The Hereditarily Finite Sets

Lawrence C. Paulson

March 17, 2025

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

1 The Hereditarily Finite Sets **3**
- 1.1 Basic Definitions and Lemmas . . . 3
- 1.2 Verifying the Axioms of HF . . . 5
- 1.3 Ordered Pairs, from ZF/ZF.thy . . . 6
- 1.4 Unions, Comprehensions, Intersections . . . 7
  - 1.4.1 Unions . . . 7
  - 1.4.2 Set comprehensions . . . 8
  - 1.4.3 Union operators . . . 8
  - 1.4.4 Definition 1.8, Intersections . . . 9
  - 1.4.5 Set Difference . . . 10
- 1.5 Replacement . . . 10
- 1.6 Subset relation and the Lattice Properties . . . 12
  - 1.6.1 Rules for subsets . . . 12
  - 1.6.2 Lattice properties . . . 13
- 1.7 Foundation, Cardinality, Powersets . . . 14
  - 1.7.1 Foundation . . . 14
  - 1.7.2 Cardinality . . . 15
  - 1.7.3 Powerset Operator . . . 16
- 1.8 Bounded Quantifiers . . . 17
- 1.9 Relations and Functions . . . 19
- 1.10 Operations on families of sets . . . 21
  - 1.10.1 Rules for Unions and Intersections of families . . . 21
  - 1.10.2 Generalized Cartesian product . . . 22
- 1.11 Disjoint Sum . . . 23

2 Ordinals, Sequences and Ordinal Recursion **26**
- 2.1 Ordinals . . . 26
  - 2.1.1 Basic Definitions . . . 26
  - 2.1.2 Definition 2.2 (Successor). . . . 26
  - 2.1.3 Induction, Linearity, etc. . . . 28
  - 2.1.4 Supremum and Infimum . . . 30
  - 2.1.5 Converting Between Ordinals and Natural Numbers . . . 32
- 2.2 Sequences and Ordinal Recursion . . . 33

3 V-Sets, Epsilon Closure, Ranks **38**
3.1 V-sets . . . 38
3.2 Least Ordinal Operator . . . 39
3.3 Rank Function . . . 40
3.4 Epsilon Closure . . . 42
3.5 Epsilon-Recursion . . . 43

4 An Application: Finite Automata **46**

5 Addition, Sequences and their Concatenation **53**
5.1 Generalised Addition — Also for Ordinals . . . 53
5.1.1 Cancellation laws for addition . . . 54
5.1.2 The predecessor function . . . 55
5.2 A Concatentation Operation for Sequences . . . 55
5.3 Nonempty sequences indexed by ordinals . . . 57
5.3.1 Sequence-building operators . . . 58
5.3.2 Showing that Sequences can be Constructed . . . 59
5.3.3 Proving Properties of Given Sequences . . . 62
5.4 A Unique Predecessor for every non-empty set . . . 64

Chapter 1

The Hereditarily Finite Sets

theory *HF*
imports *HOL−Library.Nat-Bijection*
abbrevs $<: = \in$
  and $^{\sim}<: = \notin$
begin

From "Finite sets and Gödel'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*

abbreviation *not-hmem* :: *hf* ⇒ *hf* ⇒ *bool* (**infixl** ‹∉› *50*)
  where *a* ∉ *b* ≡ ¬ *a* ∈ *b*

notation (*ASCII*)
  hmem (**infixl** ‹<:› *50*)

instantiation *hf* :: *zero*
begin

definition *Zero-hf-def*: $0 = HF\ \{\}$
instance ..
end

lemma *Abs-hf-0* [*simp*]: $Abs\text{-}hf\ 0 = 0$
by (*simp add*: *HF-def Zero-hf-def*)

HF Set enumerations

abbreviation *inserthf* :: $hf \Rightarrow hf \Rightarrow hf$ (**infixl** ‹◁› 60)
where $y \triangleleft x \equiv hinsert\ x\ y$

syntax (*ASCII*)
-*HFinset* :: *args* ⇒ *hf* (‹{|(-)|}›)
syntax
-*HFinset* :: *args* ⇒ *hf* (‹⦃-⦄›)
syntax-consts
-*HFinset* ⇌ *inserthf*
translations
$\{\!|x, y|\!\} \rightleftharpoons \{\!|y|\!\} \triangleleft x$
$\{\!|x|\!\} \rightleftharpoons 0 \triangleleft x$

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* $A \Longrightarrow hfset\ (HF\ A) = A$
unfolding *HF-def hfset-def*
by (*simp add*: *image-image Abs-hf-inverse Rep-hf-inverse*)

lemma *inj-on-HF*: *inj-on HF* (*Collect finite*)
by (*metis hfset-HF inj-onI mem-Collect-eq*)

lemma *hmem-hempty* [*simp*]: $a \notin 0$
unfolding *hmem-def Zero-hf-def* **by** *simp*

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

lemma *hmem-hinsert* [*iff*]:
$hmem\ a\ (c \triangleleft b) \longleftrightarrow a = b \vee a \in c$
unfolding *hmem-def hinsert-def* **by** *simp*

lemma *hf-ext*: $a = b \longleftrightarrow (\forall x.\ x \in a \longleftrightarrow x \in b)$
unfolding *hmem-def set-eq-iff* [*symmetric*]
by (*metis HF-hfset*)

lemma *finite-cases* [*consumes 1*, *case-names empty insert*]:

$[\![\mathit{finite}\ F;\ F = \{\} \Longrightarrow P;\ \bigwedge A\ x.\ [\![F = \mathit{insert}\ x\ A;\ x \notin A;\ \mathit{finite}\ A]\!] \Longrightarrow P]\!] \Longrightarrow P$
by (*induct F rule*: *finite-induct*, *simp-all*)

lemma *hf-cases* [*cases type*: *hf*, *case-names 0 hinsert*]:
obtains $y = 0 \mid a\ b$ **where** $y = b \triangleleft a$ **and** $a \notin b$
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*:
assumes $a \notin b$ **shows** *Rep-hf* (*hinsert a b*) = $2\ \widehat{}\ (\mathit{Rep\text{-}hf}\ a) + \mathit{Rep\text{-}hf}\ b$
proof −
have *Rep-hf a* $\notin$ *set-decode* (*Rep-hf b*)
by (*metis Rep-hf-inverse hfset-def hmem-def image-eqI assms*)
then show *?thesis*
by (*simp add*: *hinsert-def HF-def hfset-def image-image Abs-hf-inverse Rep-hf-inverse*)
qed

1.2 Verifying the Axioms of HF

HF1

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

HF2

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

HF induction

lemma *hf-induct* [*induct type*: *hf*, *case-names 0 hinsert*]:
assumes [*simp*]: *P 0*
$\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ x \notin y]\!] \Longrightarrow P\ (y \triangleleft x)$
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-exp le-add1*])
qed
qed

HF3

lemma *hf-induct-ax*: $[\![P\ 0;\ \forall x.\ P\ x \longrightarrow (\forall y.\ P\ y \longrightarrow P\ (x \lhd y))]\!] \Longrightarrow P\ x$
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*]: $A \lhd a \neq 0$
by (*auto simp*: *hf-ext*)

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

lemma *hmem-HF-iff* [*simp*]: $x \in HF\ A \longleftrightarrow x \in A \wedge finite\ A$
by (*metis Abs-hf-0 HF-def Rep-hf-inverse finite-imageD hemptyE hfset-HF hmem-def inj-onI set-encode-inf*)

1.3 Ordered Pairs, from ZF/ZF.thy

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

lemma *doubleton-eq-iff*: $\{\!|a,b|\!\} = \{\!|c,d|\!\} \longleftrightarrow (a{=}c \wedge b{=}d) \vee (a{=}d \wedge b{=}c)$
by *auto* (*metis hmem-hempty hmem-hinsert*)+

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

definition *hfst* :: *hf* ⇒ *hf*
where *hfst p* ≡ *THE x*. ∃ *y*. *p* = *hpair x y*

definition *hsnd* :: *hf* ⇒ *hf*
where *hsnd p* ≡ *THE y*. ∃ *x*. *p* = *hpair x y*

definition *hsplit* :: [[*hf*, *hf*] ⇒ *'a*, *hf*] ⇒ *'a*::{} — for pattern-matching
where *hsplit c* ≡ λ*p*. *c* (*hfst p*) (*hsnd p*)

Ordered Pairs, from ZF/ZF.thy

nonterminal *hfs*
syntax (*ASCII*)
-Tuple :: [*hf*, *hfs*] ⇒ *hf* (‹<(-,/ -)>›)
-hpattern :: [*pttrn*, *patterns*] ⇒ *pttrn* (‹<-,/ ->›)
syntax
:: *hf* ⇒ *hfs* (‹-›)
-Enum :: [*hf*, *hfs*] ⇒ *hfs* (‹-,/ -›)
-Tuple :: [*hf*, *hfs*] ⇒ *hf* (‹⟨(-,/ -)⟩›)
-hpattern :: [*pttrn*, *patterns*] ⇒ *pttrn* (‹⟨-,/ -⟩›)
syntax-consts
-Enum -Tuple ⇌ *hpair* **and**
-hpattern ⇌ *hsplit*

