \longleftrightarrow A=C \ \& \ B=D$
by (*auto simp*: *hf-ext sum-subset-iff*)

2.4 Ordinals

2.4.1 Basic Definitions

Definition 2.1. We say that x is transitive if every element of x is a subset of x.

definition
Transset :: *hf* $\Rightarrow$ *bool* **where**
$Transset(x) \equiv \forall y.\ y \in x \longrightarrow y \leq x$

lemma *Transset-sup*: $Transset\ x \Longrightarrow Transset\ y \Longrightarrow Transset\ (x \sqcup y)$
by (*auto simp*: *Transset-def*)

lemma *Transset-inf*: $Transset\ x \Longrightarrow Transset\ y \Longrightarrow Transset\ (x \sqcap y)$
by (*auto simp*: *Transset-def*)

lemma *Transset-hinsert*: $Transset\ x \Longrightarrow y \leq x \Longrightarrow Transset\ (x \triangleleft y)$
by (*auto simp*: *Transset-def*)

In HF, the ordinals are simply the natural numbers. But the definitions are the same as for transfinite ordinals.

definition
Ord :: *hf* $\Rightarrow$ *bool* **where**
$Ord(k) \equiv Transset(k)\ \&\ (\forall x \in k.\ Transset(x))$

2.4.2 Definition 2.2 (Successor).

definition
succ :: *hf* $\Rightarrow$ *hf* **where**
$succ(x) \equiv hinsert\ x\ x$

lemma *succ-iff* [*simp*]: $x \in succ\ y \longleftrightarrow x=y \vee x \in y$
by (*simp add*: *succ-def*)

lemma *succ-ne-self* [*simp*]: $i \neq succ\ i$
by (*metis hmem-ne succ-iff*)

lemma *succ-notin-self*: $\sim\ succ\ i <: i$
by (*metis hmem-ne succ-iff*)

lemma *succE* [*elim?*]: **assumes** $x \in succ\ y$ **obtains** $x=y \mid x \in y$
by (*metis assms succ-iff*)

lemma *hmem-succ-ne*: $succ\ x <: y \Longrightarrow x \neq y$
by (*metis hmem-not-refl succ-iff*)

lemma *hball-succ* [*simp*]: $(\forall x \in \mathit{succ}\ k.\ P\ x) \longleftrightarrow P\ k\ \&\ (\forall x \in k.\ P\ x)$
by (*auto simp*: *HBall-def*)

lemma *hbex-succ* [*simp*]: $(\exists x \in \mathit{succ}\ k.\ P\ x) \longleftrightarrow P\ k \mid (\exists x \in k.\ P\ x)$
by (*auto simp*: *HBex-def*)

lemma *One-hf-eq-succ*: $1 = \mathit{succ}\ 0$
by (*metis One-hf-def succ-def*)

lemma *zero-hmem-one* [*iff*]: $x \in 1 \longleftrightarrow x = 0$
by (*metis One-hf-eq-succ hmem-hempty succ-iff*)

lemma *hball-One* [*simp*]: $(\forall x{\in}1.\ P\ x) = P\ 0$
by (*simp add*: *One-hf-eq-succ*)

lemma *hbex-One* [*simp*]: $(\exists x{\in}1.\ P\ x) = P\ 0$
by (*simp add*: *One-hf-eq-succ*)

lemma *hpair-neq-succ* [*simp*]: $\langle x,y\rangle \neq \mathit{succ}\ k$
by (*auto simp*: *succ-def hpair-def*) (*metis hemptyE hmem-hinsert hmem-ne*)

lemma *succ-neq-hpair* [*simp*]: $\mathit{succ}\ k \neq \langle x,y\rangle$
by (*metis hpair-neq-succ*)

lemma *hpair-neq-one* [*simp*]: $\langle x,y\rangle \neq 1$
by (*metis One-hf-eq-succ hpair-neq-succ*)

lemma *one-neq-hpair* [*simp*]: $1 \neq \langle x,y\rangle$
by (*metis hpair-neq-one*)

lemma *hmem-succ-self* [*simp*]: $k \in \mathit{succ}\ k$
by (*metis succ-iff*)

lemma *hmem-succ*: $l \in k \Longrightarrow l \in \mathit{succ}\ k$
by (*metis succ-iff*)

Theorem 2.3.

lemma *Ord-0* [*iff*]: *Ord 0*
by (*simp add*: *Ord-def Transset-def*)

lemma *Ord-succ*: $\mathit{Ord}(k) \Longrightarrow \mathit{Ord}(\mathit{succ}(k))$
by (*simp add*: *Ord-def Transset-def succ-def less-eq-insert2-iff HBall-def*)

lemma *Ord-1* [*iff*]: *Ord 1*
by (*metis One-hf-def Ord-0 Ord-succ succ-def*)

lemma *OrdmemD*: $\mathit{Ord}(k) \Longrightarrow j \in k \Longrightarrow j \le k$
by (*simp add*: *Ord-def Transset-def HBall-def*)

lemma *Ord-trans*: ⟦ $i \in j$; $j \in k$; *Ord*(k) ⟧ $\Longrightarrow$ $i \in k$
  by (*blast dest*: *OrdmemD*)

lemma *hmem-0-Ord*:
  assumes k: *Ord*(k) **and** *knz*: $k \neq 0$ **shows** $0 \in k$
  by (*metis foundation* [*OF knz*] *Ord-trans hempty-iff hinter-iff k*)

lemma *Ord-in-Ord*: ⟦ *Ord*(k); $m \in k$ ⟧ $\Longrightarrow$ *Ord*(m)
  by (*auto simp*: *Ord-def Transset-def*)

2.4.3 Induction, Linearity, etc.

lemma *Ord-induct* [*consumes 1*, *case-names step*]:
  assumes k: *Ord*(k)
    and *step*: !!x.⟦ *Ord*(x); $\bigwedge y.\ y \in x \Longrightarrow P(y)$ ⟧ $\Longrightarrow P(x)$
  shows $P(k)$
proof −
  have $\forall\, m \in k.$ *Ord*(m) $\longrightarrow P(m)$
    proof (*induct k rule*: *hf-induct*)
      case *0* **thus** *?case* **by** *simp*
    next
      case (*hinsert a b*)
      thus *?case*
        by (*auto intro*: *Ord-in-Ord step*)
    qed
  thus *?thesis* **using** k
    by (*auto intro*: *Ord-in-Ord step*)
qed

Theorem 2.4 (Comparability of ordinals).

lemma *Ord-linear*: *Ord*(k) $\Longrightarrow$ *Ord*(l) $\Longrightarrow$ $k \in l \mid k = l \mid l \in k$
proof (*induct k arbitrary*: *l rule*: *Ord-induct*)
  case (*step k*)
  note *step-k* = *step*
  show *?case* **using** ‹*Ord*(l)›
    proof (*induct l rule*: *Ord-induct*)
      case (*step l*)
      thus *?case* **using** *step-k*
        by (*metis Ord-trans hf-equalityI*)
    qed
qed

The trichotomy law for ordinals

lemma *Ord-linear-lt*:
  assumes *o*: *Ord*(k) *Ord*(l)
  obtains (*lt*) $k \in l$ | (*eq*) $k = l$ | (*gt*) $l \in k$
by (*metis Ord-linear o*)

lemma *Ord-linear2*:

assumes *o*: $Ord(k)$ $Ord(l)$
obtains (lt) $k{\in}l$ | (ge) $l \le k$
by (*metis Ord-linear OrdmemD order-eq-refl o*)

lemma *Ord-linear-le*:
assumes *o*: $Ord(k)$ $Ord(l)$
obtains (le) $k \le l$ | (ge) $l \le k$
by (*metis Ord-linear2 OrdmemD o*)

lemma *hunion-less-iff* [*simp*]: $[\![Ord\ i;\ Ord\ j]\!] \Longrightarrow i \sqcup j < k \longleftrightarrow i{<}k \wedge j{<}k$
by (*metis Ord-linear-le le-iff-sup sup.order-iff sup.strict-boundedE*)

Theorem 2.5

lemma *Ord-mem-iff-lt*: $Ord(k) \Longrightarrow Ord(l) \Longrightarrow k{\in}l \longleftrightarrow k < l$
by (*metis Ord-linear OrdmemD hmem-not-refl less-hf-def less-le-not-le*)

lemma *le-succE*: $succ\ i \le succ\ j \Longrightarrow i \le j$
by (*simp add*: *less-eq-hf-def*) (*metis hmem-not-sym*)

lemma *le-succ-iff*: $Ord\ i \Longrightarrow Ord\ j \Longrightarrow succ\ i \le succ\ j \longleftrightarrow i \le j$
by (*metis Ord-linear-le Ord-succ le-succE order-antisym*)

lemma *succ-inject-iff* [*iff*]: $succ\ i = succ\ j \longleftrightarrow i = j$
by (*metis succ-def hmem-hinsert hmem-not-sym*)

lemma *mem-succ-iff* [*simp*]: $Ord\ j \Longrightarrow succ\ i \in succ\ j \longleftrightarrow i \in j$
by (*metis Ord-in-Ord Ord-mem-iff-lt Ord-succ succ-def less-eq-insert1-iff less-hf-def succ-iff*)

lemma *Ord-mem-succ-cases*:
assumes $Ord(k)$ $l \in k$
shows $succ\ l = k \vee succ\ l \in k$
by (*metis assms mem-succ-iff succ-iff*)

2.4.4 Supremum and Infimum

lemma *Ord-Union* [*intro,simp*]: $[\![\ !!i.\ i{\in}A \Longrightarrow Ord(i)\ ]\!] \Longrightarrow Ord(\bigsqcup\ A)$
by (*auto simp*: *Ord-def Transset-def*) *blast*

lemma *Ord-Inter* [*intro,simp*]: $[\![\ !!i.\ i{\in}A \Longrightarrow Ord(i)\ ]\!] \Longrightarrow Ord(\bigsqcap\ A)$
apply (*case-tac A=0*, *auto simp*: *Ord-def Transset-def*)
apply (*force simp add*: *hf-ext*)+
done

Theorem 2.7. Every set x of ordinals is ordered by the binary relation ¡. Moreover if x = 0 then x has a smallest and a largest element.

lemma *hmem-Sup-Ords*: $[\![A{\neq}0;\ !!i.\ i{\in}A \Longrightarrow Ord(i)]\!] \Longrightarrow \bigsqcup A \in A$
proof (*induction A rule*: *hf-induct*)
case *0* **thus** *?case* **by** *simp*

next
  case (*hinsert x A*)
  show *?case*
    proof (*cases A rule*: *hf-cases*)
      case *0* **thus** *?thesis* **by** *simp*
    next
      case (*hinsert y A'*)
      hence *UA*: $\bigsqcup A \in A$
        by (*metis hinsert.IH*(*2*) *hinsert.prems*(*2*) *hinsert-nonempty hmem-hinsert*)
      hence $\bigsqcup A \le x \mid x \le \bigsqcup A$
        by (*metis Ord-linear2 OrdmemD hinsert.prems*(*2*) *hmem-hinsert*)
      thus *?thesis*
      by (*metis HUnion-hinsert UA le-iff-sup less-eq-insert1-iff order-refl sup.commute*)
    qed
qed

lemma *hmem-Inf-Ords*: $[\![A \neq 0;\ !!i.\ i \in A \Longrightarrow Ord(i)]\!] \Longrightarrow \bigsqcap A \in A$
proof (*induction A rule*: *hf-induct*)
  case *0* **thus** *?case* **by** *simp*
next
  case (*hinsert x A*)
  show *?case*
    proof (*cases A rule*: *hf-cases*)
      case *0* **thus** *?thesis* **by** *auto*
    next
      case (*hinsert y A'*)
      hence *IA*: $\bigsqcap A \in A$
        by (*metis hinsert.IH*(*2*) *hinsert.prems*(*2*) *hinsert-nonempty hmem-hinsert*)
      hence $\bigsqcap A \le x \mid x \le \bigsqcap A$
        by (*metis Ord-linear2 OrdmemD hinsert.prems*(*2*) *hmem-hinsert*)
      thus *?thesis*
      by (*metis HInter-hinsert IA hmem-hempty hmem-hinsert inf-absorb2 le-iff-inf*)
    qed
qed

lemma *Ord-pred*: $[\![Ord(k);\ k \neq 0]\!] \Longrightarrow succ(\bigsqcup k) = k$
by (*metis* (*full-types*) *HUnion-iff Ord-in-Ord Ord-mem-succ-cases hmem-Sup-Ords hmem-ne succ-iff*)

lemma *Ord-cases* [*cases type*: *hf*, *case-names 0 succ*]:
  assumes *Ok*: *Ord*(*k*)
  obtains $k = 0 \mid l$ **where** *Ord l succ l* $= k$
by (*metis Ok Ord-in-Ord Ord-pred succ-iff*)

lemma *Ord-induct2* [*consumes 1*, *case-names 0 succ*, *induct type*: *hf*]:
  assumes *k*: *Ord*(*k*)
    and *P*: $P\ 0\ \bigwedge k.\ Ord\ k \Longrightarrow P\ k \Longrightarrow P\ (succ\ k)$
  shows *P k*
using *k*

proof (*induction k rule*: *Ord-induct*)
  case (*step k*) **thus** *?case*
    by (*metis Ord-cases P hmem-succ-self*)
qed

lemma *Ord-succ-iff* [*iff*]: *Ord* (*succ k*) = *Ord k*
  by (*metis Ord-in-Ord Ord-succ less-eq-insert1-iff order-refl succ-def*)

lemma [*simp*]: $succ\ k \neq 0$
  by (*metis hinsert-nonempty succ-def*)

lemma *Ord-Sup-succ-eq* [*simp*]: $Ord\ k \Longrightarrow \bigsqcup(succ\ k) = k$
  by (*metis Ord-pred Ord-succ-iff succ-inject-iff hinsert-nonempty succ-def*)

lemma *Ord-lt-succ-iff-le*: $Ord\ k \Longrightarrow Ord\ l \Longrightarrow k < succ\ l \longleftrightarrow k \leq l$
  by (*metis Ord-mem-iff-lt Ord-succ-iff less-le-not-le order-eq-iff succ-iff*)

lemma *zero-in-Ord*: $Ord\ k \Longrightarrow k{=}0 \vee 0 \in k$
  by (*induct k*) *auto*

lemma *hpair-neq-Ord*: $Ord\ k \Longrightarrow \langle x,y\rangle \neq k$
  by (*cases k*) *auto*

lemma *hpair-neq-Ord'*: **assumes** *k*: *Ord k* **shows** $k \neq \langle x,y\rangle$
  by (*metis k hpair-neq-Ord*)

lemma *Not-Ord-hpair* [*iff*]: $\sim\ Ord\ \langle x,y\rangle$
  by (*metis hpair-neq-Ord*)

lemma *is-hpair* [*simp*]: $is\text{-}hpair\ \langle x,y\rangle$
  by (*force simp add*: *is-hpair-def*)

lemma *Ord-not-hpair*: $Ord\ x \Longrightarrow \neg\ is\text{-}hpair\ x$
  by (*metis Not-Ord-hpair is-hpair-def*)

lemma *zero-in-succ* [*simp*,*intro*]: $Ord\ i \Longrightarrow 0 \in succ\ i$
  by (*metis succ-iff zero-in-Ord*)