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

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

lemma *hpair-iff* [*simp*]: $hpair\ a\ b = hpair\ a'\ b' \longleftrightarrow a=a' \wedge b=b'$
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*: $\exists z.\ \forall u.\ u \in z \longleftrightarrow u \in x \vee u \in y$
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*: $\exists z.\ \forall u.\ u \in z \longleftrightarrow (\exists y.\ y \in x \wedge u \in y)$
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*: $\exists\, z.\ \forall\, u.\ u \in z \longleftrightarrow u \in x \wedge P\ u$
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* ⇒ *bool*) ⇒ *hf* ⇒ *hf* — comprehension
  where *HCollect P A* = (*THE z*. ∀ *u*. *u* ∈ *z* = (*P u* ∧ *u* ∈ *A*))

syntax (*ASCII*)
  -HCollect :: *idt* ⇒ *hf* ⇒ *bool* ⇒ *hf* (‹(*1*{|- <:/ -./ -|})›)
syntax
  -HCollect :: *idt* ⇒ *hf* ⇒ *bool* ⇒ *hf* (‹(*1*{|- ∈/ -./ -|})›)
syntax-consts
  -HCollect ⇌ *HCollect*
translations
  {|*x* ∈ *A*. *P*|} ⇌ *CONST HCollect* (λ*x*. *P*) *A*

lemma *HCollect-iff* [*iff*]: *hmem x* (*HCollect P A*) ⟷ *P x* ∧ *x* ∈ *A*
  using *comprehension* [*of A P*]
  apply (*clarsimp simp*: *HCollect-def*)
  apply (*rule theI2*, *blast*)
   apply (*auto simp*: *hf-ext*)
  done

lemma *HCollectI*: *a* ∈ *A* ⟹ *P a* ⟹ *hmem a* {|*x* ∈ *A*. *P x*|}
  by *simp*

lemma *HCollectE*:
  assumes *a* ∈ {|*x* ∈ *A*. *P x*|} **obtains** *a* ∈ *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 a b* = (*THE z*. ∀ *u*. *u* ∈ *z* ⟷ *u* ∈ *a* ∨ *u* ∈ *b*)
  instance ..
end

abbreviation *hunion* :: *hf* ⇒ *hf* ⇒ *hf* (**infixl** ‹⊔› *65*) **where**
  hunion ≡ *sup*

lemma *hunion-iff* [*iff*]: *hmem x* (*a* ⊔ *b*) ⟷ *x* ∈ *a* ∨ *x* ∈ *b*
using *binary-union* [*of a b*]
apply (*clarsimp simp*: *sup-hf-def*)
apply (*rule theI2*, *auto simp*: *hf-ext*)
done

definition *HUnion* :: *hf* ⇒ *hf* (‹⨆ -› [*900*] *900*)
where *HUnion A* = (*THE z*. ∀ *u*. *u* ∈ *z* ⟷ (∃ *y*. *y* ∈ *A* ∧ *u* ∈ *y*))

lemma *HUnion-iff* [*iff*]: *hmem x* (⨆ *A*) ⟷ (∃ *y*. *y* ∈ *A* ∧ *x* ∈ *y*)
using *union-of-set* [*of A*]
apply (*clarsimp simp*: *HUnion-def*)
apply (*rule theI2*, *auto simp*: *hf-ext*)
done

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

lemma *HUnion-hinsert* [*simp*]: ⨆(*A* ◁ *a*) = *a* ⊔ ⨆*A*
by (*auto simp*: *hf-ext*)

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

1.4.4 Definition 1.8, Intersections

instantiation *hf* :: *inf*
begin
definition *inf a b* = {‖*x* ∈ *a*. *x* ∈ *b*‖}
instance ..
end

abbreviation *hinter* :: *hf* ⇒ *hf* ⇒ *hf* (**infixl** ‹⊓› *70*) **where**
hinter ≡ *inf*

lemma *hinter-iff* [*iff*]: *hmem u* (*x* ⊓ *y*) ⟷ *u* ∈ *x* ∧ *u* ∈ *y*
by (*metis HCollect-iff inf-hf-def*)

definition *HInter* :: *hf* ⇒ *hf* (‹⨅ -› [*900*] *900*)
where *HInter*(*A*) = {‖*x* ∈ *HUnion*(*A*). ∀ *y*. *y* ∈ *A* ⟶ *x* ∈ *y*‖}

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

lemma *HInter-iff* [*simp*]: *A*≠*0* ⟹ *hmem x* (⨅ *A*) ⟷ (∀ *y*. *y* ∈ *A* ⟶ *x* ∈ *y*)
by (*auto simp*: *HInter-def*)

lemma *HInter-hinsert* [*simp*]: *A*≠*0* ⟹ ⨅(*A* ◁ *a*) = *a* ⊓ ⨅*A*

by (*auto simp*: *hf-ext HInter-iff* [*OF hinsert-nonempty*])

1.4.5 Set Difference

instantiation *hf* :: *minus*
begin
definition $A - B = \{\!|x \in A.\ x \notin B|\!\}$
instance ..
end

lemma *hdiff-iff* [*iff*]: *hmem u* $(x - y) \longleftrightarrow u \in x \wedge u \notin y$
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*:
$(\forall u\ v\ v'.\ u \in x \longrightarrow R\ u\ v \longrightarrow R\ u\ v' \longrightarrow v'{=}v) \Longrightarrow \exists z.\ \forall v.\ v \in z \longleftrightarrow (\exists u.\ u \in x \wedge R\ u\ v)$
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*: $\exists z.\ \forall v.\ v \in z \longleftrightarrow (\exists u.\ u \in x \wedge v = f\ u)$
by (*rule replacement* [**where** $R = \lambda u\ v.\ v = f\ u$]) *auto*

definition *PrimReplace* :: *hf* ⇒ (*hf* ⇒ *hf* ⇒ *bool*) ⇒ *hf*
where *PrimReplace A R* = (*THE z*. ∀ *v*. *v* ∈ *z* ⟷ (∃ *u*. *u* ∈ *A* ∧ *R u v*))

definition *Replace* :: *hf* ⇒ (*hf* ⇒ *hf* ⇒ *bool*) ⇒ *hf*
where *Replace A R* = *PrimReplace A* (λ*x y*. (∃!*z*. *R x z*) ∧ *R x y*)

definition *RepFun* :: *hf* ⇒ (*hf* ⇒ *hf*) ⇒ *hf*
  where *RepFun A f* = *Replace A* (λ*x y*. *y* = *f x*)

syntax (*ASCII*)
  -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
  -HReplace :: [*pttrn*, *pttrn*, *hf*, *bool*] ⇒ *hf* (‹(*1*⦃- ./ - ∈ -, -⦄)›)
  -HRepFun :: [*hf*, *pttrn*, *hf*] ⇒ *hf* (‹(*1*⦃- ./ - ∈ -⦄)› [*51*,*0*,*51*])
  -HINTER :: [*pttrn*, *hf*, *hf*] ⇒ *hf* (‹(*3*⋂ -∈-./ -)› *10*)
  -HUNION :: [*pttrn*, *hf*, *hf*] ⇒ *hf* (‹(*3*⋃ -∈-./ -)› *10*)
syntax-consts
  -HReplace ⇌ *Replace* **and**
  -HRepFun ⇌ *RepFun* **and**
  -HINTER ⇌ *HInter* **and**
  -HUNION ⇌ *HUnion*
translations
  ⦃*y*. *x*∈*A*, *Q*⦄ ⇌ *CONST Replace A* (λ*x y*. *Q*)
  ⦃*b*. *x*∈*A*⦄ ⇌ *CONST RepFun A* (λ*x*. *b*)
  ⋂*x*∈*A*. *B* ⇌ *CONST HInter*(*CONST RepFun A* (λ*x*. *B*))
  ⋃*x*∈*A*. *B* ⇌ *CONST HUnion*(*CONST RepFun A* (λ*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*)
  using *replacement* [*OF sv*]
  apply (*clarsimp simp*: *PrimReplace-def*)
  apply (*rule theI2*, *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*))
  unfolding *Replace-def*
  by (*smt* (*verit*, *ccfv-threshold*) *PrimReplace-iff*)

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*]: $v \in (\mathit{RepFun}\ A\ f) \longleftrightarrow (\exists\, u.\ u \in A \wedge v = f\ u)$
by (*auto simp*: *RepFun-def*)

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

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

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

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

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

lemma *HF-HUnion*: $[\![\mathit{finite}\ A;\ \bigwedge x.\ x{\in}A \Longrightarrow \mathit{finite}\ (B\ x)]\!] \Longrightarrow \mathit{HF}\ (\bigcup x \in A.\ B\ x) = (\bigsqcup x{\in}\mathit{HF}\ A.\ \mathit{HF}\ (B\ x))$
by (*rule hf-equalityI*) (*auto*)

1.6 Subset relation and the Lattice Properties

Definition 1.10 (Subset relation).