2.4.5 Converting Between Ordinals and Natural Numbers

fun *ord-of* :: $nat \Rightarrow hf$
  where
   ord-of 0 = *0*
 | *ord-of* (*Suc k*) = *succ* (*ord-of k*)

lemma *Ord-ord-of* [*simp*]: *Ord* (*ord-of k*)
  by (*induct k*, *auto*)

lemma *ord-of-inject* [*iff*]: $ord\text{-}of\ i = ord\text{-}of\ j \longleftrightarrow i{=}j$

proof (*induct i arbitrary*: *j*)
  case *0* **show** *?case*
    by (*metis Zero-neq-Suc hempty-iff hmem-succ-self ord-of.elims*)
next
  case (*Suc i*) **show** *?case*
    by (*cases j*) (*auto simp*: *Suc*)
qed

lemma *ord-of-minus-1*: $n > 0 \Longrightarrow$ *ord-of* n = *succ* (*ord-of* $(n - 1)$)
  by (*metis Suc-diff-1 ord-of.simps*(*2*))

definition *nat-of-ord* :: *hf* ⇒ *nat*
  where *nat-of-ord x* = (*THE n. x* = *ord-of n*)

lemma *nat-of-ord-ord-of* [*simp*]: *nat-of-ord* (*ord-of n*) = *n*
  by (*auto simp*: *nat-of-ord-def*)

lemma *nat-of-ord-0* [*simp*]: *nat-of-ord 0* = *0*
  by (*metis* (*mono-tags*) *nat-of-ord-ord-of ord-of.simps*(*1*))

lemma *ord-of-nat-of-ord* [*simp*]: *Ord x* ⟹ *ord-of* (*nat-of-ord x*) = *x*
  apply (*erule Ord-induct2*, *simp*)
  apply (*metis nat-of-ord-ord-of ord-of.simps*(*2*))
  done

lemma *nat-of-ord-inject*: *Ord x* ⟹ *Ord y* ⟹ *nat-of-ord x* = *nat-of-ord y* ⟷ *x* = *y*
  by (*metis ord-of-nat-of-ord*)

lemma *nat-of-ord-succ* [*simp*]: *Ord x* ⟹ *nat-of-ord* (*succ x*) = *Suc* (*nat-of-ord x*)
  by (*metis nat-of-ord-ord-of ord-of.simps*(*2*) *ord-of-nat-of-ord*)

2.5 Sequences and Ordinal Recursion

Definition 3.2 (Sequence).

definition *Seq* :: *hf* ⇒ *hf* ⇒ *bool*
  where *Seq s k* ⟷ *hrelation s* & *hfunction s* & *k* ≤ *hdomain s*

lemma *Seq-0* [*iff*]: *Seq 0 0*
  by (*auto simp*: *Seq-def hrelation-def hfunction-def*)

lemma *Seq-succ-D*: *Seq s* (*succ k*) ⟹ *Seq s k*
  by (*simp add*: *Seq-def succ-def*)

lemma *Seq-Ord-D*: *Seq s k* ⟹ *l* ∈ *k* ⟹ *Ord k* ⟹ *Seq s l*
  by (*auto simp*: *Seq-def intro*: *Ord-trans*)

lemma *Seq-restr*: $Seq\ s\ (succ\ k) \Longrightarrow Seq\ (hrestrict\ s\ k)\ k$
  by (*simp add*: *Seq-def hfunction-restr succ-def*)

lemma *Seq-Ord-restr*: $[\![Seq\ s\ k;\ l \in k;\ Ord\ k]\!] \Longrightarrow Seq\ (hrestrict\ s\ l)\ l$
  by (*auto simp*: *Seq-def hfunction-restr intro*: *Ord-trans*)

lemma *Seq-ins*: $[\![Seq\ s\ k;\ {\sim}\ k <: hdomain\ s]\!] \Longrightarrow Seq\ (s \triangleleft \langle k,\ y\rangle)\ (succ\ k)$
  by (*auto simp*: *Seq-def hrelation-def succ-def hfunction-def hdomainI*)

definition $insf :: hf \Rightarrow hf \Rightarrow hf \Rightarrow hf$
  where $insf\ s\ k\ y \equiv nonrestrict\ s\ \{\!|k|\!\} \triangleleft \langle k,\ y\rangle$

lemma *hfunction-insf*: $hfunction\ s \Longrightarrow hfunction\ (insf\ s\ k\ y)$
  by (*auto simp*: *insf-def hfunction-def nonrestrict-def hmem-not-refl*)

lemma *Seq-insf*: $Seq\ s\ k \Longrightarrow Seq\ (insf\ s\ k\ y)\ (succ\ k)$
  apply (*auto simp*: *Seq-def hrelation-def insf-def hfunction-def nonrestrict-def*)
  apply (*force simp add*: *hdomain-def*)
  done

lemma *Seq-succ-iff*: $Seq\ s\ (succ\ k) \longleftrightarrow Seq\ s\ k \wedge (\exists\, y.\ \langle k,\ y\rangle <: s)$
  apply (*auto simp*: *Seq-def hdomain-def*)
  apply (*metis hfst-conv*, *blast*)
  done

lemma *nonrestrictD*: $a \in nonrestrict\ s\ X \Longrightarrow a \in s$
  by (*auto simp*: *nonrestrict-def*)

lemma *hpair-in-nonrestrict-iff* [*simp*]: $\langle a,b\rangle \in nonrestrict\ s\ X \longleftrightarrow \langle a,b\rangle \in s \wedge \neg\ a \in X$
  by (*auto simp*: *nonrestrict-def*)

lemma *app-nonrestrict-Seq*: $Seq\ s\ k \Longrightarrow {\sim}\ z <: X \Longrightarrow app\ (nonrestrict\ s\ X)\ z = app\ s\ z$
  by (*auto simp*: *Seq-def nonrestrict-def app-def*)

lemma *app-insf-Seq*: $Seq\ s\ k \Longrightarrow app\ (insf\ s\ k\ y)\ k = y$
  by (*metis Seq-def hfunction-insf app-equality hmem-hinsert insf-def*)

lemma *app-insf2-Seq*: $Seq\ s\ k \Longrightarrow k' \neq k \Longrightarrow app\ (insf\ s\ k\ y)\ k' = app\ s\ k'$
  by (*simp add*: *app-nonrestrict-Seq insf-def app-ins2*)

lemma *app-insf-Seq-if*: $Seq\ s\ k \Longrightarrow app\ (insf\ s\ k\ y)\ k' = (if\ k' = k\ then\ y\ else\ app\ s\ k')$
  by (*metis app-insf2-Seq app-insf-Seq*)

lemma *Seq-imp-eq-app*: $[\![Seq\ s\ d;\ \langle x,y\rangle \in s]\!] \Longrightarrow app\ s\ x = y$
  by (*metis Seq-def app-equality*)

```
lemma Seq-iff-app: ⟦Seq s d; x ∈ d⟧ ⟹ ⟨x,y⟩ ∈ s ⟷ app s x = y
  by (auto simp: Seq-def hdomain-def app-equality)

lemma Exists-iff-app: Seq s d ⟹ x ∈ d ⟹ (∃ y. ⟨x, y⟩ ∈ s & P y) = P (app s
x)
  by (metis Seq-iff-app)

lemma Ord-trans2: ⟦i2 ∈ i; i ∈ j;  j ∈ k;  Ord k⟧ ⟹ i2∈k
  by (metis Ord-trans)

definition ord-rec-Seq :: hf ⇒ (hf ⇒ hf) ⇒ hf ⇒ hf ⇒ hf ⇒ bool
  where
   ord-rec-Seq T G s k y ⟷
        (Seq s k & y = G (app s (⨆k)) & app s 0 = T &
                   (∀ n. succ n ∈ k ⟶ app s (succ n) = G (app s n)))

lemma Seq-succ-insf:
  assumes s: Seq s (succ k)  shows ∃ y. s = insf s k y
proof −
  obtain y where y: ⟨k, y⟩ <: s by (metis Seq-succ-iff s)
  hence yuniq: ∀ y'. ⟨k, y'⟩ <: s ⟶ y' = y using s
    by (simp add: Seq-def hfunction-def)
  { fix z
    assume z: z <: s
    then obtain u v where uv: z = ⟨u, v⟩ using s
      by (metis Seq-def hrelation-def is-hpair-def)
    hence z <: insf s k y
      by (metis hemptyE hmem-hinsert hpair-in-nonrestrict-iff insf-def yuniq z)
  }
  note left2right = this
  show ?thesis
    proof
      show s = insf s k y
         by (rule hf-equalityI) (metis hmem-hinsert insf-def left2right nonrestrictD
y)
    qed
qed

lemma ord-rec-Seq-succ-iff:
  assumes k: Ord k and knz: k ≠ 0
  shows ord-rec-Seq T G s (succ k) z ⟷ (∃ s' y. ord-rec-Seq T G s' k y & z =
G y & s = insf s' k y)
proof
  assume os: ord-rec-Seq T G s (succ k) z
  show ∃s' y. ord-rec-Seq T G s' k y ∧ z = G y ∧ s = insf s' k y
    apply (rule-tac x=s in exI)  using os k knz
    apply (auto simp: Seq-insf ord-rec-Seq-def app-insf-Seq app-insf2-Seq
                  hmem-succ-ne hmem-ne hmem-Sup-ne Seq-succ-iff hmem-0-Ord)
    apply (metis Ord-pred)
```

apply (*metis Ord-pred Seq-succ-iff Seq-succ-insf app-insf-Seq*)
done
next
assume *ok*: $\exists s'\ y.\ ord\text{-}rec\text{-}Seq\ T\ G\ s'\ k\ y \wedge z = G\ y \wedge s = insf\ s'\ k\ y$
thus *ord-rec-Seq T G s* (*succ k*) *z* **using** *ok k knz*
by (*auto simp*: *ord-rec-Seq-def app-insf-Seq-if hmem-ne hmem-succ-ne Seq-insf*)
qed

lemma *ord-rec-Seq-functional*:
$Ord\ k \Longrightarrow k \neq 0 \Longrightarrow ord\text{-}rec\text{-}Seq\ T\ G\ s\ k\ y \Longrightarrow ord\text{-}rec\text{-}Seq\ T\ G\ s'\ k\ y' \Longrightarrow y' = y$
proof (*induct k arbitrary*: *y y' s s' rule*: *Ord-induct2*)
case *0* **thus** *?case*
by (*simp add*: *ord-rec-Seq-def*)
next
case (*succ k*) **show** *?case*
proof (*cases k=0*)
case *True* **thus** *?thesis* **using** *succ*
by (*auto simp*: *ord-rec-Seq-def*)
next
case *False*
thus *?thesis* **using** *succ*
by (*auto simp*: *ord-rec-Seq-succ-iff*)
qed
qed

definition *ord-recp* :: $hf \Rightarrow (hf \Rightarrow hf) \Rightarrow (hf \Rightarrow hf) \Rightarrow hf \Rightarrow hf \Rightarrow bool$
where
ord-recp T G H x y =
(*if x=0 then y = T*
else
if Ord(*x*) *then* $\exists$ *s. ord-rec-Seq T G s x y*
else y = H x)

lemma *ord-recp-functional*: $ord\text{-}recp\ T\ G\ H\ x\ y \Longrightarrow ord\text{-}recp\ T\ G\ H\ x\ y' \Longrightarrow y' = y$
by (*auto simp*: *ord-recp-def ord-rec-Seq-functional split*: *split-if-asm*)

lemma *ord-recp-succ-iff*:
assumes *k*: *Ord k* **shows** *ord-recp T G H* (*succ k*) $z \longleftrightarrow (\exists y.\ z = G\ y\ \&\ ord\text{-}recp\ T\ G\ H\ k\ y)$
proof (*cases k=0*)
case *True* **thus** *?thesis*
by (*simp add*: *ord-recp-def ord-rec-Seq-def*) (*metis Seq-0 Seq-insf app-insf-Seq*)
next
case *False*
thus *?thesis* **using** *k*
by (*auto simp*: *ord-recp-def ord-rec-Seq-succ-iff*)
qed

```
definition ord-rec :: hf ⇒ (hf ⇒ hf) ⇒ (hf ⇒ hf) ⇒ hf ⇒ hf
  where
   ord-rec T G H x = (THE y. ord-recp T G H x y)

lemma ord-rec-0 [simp]: ord-rec T G H 0 = T
  by (simp add: ord-recp-def ord-rec-def)

lemma ord-recp-total: ∃ y. ord-recp T G H x y
proof (cases Ord x)
  case True thus ?thesis
  proof (induct x rule: Ord-induct2)
    case 0 thus ?case
      by (simp add: ord-recp-def)
  next
    case (succ x) thus ?case
      by (metis ord-recp-succ-iff)
  qed
next
  case False thus ?thesis
    by (auto simp: ord-recp-def)
qed

lemma ord-rec-succ [simp]:
  assumes k: Ord k shows ord-rec T G H (succ k) = G (ord-rec T G H k)
proof −
  from ord-recp-total [of T G H k]
  obtain y where ord-recp T G H k y by auto
  thus ?thesis using k
    apply (simp add: ord-rec-def ord-recp-succ-iff)
    apply (rule theI2)
    apply (auto dest: ord-recp-functional)
    done
qed

lemma ord-rec-non [simp]: ~ Ord x ⟹ ord-rec T G H x = H x
  by (metis Ord-0 ord-rec-def ord-recp-def the-equality)

end
```

Chapter 3

V-Sets, Epsilon Closure, Ranks

theory *Rank* **imports** *Ordinal*
begin

3.1 V-sets

Definition 4.1

definition *Vset* :: *hf* ⇒ *hf*
  where *Vset x* = *ord-rec 0 HPow* (*λz. 0*) *x*

lemma *Vset-0* [*simp*]: *Vset 0* = *0*
  by (*simp add*: *Vset-def*)

lemma *Vset-succ* [*simp*]: *Ord k* ⟹ *Vset* (*succ k*) = *HPow* (*Vset k*)
  by (*simp add*: *Vset-def*)

lemma *Vset-non* [*simp*]: ~ *Ord x* ⟹ *Vset x* = *0*
  by (*simp add*: *Vset-def*)

Theorem 4.2(a)

lemma *Vset-mono-strict*:
  assumes *Ord m n* <: *m* **shows** *Vset n* < *Vset m*
proof −
  have *n*: *Ord n*
    by (*metis Ord-in-Ord assms*)
  hence *Ord m* ⟹ *n* <: *m* ⟹ *Vset n* < *Vset m*
  proof (*induct n arbitrary*: *m rule*: *Ord-induct2*)
    case *0* **thus** *?case*
      by (*metis HPow-iff Ord-cases Vset-0 Vset-succ hemptyE le-imp-less-or-eq zero-le*)
  next
    case (*succ n*)

then show *?case* **using** ‹*Ord m*›
by (*metis Ord-cases hemptyE HPow-mono-strict-iff Vset-succ mem-succ-iff*)
qed
thus *?thesis* **using** *assms* **.**
qed

lemma *Vset-mono*: $[\![Ord\ m;\ n \le m]\!] \Longrightarrow Vset\ n \le Vset\ m$
by (*metis Ord-linear2 Vset-mono-strict Vset-non assms order.order-iff-strict order-class.order.antisym zero-le*)