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

definition *less-hf* **where** $A < B \longleftrightarrow A \leq B \wedge A \neq (B{::}\mathit{hf})$

instance by *standard* (*auto simp*: *less-eq-hf-def less-hf-def*)
end

1.6.1 Rules for subsets

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

Classical elimination rule

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

Rule in Modus Ponens style

lemma *hsubsetD* [*elim*]: ⟦ $A \leq B$; $c \in A$ ⟧ ⟹ $c \in B$
by *auto*

Sometimes useful with premises in this order

lemma *rev-hsubsetD*: ⟦ $c \in A$; $A \leq B$ ⟧ ⟹ $c \in B$
by *blast*

lemma *contra-hsubsetD*: ⟦ $A \leq B$; $c \notin B$ ⟧ ⟹ $c \notin A$
by *blast*

lemma *rev-contra-hsubsetD*: ⟦ $c \notin B$; $A \leq B$ ⟧ ⟹ $c \notin A$
by *blast*

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

1.6.2 Lattice properties

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

instantiation *hf* :: *bounded-lattice-bot*
begin
definition $bot = (0::hf)$
instance by *standard* (*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*]: $u \leq 0 \longleftrightarrow u = (0::hf)$
by (*metis hempty-iff less-eq-hf-def*)

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

lemma *less-eq-insert2-iff*:
$z \leq (hinsert\ x\ y) \longleftrightarrow z \leq y \vee (\exists u.\ hinsert\ x\ u = z \wedge x \notin u \wedge u \leq y)$

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 $z \leq y \vee (\exists\, u.\ \mathit{hinsert}\ x\ u = z \wedge x \notin u \wedge u \leq y)$
        by (*simp add*: *less-eq-hf-def*) (*metis u hdiff-iff hmem-hinsert*)
    next
      assume $z \leq y \vee (\exists\, u.\ \mathit{hinsert}\ x\ u = z \wedge x \notin u \wedge u \leq y)$
      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** $\exists\, w.\ w \in z \wedge w \sqcap z = 0$
proof −
  { fix x
    assume *z*: $(\forall\, w.\ w \in z \longrightarrow w \sqcap z \neq 0)$
    have $x \notin z \wedge x \sqcap z = 0$

```
    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 ∉ x
  using foundation [of {|x|}]
  by (metis hinter-iff hmem-hempty hmem-hinsert)

lemma hmem-not-sym: ¬ (x ∈ y ∧ y ∈ x)
  using foundation [of {|x,y|}]
  by (metis hinter-iff hmem-hempty hmem-hinsert)

lemma hmem-ne: x ∈ y ⟹ x ≠ y
  by (metis hmem-not-refl)

lemma hmem-Sup-ne: x ∈ y ⟹ ⨆ x ≠ y
  by (metis HUnion-iff hmem-not-sym)

lemma hpair-neq-fst: ⟨a,b⟩ ≠ a
  by (metis hpair-def hinsert-iff hmem-not-sym)

lemma hpair-neq-snd: ⟨a,b⟩ ≠ b
  by (metis hpair-def hinsert-iff hmem-not-sym)

lemma hpair-nonzero [simp]: ⟨x,y⟩ ≠ 0
  by (auto simp: hpair-def)

lemma zero-notin-hpair: 0 ∉ ⟨x,y⟩
  by (auto simp: hpair-def)
```