Theorem 4.2(b)

lemma *Vset-Transset*: $Ord\ m \Longrightarrow Transset\ (Vset\ m)$
by (*induct rule*: *Ord-induct2*) (*auto simp*: *Transset-def*)

lemma *Ord-sup* [*simp*]: $Ord\ k \Longrightarrow Ord\ l \Longrightarrow Ord\ (k \sqcup l)$
by (*metis Ord-linear-le le-iff-sup sup-absorb1*)

lemma *Ord-inf* [*simp*]: $Ord\ k \Longrightarrow Ord\ l \Longrightarrow Ord\ (k \sqcap l)$
by (*metis Ord-linear-le inf-absorb2 le-iff-inf*)

Theorem 4.3

lemma *Vset-universal*: $\exists n.\ Ord\ n\ \&\ x \in Vset\ n$
proof (*induct x rule*: *hf-induct*)
case *0* **thus** *?case*
by (*metis HPow-iff Ord-0 Ord-succ Vset-succ zero-le*)
next
case (*hinsert a b*)
then obtain *na nb* **where** *nab*: $Ord\ na\ a \in Vset\ na\ Ord\ nb\ b \in Vset\ nb$
by *blast*
hence $b \le Vset\ nb$ **using** *Vset-Transset* [*of nb*]
by (*auto simp*: *Transset-def*)
also have $... \le Vset\ (na \sqcup nb)$ **using** *nab*
by (*metis Ord-sup Vset-mono sup-ge2*)
finally have $b \triangleleft a \in Vset\ (succ\ (na \sqcup nb))$ **using** *nab*
by *simp* (*metis Ord-sup Vset-mono sup-ge1 rev-hsubsetD*)
thus *?case* **using** *nab*
by (*metis Ord-succ Ord-sup*)
qed

3.2 Least Ordinal Operator

Definition 4.4. For every x, let rank(x) be the least ordinal n such that...

lemma *Ord-minimal*:
$Ord\ k \Longrightarrow P\ k \Longrightarrow \exists n.\ Ord\ n\ \&\ P\ n\ \&\ (\forall m.\ Ord\ m\ \&\ P\ m \longrightarrow n \le m)$
by (*induct k rule*: *Ord-induct*) (*metis Ord-linear2*)

lemma *OrdLeastI*: $Ord\ k \Longrightarrow P\ k \Longrightarrow P(LEAST\ n.\ Ord\ n\ \&\ P\ n)$
by (*metis* (*lifting*, *no-types*) *Least-equality Ord-minimal*)

lemma *OrdLeast-le*: *Ord k* $\Longrightarrow$ *P k* $\Longrightarrow$ (*LEAST n*. *Ord n* & *P n*) $\leq$ *k*
by (*metis* (*lifting*, *no-types*) *Least-equality Ord-minimal*)

lemma *OrdLeast-Ord*:
  assumes *Ord k P k***shows** *Ord*(*LEAST n*. *Ord n* & *P n*)
proof −
  obtain *n* **where** *Ord n P n* $\forall$ *m*. *Ord m* & *P m* $\longrightarrow$ *n* $\leq$ *m*
    by (*metis Ord-minimal assms*)
  thus *?thesis*
    by (*metis* (*lifting*) *Least-equality*)
qed

3.3 Rank Function

definition *rank* :: *hf* $\Rightarrow$ *hf*
  where *rank x* = (*LEAST n*. *Ord n* & *x* $\in$ *Vset* (*succ n*))

lemma [*simp*]: *rank 0* = *0*
  by (*simp add*: *rank-def*) (*metis* (*lifting*) *HPow-iff Least-equality Ord-0 Vset-succ zero-le*)

lemma *in-Vset-rank*: *a* $\in$ *Vset*(*succ*(*rank a*))
proof −
  from *Vset-universal* [*of a*]
  obtain *na* **where** *na*: *Ord na a* $\in$ *Vset* (*succ na*)
    by (*metis Ord-Union Ord-in-Ord Ord-pred Vset-0 hempty-iff*)
  thus *?thesis*
    by (*unfold rank-def*) (*rule OrdLeastI*)
qed

lemma *Ord-rank* [*simp*]: *Ord* (*rank a*)
  by (*metis Ord-succ-iff Vset-non hemptyE in-Vset-rank*)

lemma *le-Vset-rank*: *a* $\leq$ *Vset*(*rank a*)
  by (*metis HPow-iff Ord-succ-iff Vset-non Vset-succ hemptyE in-Vset-rank*)

lemma *VsetI*: *succ*(*rank a*) $\leq$ *k* $\Longrightarrow$ *Ord k* $\Longrightarrow$ *a* $\in$ *Vset k*
  by (*metis Vset-mono hsubsetCE in-Vset-rank*)

lemma *Vset-succ-rank-le*: *Ord k* $\Longrightarrow$ *a* $\in$ *Vset* (*succ k*) $\Longrightarrow$ *rank a* $\leq$ *k*
  by (*unfold rank-def*) (*rule OrdLeast-le*)

lemma *Vset-rank-lt*: **assumes** *a*: *a* $\in$ *Vset k* **shows** *rank a* < *k*
proof −
  { **assume** *k*: *Ord k*
    hence *?thesis*
    proof (*cases k rule*: *Ord-cases*)
      case *0* **thus** *?thesis* **using** *a*

by *simp*
next
case (*succ l*) **thus** *?thesis* **using** *a*
by (*metis Ord-lt-succ-iff-le Ord-succ-iff Vset-non Vset-succ-rank-le hemptyE in-Vset-rank*)
qed
}
thus *?thesis* **using** *a*
by (*metis Vset-non hemptyE*)
qed

Theorem 4.5

theorem *rank-lt*: $a \in b \Longrightarrow rank(a) < rank(b)$
by (*metis Vset-rank-lt hsubsetD le-Vset-rank*)

lemma *rank-mono*: $x \leq y \Longrightarrow rank\ x \leq rank\ y$
by (*metis HPow-iff Ord-rank Vset-succ Vset-succ-rank-le dual-order.trans le-Vset-rank*)

lemma *rank-sup* [*simp*]: $rank\ (a \sqcup b) = rank\ a \sqcup rank\ b$
proof (*rule antisym*)
have *o*: *Ord* $(rank\ a \sqcup rank\ b)$
by *simp*
thus $rank\ (a \sqcup b) \leq rank\ a \sqcup rank\ b$
apply (*rule Vset-succ-rank-le*, *simp*)
apply (*metis le-Vset-rank order-trans Vset-mono sup-ge1 sup-ge2 o*)
done
next
show $rank\ a \sqcup rank\ b \leq rank\ (a \sqcup b)$
by (*metis le-supI le-supI1 le-supI2 order-eq-refl rank-mono*)
qed

lemma *rank-singleton* [*simp*]: $rank\ \{\!|a|\!\} = succ(rank\ a)$
proof −
have *oba*: *Ord* $(succ\ (rank\ a))$
by *simp*
show *?thesis*
proof (*rule antisym*)
show $rank\ \{\!|a|\!\} \leq succ\ (rank\ a)$
by (*metis Vset-succ-rank-le HPow-iff Vset-succ in-Vset-rank less-eq-insert1-iff oba zero-le*)
next
show $succ\ (rank\ a) \leq rank\{\!|a|\!\}$
by (*metis Ord-linear-le Ord-lt-succ-iff-le rank-lt Ord-rank hmem-hinsert less-le-not-le oba*)
qed
qed

lemma *rank-hinsert* [*simp*]: $rank\ (b \triangleleft a) = rank\ b \sqcup succ(rank\ a)$
by (*metis hinsert-eq-sup rank-singleton rank-sup*)

Definition 4.6. The transitive closure of x is the minimal transitive set y such that $x \leq y$.

3.4 Epsilon Closure

definition
  eclose :: *hf* ⇒ *hf* **where**
    $eclose\ X = \bigsqcap\ \{\!|Y \in HPow(Vset\ (rank\ X)).\ Transset\ Y\ \&\ X{\le}Y|\!\}$

lemma *eclose-facts*:
  shows *Transset-eclose*: *Transset* (*eclose X*)
  and *le-eclose*: $X \le eclose\ X$
proof −
  have *nz*: $\{\!|Y \in HPow(Vset\ (rank\ X)).\ Transset\ Y\ \&\ X{\le}Y|\!\} \neq 0$
    by (*simp add*: *eclose-def hempty-iff*) (*metis Ord-rank Vset-Transset le-Vset-rank order-refl*)
  show *Transset* (*eclose X*) $X \le eclose\ X$ **using** *HInter-iff* [*OF nz*]
    by (*auto simp*: *eclose-def Transset-def*)
qed

lemma *eclose-minimal*:
  assumes *Y*: *Transset Y* $X{\le}Y$ **shows** $eclose\ X \le Y$
proof −
  have $\{\!|Y \in HPow(Vset\ (rank\ X)).\ Transset\ Y\ \&\ X{\le}Y|\!\} \neq 0$
    by (*simp add*: *eclose-def hempty-iff*) (*metis Ord-rank Vset-Transset le-Vset-rank order-refl*)
  moreover have *Transset* $(Y \sqcap Vset\ (rank\ X))$
    by (*metis Ord-rank Transset-inf Vset-Transset Y(1)*)
  moreover have $X \le Y \sqcap Vset\ (rank\ X)$
    by (*metis Y(2) le-Vset-rank le-inf-iff*)
  ultimately show $eclose\ X \le Y$
    apply (*auto simp*: *eclose-def*)
    apply (*metis hinter-iff le-inf-iff order-refl*)
    done
qed

lemma *eclose-0* [*simp*]: *eclose 0 = 0*
  by (*metis Ord-0 Vset-0 Vset-Transset eclose-minimal less-eq-hempty*)

lemma *eclose-sup* [*simp*]: $eclose\ (a \sqcup b) = eclose\ a \sqcup eclose\ b$
proof (*rule order-antisym*)
  show $eclose\ (a \sqcup b) \le eclose\ a \sqcup eclose\ b$
    by (*metis Transset-eclose Transset-sup eclose-minimal le-eclose sup-mono*)
next
  show $eclose\ a \sqcup eclose\ b \le eclose\ (a \sqcup b)$
    by (*metis Transset-eclose eclose-minimal le-eclose le-sup-iff*)
qed

lemma *eclose-singleton* [*simp*]: *eclose* $\{\!|a|\!\}$ = (*eclose* *a*) $\triangleleft$ *a*
proof (*rule order-antisym*)
  show *eclose* $\{\!|a|\!\}$ $\leq$ *eclose a* $\triangleleft$ *a*
    by (*metis eclose-minimal Transset-eclose Transset-hinsert*
        le-eclose less-eq-insert1-iff order-refl zero-le)
next
  show *eclose a* $\triangleleft$ *a* $\leq$ *eclose* $\{\!|a|\!\}$
   by (*metis Transset-def Transset-eclose eclose-minimal le-eclose less-eq-insert1-iff*)
qed

lemma *eclose-hinsert* [*simp*]: *eclose* (*b* $\triangleleft$ *a*) = *eclose b* $\sqcup$ (*eclose a* $\triangleleft$ *a*)
  by (*metis eclose-singleton eclose-sup hinsert-eq-sup*)

lemma *eclose-succ* [*simp*]: *eclose* (*succ a*) = *eclose a* $\triangleleft$ *a*
  by (*auto simp*: *succ-def*)

lemma *fst-in-eclose* [*simp*]: $x \in$ *eclose* $\langle x, y\rangle$
  by (*metis eclose-hinsert hmem-hinsert hpair-def hunion-iff*)

lemma *snd-in-eclose* [*simp*]: $y \in$ *eclose* $\langle x, y\rangle$
  by (*metis eclose-hinsert hmem-hinsert hpair-def hunion-iff*)

Theorem 4.7. rank(x) = rank(cl(x)).

lemma *rank-eclose* [*simp*]: *rank* (*eclose x*) = *rank x*
proof (*induct x rule*: *hf-induct*)
  case *0* **thus** *?case* **by** *simp*
next
  case (*hinsert a b*) **thus** *?case*
    by *simp* (*metis hinsert-eq-sup succ-def sup.left-idem*)
qed

3.5 Epsilon-Recursion

Theorem 4.9. Definition of a function by recursion on rank.

lemma *hmem-induct* [*case-names step*]:
  assumes *ih*: $\bigwedge x.\ (\bigwedge y.\ y \in x \Longrightarrow P\ y) \Longrightarrow P\ x$ **shows** $P\ x$
proof −
  have $\bigwedge y.\ y \in x \Longrightarrow P\ y$
  proof (*induct x rule*: *hf-induct*)
    case *0* **thus** *?case* **by** *simp*
  next
    case (*hinsert a b*) **thus** *?case*
      by (*metis assms hmem-hinsert*)
  qed
  thus *?thesis* **by** (*metis ih*)
qed

definition

hmem-rel :: (*hf* ∗ *hf*) *set* **where**
hmem-rel = *trancl* {(*x*,*y*). *x* <: *y*}

lemma *wf-hmem-rel*: *wf hmem-rel*
proof −
have *wf* {(*x*,*y*). *x* <: *y*}
by (*metis* (*full-types*) *hmem-induct wfPUNIVI wfP-def*)
thus *?thesis*
by (*metis hmem-rel-def wf-trancl*)
qed

lemma *hmem-eclose-le*: *y* ∈ *x* ⟹ *eclose y* ≤ *eclose x*
by (*metis Transset-def Transset-eclose eclose-minimal hsubsetD le-eclose*)

lemma *hmem-rel-iff-hmem-eclose*: (*x*,*y*) ∈ *hmem-rel* ⟷ *x* <: *eclose y*
proof (*unfold hmem-rel-def*, *rule iffI*)
assume (*x*, *y*) ∈ *trancl* {(*x*, *y*). *x* ∈ *y*}
thus *x* ∈ *eclose y*
proof (*induct rule*: *trancl-induct*)
case (*base y*) **thus** *?case*
by (*metis hsubsetCE le-eclose mem-Collect-eq split-conv*)
next
case (*step y z*) **thus** *?case*
by (*metis hmem-eclose-le hsubsetD mem-Collect-eq split-conv*)
qed
next
have *Transset* ⦃*x* ∈ *eclose y*. (*x*, *y*) ∈ *hmem-rel*⦄ **using** *Transset-eclose*
by (*auto simp*: *Transset-def hmem-rel-def intro*: *trancl-trans*)
hence *eclose y* ≤ ⦃*x* ∈ *eclose y*. (*x*, *y*) ∈ *hmem-rel*⦄
by (*rule eclose-minimal*) (*auto simp*: *le-HCollect-iff le-eclose hmem-rel-def*)
moreover assume *x* ∈ *eclose y*
ultimately show (*x*, *y*) ∈ *trancl* {(*x*, *y*). *x* ∈ *y*}
by (*metis HCollect-iff hmem-rel-def hsubsetD*)
qed

definition *hmemrec* :: ((*hf* ⇒ *'a*) ⇒ *hf* ⇒ *'a*) ⇒ *hf* ⇒ *'a* **where**
hmemrec G ≡ *wfrec hmem-rel G*

definition *ecut* :: (*hf* ⇒ *'a*) ⇒ *hf* ⇒ *hf* ⇒ *'a* **where**
ecut f x ≡ (λ*y*. *if y* ∈ *eclose x then f y else undefined*)

lemma *hmemrec*: *hmemrec G a* = *G* (*ecut* (*hmemrec G*) *a*) *a*
by (*simp add*: *cut-def ecut-def hmem-rel-iff-hmem-eclose def-wfrec* [*OF hmemrec-def wf-hmem-rel*])