1.7.2 Cardinality

First we need to hack the underlying representation

```
lemma hfset-0 [simp]: hfset 0 = {}
  by (metis Zero-hf-def finite.emptyI hfset-HF)

lemma hfset-hinsert: hfset (b ◃ 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*
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*)

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*
proof (*induct x arbitrary*: *y rule*: *hf-induct*)
next
case (*hinsert x y*)
then show *?case*
by (*simp add*: *hcard-hinsert-if hinter-hinsert-left hunion-hinsert-left*)
qed *auto*

lemma *hcard-hdiff1-less*: *x* ∈ *z* ⟹ *hcard* (*z* − {|*x*|}) < *hcard z*
unfolding *hmem-def*
by (*metis card-Diff1-less finite-hfset hcard-def hfset-0 hfset-hdiff hfset-hinsert*)

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*]
unfolding *less-eq-insert2-iff*
by (*smt* (*verit*, *ccfv-threshold*) *hinsert.hyps less-eq-hf-def*)
qed

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

lemma *HPow-iff* [*iff*]: $u \in HPow\ x \longleftrightarrow u \leq x$
using *powerset* [*of x*]
apply (*clarsimp simp*: *HPow-def*)
apply (*rule theI2*, *auto simp*: *hf-ext*)
done

lemma *HPow-mono*: $x \leq y \Longrightarrow HPow\ x \leq HPow\ y$
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*]: $HPow\ x \leq HPow\ y \longleftrightarrow x \leq y$
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**
$HBall\ A\ P \longleftrightarrow (\forall x.\ x \in A \longrightarrow P\ x)$ — bounded universal quantifiers

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

syntax (*ASCII*)
-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
-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-consts
-HBall ⇌ *HBall* **and**
-HBex ⇌ *HBex* **and**
-HBex1 ⇌ *Ex1*
translations
$\forall x{\in}A.\ P \rightleftharpoons CONST\ HBall\ A\ (\lambda x.\ P)$
$\exists x{\in}A.\ P \rightleftharpoons CONST\ HBex\ A\ (\lambda x.\ P)$
$\exists!x{\in}A.\ P \rightharpoonup \exists!x.\ x{\in}A \wedge P$

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

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

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

lemma *hballE* [*elim*]: $\forall\, x{\in}A.\ P\ x \Longrightarrow (P\ x \Longrightarrow Q) \Longrightarrow (x \notin A \Longrightarrow Q) \Longrightarrow Q$
by (*force simp add*: *HBall-def*)

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

lemma *hbexI* [*intro*]: $P\ x \Longrightarrow x{\in}A \Longrightarrow \exists\, x{\in}A.\ P\ x$
and *rev-hbexI* [*intro?*]: $x{\in}A \Longrightarrow P\ x \Longrightarrow \exists\, x{\in}A.\ P\ x$
and *bexCI*: $(\forall\, x{\in}A.\ \neg\ P\ x \Longrightarrow P\ a) \Longrightarrow a{\in}A \Longrightarrow \exists\, x{\in}A.\ P\ x$
and *hbexE* [*elim*!]: $\exists\, x{\in}A.\ P\ x \Longrightarrow (\bigwedge x.\ x{\in}A \Longrightarrow P\ x \Longrightarrow Q) \Longrightarrow Q$
using *HBex-def* **by** *auto*

lemma *hball-triv* [*simp*]: $(\forall\, x{\in}A.\ P) = ((\exists\, x.\ x{\in}A) \longrightarrow P)$
and *hbex-triv* [*simp*]: $(\exists\, x{\in}A.\ P) = ((\exists\, x.\ x{\in}A) \wedge P)$
— Dual form for existentials.
by *blast+*

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

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

lemma *hbex-one-point1* [*simp*]: $(\exists\, x{\in}A.\ x = a \wedge P\ x) = (a{\in}A \wedge P\ a)$
by *blast*

lemma *hbex-one-point2* [*simp*]: $(\exists\, x{\in}A.\ a = x \wedge P\ x) = (a{\in}A \wedge P\ a)$
by *blast*

lemma *hball-one-point1* [*simp*]: $(\forall\, x{\in}A.\ x = a \longrightarrow P\ x) = (a{\in}A \longrightarrow P\ a)$
by *blast*

lemma *hball-one-point2* [*simp*]: $(\forall\, x{\in}A.\ a = x \longrightarrow P\ x) = (a{\in}A \longrightarrow P\ a)$
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*]:
$\bigwedge A\ P\ Q.\ (\forall\, x \in A.\ P\ x \vee Q) \longleftrightarrow ((\forall\, x \in A.\ P\ x) \vee Q)$
$\bigwedge A\ P\ Q.\ (\forall\, x \in A.\ P \vee Q\ x) \longleftrightarrow (P \vee (\forall\, x \in A.\ Q\ x))$
$\bigwedge A\ P\ Q.\ (\forall\, x \in A.\ P \longrightarrow Q\ x) \longleftrightarrow (P \longrightarrow (\forall\, x \in A.\ Q\ x))$
$\bigwedge A\ P\ Q.\ (\forall\, x \in A.\ P\ x \longrightarrow Q) \longleftrightarrow ((\exists\, x \in A.\ P\ x) \longrightarrow Q)$
$\bigwedge P.\ (\forall\, x \in 0.\ P\ x) \longleftrightarrow \mathit{True}$
$\bigwedge a\ B\ P.\ (\forall\, x \in B \triangleleft a.\ P\ x) \longleftrightarrow (P\ a \wedge (\forall\, x \in B.\ P\ x))$
$\bigwedge P\ Q.\ (\forall\, x \in \mathit{HCollect}\ Q\ A.\ P\ x) \longleftrightarrow (\forall\, x \in A.\ Q\ x \longrightarrow P\ x)$
$\bigwedge A\ P.\ (\neg\ (\forall\, x \in A.\ P\ x)) \longleftrightarrow (\exists\, x \in A.\ \neg\ P\ x)$
by *auto*

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

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

1.9 Relations and Functions

definition *is-hpair* :: *hf* ⇒ *bool*
where $\mathit{is\text{-}hpair}\ z = (\exists\, x\ y.\ z = \langle x,y\rangle)$

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

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

definition *hrange* :: *hf* ⇒ *hf*
where $\mathit{hrange}(r) = \mathit{hdomain}(\mathit{hconverse}(r))$

definition *hrelation* :: *hf* ⇒ *bool*
where $\mathit{hrelation}(r) = (\forall\, z.\ z \in r \longrightarrow \mathit{is\text{-}hpair}\ z)$

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

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

definition *hfunction* :: *hf* ⇒ *bool*
  where *hfunction*(*r*) = (∀ *x y*. ⟨*x*,*y*⟩ ∈ *r* ⟶ (∀ *y′*. ⟨*x*,*y′*⟩ ∈ *r* ⟶ *y*=*y′*))

definition *app* :: *hf* ⇒ *hf* ⇒ *hf*
  where *app f x* = (*THE y*. ⟨*x*, *y*⟩ ∈ *f*)

lemma *hrestrict-iff* [*iff*]:
    z ∈ *hrestrict r A* ⟷ *z* ∈ *r* ∧ (∃ *x y*. *z* = ⟨*x*, *y*⟩ ∧ *x* ∈ *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* ⊔ *g*) ⟷ *hrelation f* ∧ *hrelation g*
  by (*auto simp*: *hrelation-def*)

lemma *hfunction-restr*: *hfunction r* ⟹ *hfunction* (*hrestrict r x*)
  by (*auto simp*: *hfunction-def hrestrict-def*)

lemma *hdomain-restr* [*simp*]: *hdomain* (*hrestrict r x*) = *hdomain r* ⊓ *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* ◁ ⟨*x*, *y*⟩) = *hdomain r* ◁ *x*
  by (*force simp add*: *hdomain-def*)

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

lemma *hdomain-not-mem* [*iff*]: ⟨*hdomain r*, *a*⟩ ∉ *r*
  by (*metis hdomain-ins hinter-hinsert-right hmem-hinsert hmem-not-refl*
        hunion-hinsert-right sup-inf-absorb)

lemma *app-singleton* [*simp*]: *app* ⦃⟨*x*, *y*⟩⦄ *x* = *y*
  by (*simp add*: *app-def*)

lemma *app-equality*: *hfunction f* ⟹ ⟨*x*, *y*⟩ ∈ *f* ⟹ *app f x* = *y*
  by (*auto simp*: *app-def hfunction-def intro*: *the1I2*)

lemma *app-ins2*: *x′* ≠ *x* ⟹ *app* (*f* ◁ ⟨*x*, *y*⟩) *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* $\Longrightarrow$ $x \notin$ *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*
$\Longrightarrow$ *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*)

1.10 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* = $\{\!|\ f \in HPow(HSigma\ A\ B).\ A \leq hdomain(f) \wedge hfunction(f)|\!\}$

syntax (*ASCII*)
-PROD :: [*pttrn*, *hf*, *hf*] $\Rightarrow$ *hf* (‹(*3PROD -<:-./ -*)› *10*)
-SUM :: [*pttrn*, *hf*, *hf*] $\Rightarrow$ *hf* (‹(*3SUM -<:-./ -*)› *10*)
-lam :: [*pttrn*, *hf*, *hf*] $\Rightarrow$ *hf* (‹(*3lam -<:-./ -*)› *10*)
syntax
-PROD :: [*pttrn*, *hf*, *hf*] $\Rightarrow$ *hf* (‹(*3*$\prod$*-∈-./ -*)› *10*)
-SUM :: [*pttrn*, *hf*, *hf*] $\Rightarrow$ *hf* (‹(*3*$\sum$*-∈-./ -*)› *10*)
-lam :: [*pttrn*, *hf*, *hf*] $\Rightarrow$ *hf* (‹(*3*λ*-∈-./ -*)› *10*)
syntax-consts
-PROD $\rightleftharpoons$ *HPi* **and**
-SUM $\rightleftharpoons$ *HSigma* **and**
-lam $\rightleftharpoons$ *HLambda*
translations
$\prod x{\in}A.\ B \rightleftharpoons$ *CONST HPi A* ($\lambda x.\ B$)
$\sum x{\in}A.\ B \rightleftharpoons$ *CONST HSigma A* ($\lambda x.\ B$)
$\lambda x{\in}A.\ f \rightleftharpoons$ *CONST HLambda A* ($\lambda x.\ f$)

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

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

1.10.2 Generalized Cartesian product

lemma *HSigma-iff* [*simp*]: $\langle a,b\rangle \in HSigma\ A\ B \longleftrightarrow a \in A \wedge b \in B(a)$
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 $A * B = HSigma\ A\ (\lambda x.\ B)$
instance ..
end

lemma *times-iff* [*simp*]: $\langle a,b\rangle \in A * B \longleftrightarrow a \in A \wedge b \in B$
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*: $A*B=0 \longleftrightarrow A=0 \vee B=(0::hf)$
by (*auto simp*: *times-hf-def hf-ext*)

instantiation *hf* :: *mult-zero*
begin
instance by *standard auto*
end

1.11 Disjoint Sum

instantiation *hf* :: *zero-neq-one*
begin
definition *One-hf-def*: $1 = \{\!|0|\!\}$
instance by *standard* (*auto simp*: *One-hf-def*)
end

instantiation *hf* :: *plus*
begin
definition $A + B = (\{\!|0|\!\} * A) \sqcup (\{\!|1|\!\} * B)$
instance ..
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* $\wedge$ *u*=*Inl*(*x*)) $\vee$ ($\exists$ *y*. *y* $\in$ *B* $\wedge$ *u*=*Inr*(*y*))
by *blast*

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

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

definition *is-hsum* :: *hf* ⇒ *bool*
  where *is-hsum z* = (∃ *x*. *z* = *Inl x* ∨ *z* = *Inr x*)

definition *sum-case* :: (*hf* ⇒ *'a*) ⇒ (*hf* ⇒ *'a*) ⇒ *hf* ⇒ *'a*
  where
  sum-case f g a ≡
    THE z. (∀ *x*. *a* = *Inl x* ⟶ *z* = *f x*) ∧ (∀ *y*. *a* = *Inr y* ⟶ *z* = *g y*) ∧ (¬ *is-hsum a* ⟶ *z* = *undefined*)

lemma *sum-case-Inl* [*simp*]: *sum-case f g* (*Inl x*) = *f x*
  by (*simp add*: *sum-case-def is-hsum-def*)

lemma *sum-case-Inr* [*simp*]: *sum-case f g* (*Inr y*) = *g y*
  by (*simp add*: *sum-case-def is-hsum-def*)

lemma *sum-case-non* [*simp*]: ¬ *is-hsum a* ⟹ *sum-case f g a* = *undefined*
  by (*simp add*: *sum-case-def is-hsum-def*)

lemma *is-hsum-cases*: (∃ *x*. *z* = *Inl x* ∨ *z* = *Inr x*) ∨ ¬ *is-hsum z*
  by (*auto simp*: *is-hsum-def*)

lemma *sum-case-split*:
  P (*sum-case f g a*) ⟷ (∀ *x*. *a* = *Inl x* ⟶ *P*(*f x*)) ∧ (∀ *y*. *a* = *Inr y* ⟶ *P*(*g y*)) ∧ (¬ *is-hsum a* ⟶ *P undefined*)
  by (*cases is-hsum a*) (*auto simp*: *is-hsum-def*)

lemma *sum-case-split-asm*:
  P (*sum-case f g a*) ⟷ ¬ ((∃ *x*. *a* = *Inl x* ∧ ¬ *P*(*f x*)) ∨ (∃ *y*. *a* = *Inr y* ∧ ¬ *P*(*g y*)) ∨ (¬ *is-hsum a* ∧ ¬ *P undefined*))
  by (*auto simp add*: *sum-case-split*)

end

Chapter 2

Ordinals, Sequences and Ordinal Recursion

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

2.1 Ordinals

2.1.1 Basic Definitions

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

definition
Transset :: *hf* ⇒ *bool* **where**
Transset(*x*) ≡ ∀ *y*. *y* ∈ *x* ⟶ *y* ≤ *x*

lemma *Transset-sup*: *Transset x* ⟹ *Transset y* ⟹ *Transset* (*x* ⊔ *y*)
by (*auto simp*: *Transset-def*)

lemma *Transset-inf*: *Transset x* ⟹ *Transset y* ⟹ *Transset* (*x* ⊓ *y*)
by (*auto simp*: *Transset-def*)

lemma *Transset-hinsert*: *Transset x* ⟹ *y* ≤ *x* ⟹ *Transset* (*x* ◁ *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* ⇒ *bool* **where**
Ord(*k*) ≡ *Transset*(*k*) ∧ (∀ *x* ∈ *k*. *Transset*(*x*))

2.1.2 Definition 2.2 (Successor).

definition

succ :: *hf* ⇒ *hf* **where**
succ(*x*) ≡ *hinsert x x*

lemma *succ-iff* [*simp*]: *x* ∈ *succ y* ⟷ *x*=*y* ∨ *x* ∈ *y*
by (*simp add*: *succ-def*)

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

lemma *succ-notin-self*: *succ i* ∉ *i*
by (*metis hmem-ne succ-iff*)

lemma *succE* [*elim?*]: **assumes** *x* ∈ *succ y* **obtains** *x*=*y* | *x* ∈ *y*
by (*metis assms succ-iff*)

lemma *hmem-succ-ne*: *succ x* ∈ *y* ⟹ *x* ≠ *y*
by (*metis hmem-not-refl succ-iff*)

lemma *hball-succ* [*simp*]: (∀ *x* ∈ *succ k*. *P x*) ⟷ *P k* ∧ (∀ *x* ∈ *k*. *P x*)
by (*auto simp*: *HBall-def*)

lemma *hbex-succ* [*simp*]: (∃ *x* ∈ *succ k*. *P x*) ⟷ *P k* ∨ (∃ *x* ∈ *k*. *P x*)
by (*auto simp*: *HBex-def*)

lemma *One-hf-eq-succ*: *1* = *succ 0*
by (*metis One-hf-def succ-def*)

lemma *zero-hmem-one* [*iff*]: *x* ∈ *1* ⟷ *x* = *0*
by (*metis One-hf-eq-succ hmem-hempty succ-iff*)

lemma *hball-One* [*simp*]: (∀ *x*∈*1*. *P x*) = *P 0*
by (*simp add*: *One-hf-eq-succ*)

lemma *hbex-One* [*simp*]: (∃ *x*∈*1*. *P x*) = *P 0*
by (*simp add*: *One-hf-eq-succ*)

lemma *hpair-neq-succ* [*simp*]: ⟨*x*,*y*⟩ ≠ *succ k*
by (*auto simp*: *succ-def hpair-def*) (*metis hemptyE hmem-hinsert hmem-ne*)

lemma *succ-neq-hpair* [*simp*]: *succ k* ≠ ⟨*x*,*y*⟩
by (*metis hpair-neq-succ*)

lemma *hpair-neq-one* [*simp*]: ⟨*x*,*y*⟩ ≠ *1*
by (*metis One-hf-eq-succ hpair-neq-succ*)

lemma *one-neq-hpair* [*simp*]: *1* ≠ ⟨*x*,*y*⟩
by (*metis hpair-neq-one*)

lemma *hmem-succ-self* [*simp*]: *k* ∈ *succ k*

by (*metis succ-iff*)

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

Theorem 2.3.

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

lemma *Ord-succ*: $Ord(k) \Longrightarrow Ord(succ(k))$
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*: $Ord(k) \Longrightarrow j \in k \Longrightarrow j \leq k$
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*)

2.1.3 Induction, Linearity, etc.

lemma *Ord-induct* [*consumes 1*, *case-names step*]:
assumes *k*: *Ord*(*k*)
and *step*: $\bigwedge x.[\![\ Ord(x);\ \ \bigwedge y.\ y \in x \Longrightarrow P(y)\ ]\!] \Longrightarrow P(x)$
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*: $Ord(k) \Longrightarrow Ord(l) \Longrightarrow k{\in}l \vee k{=}l \vee l{\in}k$
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*∈*l* | (*eq*) *k*=*l* | (*gt*) *l*∈*k*
by (*metis Ord-linear o*)

lemma *Ord-linear2*:
assumes *o*: *Ord*(*k*) *Ord*(*l*)
obtains (*lt*) *k*∈*l* | (*ge*) $l \leq k$
by (*metis Ord-linear OrdmemD order-eq-refl o*)

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

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