This form avoids giant explosions in proofs.

lemma *def-hmemrec*: *f* ≡ *hmemrec G* ⟹ *f a* = *G* (*ecut* (*hmemrec G*) *a*) *a*
by (*metis hmemrec*)

lemma *ecut-apply*: $y \in$ *eclose* $x \Longrightarrow$ *ecut* $f\ x\ y = f\ y$
by (*metis ecut-def*)

lemma *RepFun-ecut*: $y \le z \Longrightarrow$ *RepFun* y (*ecut* $f\ z$) = *RepFun* $y\ f$
apply (*auto simp*: *hf-ext*)
apply (*metis ecut-def hsubsetD le-eclose*)
apply (*metis ecut-apply le-eclose hsubsetD*)
done

Now, a stronger induction rule, for the transitive closure of membership

lemma *hmem-rel-induct* [*case-names step*]:
assumes *ih*: $\bigwedge x.\ (\bigwedge y.\ (y,x) \in$ *hmem-rel* $\Longrightarrow P\ y) \Longrightarrow P\ x$ **shows** $P\ x$
proof −
have $\bigwedge y.\ (y,x) \in$ *hmem-rel* $\Longrightarrow P\ y$
proof (*induct x rule*: *hf-induct*)
case *0* **thus** *?case*
by (*metis eclose-0 hmem-hempty hmem-rel-iff-hmem-eclose*)
next
case (*hinsert a b*)
thus *?case*
by (*metis assms eclose-hinsert hmem-hinsert hmem-rel-iff-hmem-eclose hunion-iff*)
qed
thus *?thesis* **by** (*metis assms*)
qed

lemma *rank-HUnion-less*: $x \neq 0 \Longrightarrow$ *rank* $(\bigsqcup x) <$ *rank* x
apply (*induct x rule*: *hf-induct*, *auto*)
apply (*metis hmem-hinsert rank-hinsert rank-lt*)
apply (*metis HUnion-hempty Ord-lt-succ-iff-le Ord-rank hunion-hempty-right less-supI1 less-supI2 rank-sup sup.cobounded2*)
done

corollary *Sup-ne*: $x \neq 0 \Longrightarrow \bigsqcup x \neq x$
by (*metis less-irrefl rank-HUnion-less*)

end

Chapter 4

An Application: Finite Automata

theory *Finite-Automata* **imports** *Ordinal*
begin

The point of this example is that the HF sets are closed under disjoint sums and Cartesian products, allowing the theory of finite state machines to be developed without issues of polymorphism or any tricky encodings of states.

record *'a fsm = states :: hf*
init :: hf
final :: hf
nxt :: hf ⇒ 'a ⇒ hf ⇒ bool

inductive *reaches* :: [*'a fsm*, *hf*, *'a list*, *hf*] ⇒ *bool*
where
Nil: *st* <: *states fsm* ⟹ *reaches fsm st* [] *st*
| *Cons*: ⟦*nxt fsm st x st''*; *reaches fsm st'' xs st'*; *st* <: *states fsm*⟧ ⟹ *reaches fsm st* (*x*#*xs*) *st'*

declare *reaches.intros* [*intro*]
inductive-simps *reaches-Nil* [*simp*]: *reaches fsm st* [] *st'*
inductive-simps *reaches-Cons* [*simp*]: *reaches fsm st* (*x*#*xs*) *st'*

lemma *reaches-imp-states*: *reaches fsm st xs st'* ⟹ *st* <: *states fsm* ∧ *st'* <: *states fsm*
by (*induct xs arbitrary*: *st st'*, *auto*)

lemma *reaches-append-iff*:
reaches fsm st (*xs*@*ys*) *st'* ⟷ (∃ *st''*. *reaches fsm st xs st''* ∧ *reaches fsm st'' ys st'*)
by (*induct xs arbitrary*: *ys st st'*) (*auto simp*: *reaches-imp-states*)

definition *accepts* :: *'a fsm* ⇒ *'a list* ⇒ *bool* **where**

$accepts\ fsm\ xs \equiv \exists\, st\ st'.\ reaches\ fsm\ st\ xs\ st' \wedge st <: init\ fsm \wedge st' <: final\ fsm$

definition *regular* :: *'a list set* ⇒ *bool* **where**
$regular\ S \equiv \exists fsm.\ S = \{xs.\ accepts\ fsm\ xs\}$

definition *Null* **where**
$Null = (\!|states = 0,\ init = 0,\ final = 0,\ nxt = \lambda st\ x\ st'.\ False|\!)$

theorem *regular-empty*: *regular* {}
by (*auto simp*: *regular-def accepts-def*) (*metis hempty-iff simps*(*2*))

abbreviation *NullStr* **where**
$NullStr \equiv (\!|states = 1,\ init = 1,\ final = 1,\ nxt = \lambda st\ x\ st'.\ False|\!)$

theorem *regular-emptystr*: *regular* {[]}
apply (*auto simp*: *regular-def accepts-def*)
apply (*rule exI* [**where** *x* = *NullStr*], *auto*)
apply (*case-tac x*, *auto*)
done

abbreviation *SingStr* **where**
$SingStr\ a \equiv (\!|states = \{|0,\ 1|\},\ init = \{|0|\},\ final = \{|1|\},\ nxt = \lambda st\ x\ st'.\ st{=}0 \wedge x{=}a \wedge st'{=}1|\!)$

theorem *regular-singstr*: *regular* {[*a*]}
apply (*auto simp*: *regular-def accepts-def*)
apply (*rule exI* [**where** *x* = *SingStr a*], *auto*)
apply (*case-tac x*, *auto*)
apply (*case-tac list*, *auto*)
done

definition *Reverse* **where**
$Reverse\ fsm = (\!|states = states\ fsm,\ init = final\ fsm,\ final = init\ fsm,\ nxt = \lambda st\ x\ st'.\ nxt\ fsm\ st'\ x\ st|\!)$

lemma *Reverse-Reverse-ident* [*simp*]: *Reverse* (*Reverse fsm*) = *fsm*
by (*simp add*: *Reverse-def*)

lemma *reaches-Reverse-iff* [*simp*]:
$reaches\ (Reverse\ fsm)\ st\ (rev\ xs)\ st' \longleftrightarrow reaches\ fsm\ st'\ xs\ st$
by (*induct xs arbitrary*: *st st'*) (*auto simp add*: *Reverse-def reaches-append-iff reaches-imp-states*)