Theorem 2.5

lemma *Ord-mem-iff-lt*: *Ord*(*k*) ⟹ *Ord*(*l*) ⟹ *k*∈*l* ⟷ $k < l$
by (*metis Ord-linear OrdmemD hmem-not-refl less-hf-def less-le-not-le*)

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

lemma *le-succ-iff*: *Ord i* ⟹ *Ord j* ⟹ *succ i* ≤ *succ j* ⟷ $i \leq j$
by (*metis Ord-linear-le Ord-succ le-succE order-antisym*)

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

lemma *mem-succ-iff* [*simp*]: *Ord j* ⟹ *succ i* ∈ *succ j* ⟷ $i \in j$
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* ∈ *k*
shows *succ l* = *k* ∨ *succ l* ∈ *k*
by (*metis assms mem-succ-iff succ-iff*)

2.1.4 Supremum and Infimum

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

lemma *Ord-Inter* [*intro*,*simp*]:
assumes $\bigwedge i.\ i{\in}A \Longrightarrow Ord(i)$ **shows** $Ord(\bigsqcap A)$
proof (*cases A=0*)
case *False*
with *assms* **show** *?thesis*
by (*fastforce simp add*: *Ord-def Transset-def*)
qed *auto*

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*: $[\![A{\neq}0;\ \bigwedge i.\ i{\in}A \Longrightarrow Ord(i)]\!] \Longrightarrow \bigsqcup A \in A$
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 $\bigsqcup A \leq x \vee x \leq \bigsqcup A$
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*: $[\![A{\neq}0;\ \bigwedge i.\ i{\in}A \Longrightarrow Ord(i)]\!] \Longrightarrow \bigsqcap A \in A$
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 $\bigsqcap A \leq x \vee x \leq \bigsqcap A$
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*]: $\neg\ Ord\ \langle x,y\rangle$
by (*metis hpair-neq-Ord*)

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

lemma *Ord-not-hpair*: *Ord x* $\Longrightarrow$ $\neg$ *is-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*)

2.1.5 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-of i* = *ord-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* $\Rightarrow$ *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* $\Longrightarrow$ *ord-of* (*nat-of-ord x*) = *x*
proof (*induction x rule*: *Ord-induct2*)
case (*succ k*)
then show *?case*
by (*metis nat-of-ord-ord-of ord-of.simps*(*2*))
qed *auto*

lemma *nat-of-ord-inject*: *Ord x* $\Longrightarrow$ *Ord y* $\Longrightarrow$ *nat-of-ord x* = *nat-of-ord y* $\longleftrightarrow$ *x* = *y*
by (*metis ord-of-nat-of-ord*)

lemma *nat-of-ord-succ* [*simp*]: *Ord x* $\Longrightarrow$ *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*)

lemma *inj-ord-of*: *inj-on ord-of A*
by (*simp add*: *inj-on-def*)

lemma *hfset-ord-of*: *hfset* (*ord-of n*) = *ord-of* ‘ {*0*..<*n*}
by (*induct n*) (*auto simp*: *hfset-hinsert succ-def*)

lemma *bij-betw-ord-of*: *bij-betw ord-of* {*0*..<*n*} (*hfset* (*ord-of n*))
by (*simp add*: *bij-betw-def inj-ord-of hfset-ord-of*)

lemma *bij-betw-ord-ofI*:
bij-betw h A {*0*..<*n*} ⟹ *bij-betw* (*ord-of* ∘ *h*) *A* (*hfset* (*ord-of n*))
by (*blast intro*: *bij-betw-ord-of bij-betw-trans*)

2.2 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*) ⟹ *Seq* (*hrestrict s k*) *k*
by (*simp add*: *Seq-def hfunction-restr succ-def*)

lemma *Seq-Ord-restr*: ⟦*Seq s k*; *l* ∈ *k*; *Ord k*⟧ ⟹ *Seq* (*hrestrict s l*) *l*
by (*auto simp*: *Seq-def hfunction-restr intro*: *Ord-trans*)

lemma *Seq-ins*: ⟦*Seq s k*; *k* ∉ *hdomain s*⟧ ⟹ *Seq* (*s* ◃ ⟨*k*, *y*⟩) (*succ k*)
by (*auto simp*: *Seq-def hrelation-def succ-def hfunction-def hdomainI*)

definition *insf* :: *hf* ⇒ *hf* ⇒ *hf* ⇒ *hf*
where *insf s k y* ≡ *nonrestrict s* {|*k*|} ◃ ⟨*k*, *y*⟩

lemma *hrelation-insf*: *hrelation s* ⟹ *hrelation* (*insf s k y*)
by (*simp add*: *hrelation-def insf-def nonrestrict-def*)

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

lemma *hdomain-insf*: $k \leq hdomain\ s \Longrightarrow succ\ k \leq hdomain\ (insf\ s\ k\ y)$
unfolding *insf-def*
using *hdomain-def hdomain-ins less-eq-hf-def nonrestrict-def* **by** *fastforce*

lemma *Seq-insf*: $Seq\ s\ k \Longrightarrow Seq\ (insf\ s\ k\ y)\ (succ\ k)$
by (*simp add*: *Ordinal.Seq-def hdomain-insf hfunction-insf hrelation-insf*)

lemma *Seq-succ-iff*: $Seq\ s\ (succ\ k) \longleftrightarrow Seq\ s\ k \wedge (\exists\, y.\ \langle k,\ y\rangle \in s)$
using *Ordinal.Seq-def hdomain-def succ-def* **by** *auto*

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*: $Seq\ s\ k \Longrightarrow z \notin X \Longrightarrow app\ (nonrestrict\ s\ X)\ z = app\ s\ z$
by (*auto simp*: *Seq-def nonrestrict-def app-def HBall-def*) (*metis*)

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*: $[\![Seq\ s\ d;\ x \in d]\!] \Longrightarrow \langle x,y\rangle \in s \longleftrightarrow app\ s\ x = y$
by (*auto simp*: *Seq-def hdomain-def app-equality*)

lemma *Exists-iff-app*: $Seq\ s\ d \Longrightarrow x \in d \Longrightarrow (\exists\, y.\ \langle x,\ y\rangle \in s \wedge P\ y) = P\ (app\ s\ x)$
by (*metis Seq-iff-app*)

lemma *Ord-trans2*: $[\![i2 \in i;\ i \in j;\ \ j \in k;\ \ Ord\ k]\!] \Longrightarrow i2 \in k$
by (*metis Ord-trans*)

definition *ord-rec-Seq* :: $hf \Rightarrow (hf \Rightarrow hf) \Rightarrow hf \Rightarrow hf \Rightarrow hf \Rightarrow bool$
where
ord-rec-Seq T G s k y $\longleftrightarrow$
$(Seq\ s\ k \wedge y = G\ (app\ s\ (\bigsqcup k)) \wedge app\ s\ 0 = T \wedge$
$(\forall\, n.\ succ\ n \in k \longrightarrow app\ s\ (succ\ n) = G\ (app\ s\ n)))$

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*)
} then show *?thesis*
by (*metis hf-equalityI hmem-hinsert insf-def nonrestrictD y*)
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*
have *s* = *insf s k* (*G* (*app s* (⨆ *k*)))
by (*smt* (*verit*, *best*) *Ord-pred Seq-succ-D Seq-succ-insf app-insf-Seq k knz ord-rec-Seq-def os succ-iff*)
then show ∃ *s' y*. *ord-rec-Seq T G s' k y* ∧ *z* = *G y* ∧ *s* = *insf s' k y*
by (*metis Ord-Sup-succ-eq Seq-succ-D app-insf-Seq k ord-rec-Seq-def os succ-iff*)
next
assume *ok*: ∃ *s' y*. *ord-rec-Seq T G s' k y* ∧ *z* = *G y* ∧ *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 ⟹ *k* ≠ *0* ⟹ *ord-rec-Seq T G s k y* ⟹ *ord-rec-Seq T G s' k y'* ⟹ *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 ⇒ (hf ⇒ hf) ⇒ (hf ⇒ hf) ⇒ hf ⇒ hf ⇒ bool
  where
   ord-recp T G H x y =
    (if x=0 then y = T
     else
       if Ord(x) then ∃ s. ord-rec-Seq T G s x y
       else y = H x)

lemma ord-recp-functional: ord-recp T G H x y ⟹ ord-recp T G H x y' ⟹ y'
= y
  by (auto simp: ord-recp-def ord-rec-Seq-functional split: if-split-asm)

lemma ord-recp-succ-iff:
  assumes k: Ord k shows ord-recp T G H (succ k) z ⟷ (∃ y. z = G y ∧ ord-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*
    by (*metis* (*no-types*, *lifting*) *ord-rec-def ord-recp-functional ord-recp-succ-iff the-equality*)
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* (λ*z*. *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 ≤ m⟧ ⟹ Vset n ≤ Vset m
 by (metis Ord-linear2 Vset-mono-strict Vset-non order.order-iff-strict
       order-class.order.antisym zero-le)
```