lemma *reaches-Reverse-iff2* [*simp*]:
$reaches\ (Reverse\ fsm)\ st'\ xs\ st \longleftrightarrow reaches\ fsm\ st\ (rev\ xs)\ st'$
by (*metis reaches-Reverse-iff rev-rev-ident*)

lemma [*simp*]: *init* (*Reverse fsm*) = *final fsm*
by (*simp add*: *Reverse-def*)

```
lemma [simp]: final (Reverse fsm) = init fsm
  by (simp add: Reverse-def)

theorem regular-rev: regular S ⟹ regular (rev ` S)
  apply (auto simp: regular-def accepts-def)
  apply (rule-tac x=Reverse fsm in exI, force+)
  done

definition Times where
  Times fsm1 fsm2 = (|states = states fsm1 * states fsm2,
                      init = init fsm1 * init fsm2,
                      final = final fsm1 * final fsm2,
                         nxt = λst x st'. (∃ st1 st2 st1' st2'. st = ⟨st1,st2⟩ ∧ st' =
⟨st1',st2'⟩ ∧
                                 nxt fsm1 st1 x st1' ∧ nxt fsm2 st2 x st2')|)

lemma states-Times [simp]: states (Times fsm1 fsm2) = states fsm1 * states fsm2

  by (simp add: Times-def)

lemma init-Times [simp]: init (Times fsm1 fsm2) = init fsm1 * init fsm2
  by (simp add: Times-def)

lemma final-Times [simp]: final (Times fsm1 fsm2) = final fsm1 * final fsm2

  by (simp add: Times-def)

lemma nxt-Times: nxt (Times fsm1 fsm2) ⟨st1,st2⟩ x st' ⟷
    (∃ st1' st2'. st' = ⟨st1',st2'⟩ ∧ nxt fsm1 st1 x st1' ∧ nxt fsm2 st2 x st2')
  by (simp add: Times-def)

lemma reaches-Times-iff [simp]:
     reaches (Times fsm1 fsm2) ⟨st1,st2⟩ xs ⟨st1',st2'⟩ ⟷
      reaches fsm1 st1 xs st1' ∧ reaches fsm2 st2 xs st2'
apply (induct xs arbitrary: st1 st2 st1' st2', force)
apply (force simp add: nxt-Times Times-def reaches.Cons)
done

lemma accepts-Times-iff [simp]:
     accepts (Times fsm1 fsm2) xs ⟷
      accepts fsm1 xs ∧ accepts fsm2 xs
  by (force simp add: accepts-def)

theorem regular-Int:
  assumes S: regular S and T: regular T shows regular (S ∩ T)
proof -
  obtain fsmS fsmT where S = {xs. accepts fsmS xs} T = {xs. accepts fsmT xs}
using S T
```

by (*auto simp*: *regular-def*)
hence $S \cap T = \{xs.\ accepts\ (Times\ fsmS\ fsmT)\ xs\}$
by (*auto simp*: *accepts-Times-iff* [*of fsmS fsmT*])
thus *?thesis*
by (*metis regular-def*)
qed

definition *Plus* **where**
Plus fsm1 fsm2 = (|*states* = *states fsm1* + *states fsm2*,
init = *init fsm1* + *init fsm2*,
final = *final fsm1* + *final fsm2*,
nxt = $\lambda st\ x\ st'.\ (\exists\, st1\ st1'.\ st = Inl\ st1 \wedge st' = Inl\ st1' \wedge nxt$
$fsm1\ st1\ x\ st1') \vee$
$(\exists\, st2\ st2'.\ st = Inr\ st2 \wedge st' = Inr\ st2' \wedge nxt\ fsm2$
$st2\ x\ st2')$|)

lemma *states-Plus* [*simp*]: *states* (*Plus fsm1 fsm2*) = *states fsm1* + *states fsm2*

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

lemma *init-Plus* [*simp*]: *init* (*Plus fsm1 fsm2*) = *init fsm1* + *init fsm2*
by (*simp add*: *Plus-def*)

lemma *final-Plus* [*simp*]: *final* (*Plus fsm1 fsm2*) = *final fsm1* + *final fsm2*
by (*simp add*: *Plus-def*)

lemma *nxt-Plus1*: $nxt\ (Plus\ fsm1\ fsm2)\ (Inl\ st1)\ x\ st' \longleftrightarrow (\exists\, st1'.\ st' = Inl\ st1'$
$\wedge\ nxt\ fsm1\ st1\ x\ st1')$
by (*simp add*: *Plus-def*)

lemma *nxt-Plus2*: $nxt\ (Plus\ fsm1\ fsm2)\ (Inr\ st2)\ x\ st' \longleftrightarrow (\exists\, st2'.\ st' = Inr\ st2'$
$\wedge\ nxt\ fsm2\ st2\ x\ st2')$
by (*simp add*: *Plus-def*)

lemma *reaches-Plus-iff1* [*simp*]:
$reaches\ (Plus\ fsm1\ fsm2)\ (Inl\ st1)\ xs\ st' \longleftrightarrow$
$(\exists\, st1'.\ st' = Inl\ st1' \wedge reaches\ fsm1\ st1\ xs\ st1')$
apply (*induct xs arbitrary*: *st1*, *force*)
apply (*force simp add*: *nxt-Plus1 reaches.Cons*)
done

lemma *reaches-Plus-iff2* [*simp*]:
$reaches\ (Plus\ fsm1\ fsm2)\ (Inr\ st2)\ xs\ st' \longleftrightarrow$
$(\exists\, st2'.\ st' = Inr\ st2' \wedge reaches\ fsm2\ st2\ xs\ st2')$
apply (*induct xs arbitrary*: *st2*, *force*)
apply (*force simp add*: *nxt-Plus2 reaches.Cons*)
done

lemma *reaches-Plus-iff* [*simp*]:

reaches (*Plus fsm1 fsm2*) *st xs st'* $\longleftrightarrow$
(∃ *st1 st1'*. *st* = *Inl st1* ∧ *st'* = *Inl st1'* ∧ *reaches fsm1 st1 xs st1'*) ∨
(∃ *st2 st2'*. *st* = *Inr st2* ∧ *st'* = *Inr st2'* ∧ *reaches fsm2 st2 xs st2'*)
apply (*induct xs arbitrary*: *st st'*, *auto*)
apply (*force simp add*: *nxt-Plus1 nxt-Plus2 Plus-def reaches.Cons*)
apply (*auto simp*: *Plus-def*)
done

lemma *accepts-Plus-iff* [*simp*]:
accepts (*Plus fsm1 fsm2*) *xs* $\longleftrightarrow$ *accepts fsm1 xs* ∨ *accepts fsm2 xs*
by (*auto simp*: *accepts-def*) (*metis sum-iff*)

lemma *regular-Un*:
assumes *S*: *regular S* **and** *T*: *regular T* **shows** *regular* (*S* ∪ *T*)
proof −
obtain *fsmS fsmT* **where** *S* = {*xs*. *accepts fsmS xs*} *T* = {*xs*. *accepts fsmT xs*}
using *S T*
by (*auto simp*: *regular-def*)
hence *S* ∪ *T* = {*xs*. *accepts* (*Plus fsmS fsmT*) *xs*}
by (*auto simp*: *accepts-Plus-iff* [*of fsmS fsmT*])
thus *?thesis*
by (*metis regular-def*)
qed

end

Bibliography

[1] L. Kirby. Addition and multiplication of sets. *Mathematical Logic Quarterly*, 53(1):52–65, 2007.

[2] S. Świerczkowski. Finite sets and Gödel's incompleteness theorems. *Dissertationes Mathematicae*, 422:1–58, 2003. http://journals.impan.gov.pl/dm/Inf/422-0-1.html.