Theorem 4.2(b)

```
lemma Vset-Transset: Ord m ⟹ Transset (Vset m)
 by (induct rule: Ord-induct2) (auto simp: Transset-def)

lemma Ord-sup [simp]: Ord k ⟹ Ord l ⟹ Ord (k ⊔ l)
 by (metis Ord-linear-le le-iff-sup sup-absorb1)

lemma Ord-inf [simp]: Ord k ⟹ Ord l ⟹ Ord (k ⊓ l)
 by (metis Ord-linear-le inf-absorb2 le-iff-inf)
```

Theorem 4.3

```
lemma Vset-universal: ∃ n. Ord n ∧ x ∈ 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 ∈ Vset na Ord nb b ∈ Vset nb
   by blast
 hence b ≤ Vset nb using Vset-Transset [of nb]
   by (auto simp: Transset-def)
 also have ... ≤ Vset (na ⊔ nb) using nab
   by (metis Ord-sup Vset-mono sup-ge2)
 finally have b ◁ a ∈ Vset (succ (na ⊔ 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 ⟹ P k ⟹ ∃ n. Ord n ∧ P n ∧ (∀ m. Ord m ∧ P m ⟶ n ≤ m)
 by (induct k rule: Ord-induct) (metis Ord-linear2)

lemma OrdLeastI: Ord k ⟹ P k ⟹ 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* $\wedge$ *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* $\wedge$ *P n*)
proof −
obtain *n* **where** *Ord n P n* $\forall$ *m. Ord m* $\wedge$ *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* $\wedge$ *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 ∈ b ⟹ rank(a) < rank(b)
  by (metis Vset-rank-lt hsubsetD le-Vset-rank)

lemma rank-mono: x ≤ y ⟹ rank x ≤ 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 ⊔ b) = rank a ⊔ rank b
proof (rule antisym)
  have o: Ord (rank a ⊔ rank b)
    by simp
  have a ≤ Vset (rank a ⊔ rank b) ∧ b ≤ Vset (rank a ⊔ rank b)
    by (metis le-Vset-rank order-trans Vset-mono sup-ge1 sup-ge2 o)
  thus rank (a ⊔ b) ≤ rank a ⊔ rank b
    using Vset-succ-rank-le by auto
next
  show rank a ⊔ rank b ≤ rank (a ⊔ 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|} ≤ 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) ≤ 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 ◁ a) = rank b ⊔ 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 = ⨅ {|*Y* ∈ *HPow*(*Vset* (*rank X*)). *Transset Y* ∧ *X*≤*Y*|}

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

lemma *eclose-minimal*:
assumes *Y*: *Transset Y X*≤*Y* **shows** *eclose X* ≤ *Y*
proof −
have {|*Y* ∈ *HPow*(*Vset* (*rank X*)). *Transset Y* ∧ *X*≤*Y*|} ≠ *0*
by (*simp add*: *eclose-def hempty-iff*) (*metis Ord-rank Vset-Transset le-Vset-rank order-refl*)
moreover have *Transset* (*Y* ⊓ *Vset* (*rank X*))
by (*metis Ord-rank Transset-inf Vset-Transset Y*(*1*))
moreover have *X* ≤ *Y* ⊓ *Vset* (*rank X*)
by (*metis Y*(*2*) *le-Vset-rank le-inf-iff*)
ultimately show *eclose X* ≤ *Y*
apply (*clarsimp 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* ⊔ *b*) = *eclose a* ⊔ *eclose b*
proof (*rule order-antisym*)
show *eclose* (*a* ⊔ *b*) ≤ *eclose a* ⊔ *eclose b*
by (*metis Transset-eclose Transset-sup eclose-minimal le-eclose sup-mono*)
next
show *eclose a* ⊔ *eclose b* ≤ *eclose* (*a* ⊔ *b*)
by (*metis Transset-eclose eclose-minimal le-eclose le-sup-iff*)
qed

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

lemma *eclose-hinsert* [*simp*]: *eclose* (*b* ◁ *a*) = *eclose b* ⊔ (*eclose a* ◁ *a*)
  by (*metis eclose-singleton eclose-sup hinsert-eq-sup*)

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

lemma *fst-in-eclose* [*simp*]: *x* ∈ *eclose* ⟨*x*, *y*⟩
  by (*metis eclose-hinsert hmem-hinsert hpair-def hunion-iff*)

lemma *snd-in-eclose* [*simp*]: *y* ∈ *eclose* ⟨*x*, *y*⟩
  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*: ⋀*x*. (⋀*y*. *y* ∈ *x* ⟹ *P y*) ⟹ *P x* **shows** *P x*
proof −
  have ⋀*y*. *y* ∈ *x* ⟹ *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*
by (*metis hmem-induct hmem-rel-def wfpUNIVI wfp-def wf-trancl*)

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 hmem-rec-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* ∈ *eclose x* ⟹ *ecut f x y* = *f y*
by (*metis ecut-def*)

lemma *RepFun-ecut*: *y* ≤ *z* ⟹ *RepFun y* (*ecut f z*) = *RepFun y f*
by (*meson RepFun-cong ecut-apply hsubsetCE le-eclose*)

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

lemma *hmem-rel-induct* [*case-names step*]:
assumes *ih*: $\bigwedge x.\ (\bigwedge y.\ (y,x) \in \mathit{hmem\text{-}rel} \Longrightarrow P\ y) \Longrightarrow P\ x$ **shows** $P\ x$
proof −
have $\bigwedge y.\ (y,x) \in \mathit{hmem\text{-}rel} \Longrightarrow P\ y$
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*: $x \neq 0 \Longrightarrow \mathit{rank}\ (\bigsqcup x) < \mathit{rank}\ x$
proof (*induction x rule*: *hf-induct*)
case *0*
then show *?case* **by** *auto*
next
case (*hinsert x y*)
then show *?case*
apply (*clarsimp simp*: *Ord-lt-succ-iff-le less-supI2*)
by (*metis HUnion-hempty Ord-lt-succ-iff-le Ord-rank hunion-hempty-right less-supI1 less-supI2 rank-sup sup.cobounded2*)
qed

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
next :: hf ⇒ 'a ⇒ hf ⇒ bool

inductive *reaches* :: [*'a fsm*, *hf*, *'a list*, *hf*] ⇒ *bool*
where
Nil: *st ∈ states fsm ⟹ reaches fsm st [] st*
| *Cons*: ⟦*next 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 ≡ ∃ *st st'*. *reaches fsm st xs st'* ∧ *st* ∈ *init fsm* ∧ *st'* ∈ *final fsm*

definition *regular* :: *'a list set* ⇒ *bool* **where**
regular S ≡ ∃*fsm*. *S* = {*xs*. *accepts fsm xs*}

definition *Null* **where**
Null = (|*states* = *0*, *init* = *0*, *final* = *0*, *next* = λ*st x st'*. *False*|)

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

abbreviation *NullStr* **where**
NullStr ≡ (|*states* = *1*, *init* = *1*, *final* = *1*, *next* = λ*st x st'*. *False*|)

theorem *regular-emptystr*: *regular* {[]}
proof −
have ⋀*x*::*'a list*. *reaches NullStr 0 x 0* ⟹ *x* = []
using *reaches.simps* **by** *fastforce*
then show *?thesis*
unfolding *regular-def accepts-def*
by (*rule-tac x* = *NullStr* **in** *exI*) *auto*
qed

abbreviation *SingStr* **where**
SingStr a ≡ (|*states* = {|*0*, *1*|}, *init* = {|*0*|}, *final* = {|*1*|}, *next* = λ*st x st'*. *st*=*0* ∧ *x*=*a* ∧ *st'*=*1*|)

theorem *regular-singstr*: *regular* {[*a*]}
proof −
have ⋀*x*::*'a list*. *reaches* (*SingStr a*) *0 x 1* ⟹ *x* = [*a*]
by (*smt* (*verit*, *best*) *one-neq-zero reaches.simps select-convs*(*4*))
then show *?thesis*
unfolding *regular-def accepts-def*
by (*rule-tac x* = *SingStr a* **in** *exI*) *auto*
qed

definition *Reverse* **where**
Reverse fsm = (|*states* = *states fsm*, *init* = *final fsm*, *final* = *init fsm*,
next = λ*st x st'*. *next 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'* ⟷ *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* ⟷ *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*)

lemma *accepts-Reverse*: *rev* ‘ {*xs*. *accepts fsm xs*} = {*xs*. *accepts* (*Reverse fsm*) *xs*}
by (*fastforce simp*: *accepts-def image-iff*)

theorem *regular-rev*: *regular S* ⟹ *regular* (*rev* ‘ *S*)
by (*metis accepts-Reverse regular-def*)

definition *Times* **where**
Times fsm1 fsm2 = (|*states* = *states fsm1* ∗ *states fsm2*,
init = *init fsm1* ∗ *init fsm2*,
final = *final fsm1* ∗ *final fsm2*,
next = λ*st x st′*. (∃ *st1 st2 st1′ st2′*. *st* = ⟨*st1*,*st2*⟩ ∧ *st′* = ⟨*st1′*,*st2′*⟩ ∧
next fsm1 st1 x st1′ ∧ *next 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 *next-Times*: *next* (*Times fsm1 fsm2*) ⟨*st1*,*st2*⟩ *x st′* ⟷
(∃ *st1′ st2′*. *st′* = ⟨*st1′*,*st2′*⟩ ∧ *next fsm1 st1 x st1′* ∧ *next 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′*
proof (*induction xs arbitrary*: *st1 st2 st1′ st2′*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a xs*)
then show *?case*
by (*force simp add*: *next-Times Times-def reaches.Cons*)
qed

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* ∩ *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*,
next = λ*st x st'*. (∃ *st1 st1'*. *st* = *Inl st1* ∧ *st'* = *Inl st1'* ∧ *next fsm1 st1 x st1'*) ∨
(∃ *st2 st2'*. *st* = *Inr st2* ∧ *st'* = *Inr st2'* ∧ *next 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 *next-Plus1*: *next* (*Plus fsm1 fsm2*) (*Inl st1*) *x st'* ⟷ (∃ *st1'*. *st'* = *Inl st1'* ∧ *next fsm1 st1 x st1'*)
by (*simp add*: *Plus-def*)

lemma *next-Plus2*: *next* (*Plus fsm1 fsm2*) (*Inr st2*) *x st'* ⟷ (∃ *st2'*. *st'* = *Inr st2'* ∧ *next fsm2 st2 x st2'*)
by (*simp add*: *Plus-def*)

lemma *reaches-Plus-iff1* [*simp*]:
reaches (*Plus fsm1 fsm2*) (*Inl st1*) *xs st'* ⟷
(∃ *st1'*. *st'* = *Inl st1'* ∧ *reaches fsm1 st1 xs st1'*)
proof (*induction xs arbitrary*: *st1*)
case *Nil*
then show *?case* **by** *auto*
next

case (*Cons a xs*)
then show *?case*
by (*force simp add*: *next-Plus1 reaches.Cons*)
qed

lemma *reaches-Plus-iff2* [*simp*]:
reaches (*Plus fsm1 fsm2*) (*Inr st2*) *xs st'* ⟷
(∃ *st2'*. *st'* = *Inr st2'* ∧ *reaches fsm2 st2 xs st2'*)
proof (*induction xs arbitrary*: *st2*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a xs*)
then show *?case* **by** (*force simp add*: *next-Plus2 reaches.Cons*)
qed

lemma *reaches-Plus-iff* [*simp*]:
reaches (*Plus fsm1 fsm2*) *st xs st'* ⟷
(∃ *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'*)
proof (*induction xs arbitrary*: *st st'*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a xs*)
then show *?case*
by (*smt* (*verit*) *Plus-def list.simps*(*3*) *reaches.simps reaches-Plus-iff1 reaches-Plus-iff2 select-convs*(*4*))
qed

lemma *accepts-Plus-iff* [*simp*]:
accepts (*Plus fsm1 fsm2*) *xs* ⟷ *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
theory *Finitary*
imports *Ordinal*

begin

class *finitary* =
  fixes *hf-of* :: *'a* ⇒ *hf*
  assumes *inj*: *inj hf-of*
begin
  lemma *hf-of-eq-iff* [*simp*]: *hf-of x* = *hf-of y* ⟷ *x*=*y*
    using *inj* **by** (*auto simp*: *inj-on-def*)
end

instantiation *unit* :: *finitary*
begin
  definition *hf-of-unit-def*: *hf-of* (*u*::*unit*) ≡ *0*
  instance
    by *intro-classes* (*auto simp*: *inj-on-def hf-of-unit-def*)
end

instantiation *bool* :: *finitary*
begin
  definition *hf-of-bool-def*: *hf-of b* ≡ *if b then 1 else 0*
  instance
    by *intro-classes* (*auto simp*: *inj-on-def hf-of-bool-def*)
end

instantiation *nat* :: *finitary*
begin
  definition *hf-of-nat-def*: *hf-of* ≡ *ord-of*
  instance
    by *intro-classes* (*auto simp*: *inj-on-def hf-of-nat-def*)
end

instantiation *int* :: *finitary*
begin
  definition *hf-of-int-def*:
    hf-of i ≡ *if i*≥(*0*::*int*) *then* ⟨*0*, *hf-of* (*nat i*)⟩ *else* ⟨*1*, *hf-of* (*nat* (−*i*))⟩
  instance
    by *intro-classes* (*auto simp*: *inj-on-def hf-of-int-def*)
end

Strings are char lists, and are not considered separately.

instantiation *char* :: *finitary*
begin
  definition *hf-of-char-def*:
    hf-of x ≡ *hf-of* (*of-char x* :: *nat*)
  instance
    by *standard* (*auto simp*: *inj-on-def hf-of-char-def*)
end

instantiation *prod* :: (*finitary*,*finitary*) *finitary*

```
begin
  definition hf-of-prod-def:
    hf-of ≡ λ(x,y). ⟨hf-of x, hf-of y⟩
  instance
    by intro-classes (auto simp: inj-on-def hf-of-prod-def)
end

instantiation sum :: (finitary,finitary) finitary
begin
  definition hf-of-sum-def:
    hf-of ≡ case-sum (HF.Inl o hf-of) (HF.Inr o hf-of)
  instance
    by intro-classes (auto simp: inj-on-def hf-of-sum-def split: sum.split-asm)
end

instantiation option :: (finitary) finitary
begin
  definition hf-of-option-def:
    hf-of ≡ case-option 0 (λx. {|hf-of x|})
  instance
    by intro-classes (auto simp: inj-on-def hf-of-option-def split: option.split-asm)
end

instantiation list :: (finitary) finitary
begin
  primrec hf-of-list where
    hf-of-list Nil = 0
  | hf-of-list (x#xs) = ⟨hf-of x, hf-of-list xs⟩

lemma [simp]: fixes x :: 'a list shows hf-of x = hf-of y ⟹ x = y
  proof (induction x arbitrary: y)
    case Nil
    then show ?case by (cases y) auto
  next
    case (Cons a x)
    then show ?case by (cases y) auto
  qed

instance
  by intro-classes (auto simp: inj-on-def)
end

end
```

Chapter 5

Addition, Sequences and their Concatenation

theory *OrdArith* **imports** *Rank*
begin

5.1 Generalised Addition — Also for Ordinals

Source: Laurence Kirby, Addition and multiplication of sets Math. Log. Quart. 53, No. 1, 52-65 (2007) / DOI 10.1002/malq.200610026 http://faculty.baruch.cuny.edu/lkirby/mlqarticlejan2007.pdf

definition
hadd :: *hf* ⇒ *hf* ⇒ *hf* (**infixl** ‹@+› *65*) **where**
hadd x ≡ *hmemrec* (λ*f z*. *x* ⊔ *RepFun z f*)

lemma *hadd*: *x* @+ *y* = *x* ⊔ *RepFun y* (λ*z*. *x* @+ *z*)
by (*metis def-hmemrec RepFun-ecut hadd-def order-refl*)

lemma *hmem-hadd-E*:
assumes *l*: *l* ∈ *x* @+ *y*
obtains *l* ∈ *x* | *z* **where** *z* ∈ *y l* = *x* @+ *z*
using *l* **by** (*auto simp*: *hadd* [*of x y*])

lemma *hadd-0-right* [*simp*]: *x* @+ *0* = *x*
by (*subst hadd*) *simp*

lemma *hadd-hinsert-right*: *x* @+ *hinsert y z* = *hinsert* (*x* @+ *y*) (*x* @+ *z*)
by (*metis hadd hunion-hinsert-right RepFun-hinsert*)

lemma *hadd-succ-right* [*simp*]: *x* @+ *succ y* = *succ* (*x* @+ *y*)
by (*metis hadd-hinsert-right succ-def*)

lemma *not-add-less-right*: ¬ (*x* @+ *y* < *x*)
proof (*induction y*)

case (*2 y1 y2*)
then show *?case*
using *hadd less-supI1 order-less-le* **by** *blast*
qed *auto*

lemma *not-add-mem-right*: ¬ (*x* @+ *y* ∈ *x*)
by (*metis hadd hmem-not-refl hunion-iff*)

lemma *hadd-0-left* [*simp*]: *0* @+ *x* = *x*
by (*induct x*) (*auto simp*: *hadd-hinsert-right*)

lemma *hadd-succ-left* [*simp*]: *Ord y* ⟹ *succ x* @+ *y* = *succ* (*x* @+ *y*)
by (*induct y rule*: *Ord-induct2*) *auto*

lemma *hadd-assoc*: (*x* @+ *y*) @+ *z* = *x* @+ (*y* @+ *z*)
by (*induct z*) (*auto simp*: *hadd-hinsert-right*)

lemma *RepFun-hadd-disjoint*: *x* ⊓ *RepFun y* ((@+) *x*) = *0*
by (*metis hf-equalityI RepFun-iff hinter-iff not-add-mem-right hmem-hempty*)

5.1.1 Cancellation laws for addition

lemma *Rep-le-Cancel*: *x* ⊔ *RepFun y* ((@+) *x*) ≤ *x* ⊔ *RepFun z* ((@+) *x*)
⟹ *RepFun y* ((@+) *x*) ≤ *RepFun z* ((@+) *x*)
by (*auto simp add*: *not-add-mem-right*)

lemma *hadd-cancel-right* [*simp*]: *x* @+ *y* = *x* @+ *z* ⟷ *y*=*z*
proof (*induct y arbitrary*: *z rule*: *hmem-induct*)
case (*step y z*) **show** *?case*
proof *auto*
assume *eq*: *x* @+ *y* = *x* @+ *z*
hence *RepFun y* ((@+) *x*) = *RepFun z* ((@+) *x*)
by (*metis hadd Rep-le-Cancel order-antisym order-refl*)
thus *y* = *z*
by (*metis hf-equalityI RepFun-iff step*)
qed
qed

lemma *RepFun-hadd-cancel*: *RepFun y* (λ*z*. *x* @+ *z*) = *RepFun z* (λ*z*. *x* @+ *z*)
⟷ *y*=*z*
by (*metis hadd hadd-cancel-right*)

lemma *hadd-hmem-cancel* [*simp*]: *x* @+ *y* ∈ *x* @+ *z* ⟷ *y* ∈ *z*
by (*metis RepFun-iff hadd hadd-cancel-right hunion-iff not-add-mem-right*)

lemma *ord-of-add*: *ord-of* (*i*+*j*) = *ord-of i* @+ *ord-of j*
by (*induct j*) *auto*

lemma *Ord-hadd*: *Ord x* ⟹ *Ord y* ⟹ *Ord* (*x* @+ *y*)

by (*induct x rule*: *Ord-induct2*) *auto*

lemma *hmem-self-hadd* [*simp*]: *k1* ∈ *k1* @+ *k2* ⟷ *0* ∈ *k2*
by (*metis hadd-0-right hadd-hmem-cancel*)

lemma *hadd-commute*: *Ord x* ⟹ *Ord y* ⟹ *x* @+ *y* = *y* @+ *x*
by (*induct x rule*: *Ord-induct2*) *auto*

lemma *hadd-cancel-left* [*simp*]: *Ord x* ⟹ *y* @+ *x* = *z* @+ *x* ⟷ *y=z*
by (*induct x rule*: *Ord-induct2*) *auto*

5.1.2 The predecessor function

definition *pred* :: *hf* ⇒ *hf*
where *pred x* ≡ (*THE y. succ y* = *x* ∨ *x=0* ∧ *y=0*)

lemma *pred-succ* [*simp*]: *pred* (*succ x*) = *x*
by (*simp add*: *pred-def*)

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

lemma *succ-pred* [*simp*]: *Ord x* ⟹ *x* ≠ *0* ⟹ *succ* (*pred x*) = *x*
by (*metis Ord-cases pred-succ*)

lemma *pred-mem* [*simp*]: *Ord x* ⟹ *x* ≠ *0* ⟹ *pred x* ∈ *x*
by (*metis succ-iff succ-pred*)

lemma *Ord-pred* [*simp*]: *Ord x* ⟹ *Ord* (*pred x*)
by (*metis Ord-in-Ord pred-0 pred-mem*)

lemma *hadd-pred-right*: *Ord y* ⟹ *y* ≠ *0* ⟹ *x* @+ *pred y* = *pred* (*x* @+ *y*)
by (*metis hadd-succ-right pred-succ succ-pred*)

lemma *Ord-pred-HUnion*: *Ord*(*k*) ⟹ *pred k* = ⨆ *k*
by (*metis HUnion-hempty Ordinal.Ord-pred pred-0 pred-succ*)

5.2 A Concatentation Operation for Sequences

definition *shift* :: *hf* ⇒ *hf* ⇒ *hf*
where *shift f delta* = {‖*v* . *u* ∈ *f*, ∃ *n y. u* = ⟨*n*, *y*⟩ ∧ *v* = ⟨*delta* @+ *n*, *y*⟩‖}

lemma *shiftD*: *x* ∈ *shift f delta* ⟹ ∃ *u. u* ∈ *f* ∧ *x* = ⟨*delta* @+ *hfst u*, *hsnd u*⟩
by (*auto simp*: *shift-def hsplit-def*)

lemma *hmem-shift-iff*: ⟨*m*, *y*⟩ ∈ *shift f delta* ⟷ (∃ *n. m* = *delta* @+ *n* ∧ ⟨*n*, *y*⟩ ∈ *f*)
by (*auto simp*: *shift-def hrelation-def is-hpair-def*)

lemma *hmem-shift-add-iff* [*simp*]: $\langle delta\ @+\ n,\ y\rangle \in shift\ f\ delta \longleftrightarrow \langle n,\ y\rangle \in f$
by (*metis hadd-cancel-right hmem-shift-iff*)

lemma *hrelation-shift* [*simp*]: *hrelation* (*shift f delta*)
by (*auto simp*: *shift-def hrelation-def hsplit-def*)

lemma *app-shift* [*simp*]: *app* (*shift f k*) (*k @+ j*) = *app f j*
by (*simp add*: *app-def*)

lemma *hfunction-shift-iff* [*simp*]: *hfunction* (*shift f delta*) = *hfunction f*
by (*auto simp*: *hfunction-def hmem-shift-iff*)

lemma *hdomain-shift-add*: *hdomain* (*shift f delta*) = $\{\!|delta\ @+\ n\ .\ n \in hdomain\ f|\!\}$
by (*rule hf-equalityI*) (*force simp add*: *hdomain-def hmem-shift-iff*)

lemma *hdomain-shift-disjoint*: *delta* $\sqcap$ *hdomain* (*shift f delta*) = *0*
by (*simp add*: *RepFun-hadd-disjoint hdomain-shift-add*)

definition *seq-append* :: *hf* $\Rightarrow$ *hf* $\Rightarrow$ *hf* $\Rightarrow$ *hf*
where *seq-append k f g* $\equiv$ *hrestrict f k* $\sqcup$ *shift g k*

lemma *hrelation-seq-append* [*simp*]: *hrelation* (*seq-append k f g*)
by (*simp add*: *seq-append-def*)

lemma *Seq-append*:
assumes *Seq s1 k1 Seq s2 k2*
shows *Seq* (*seq-append k1 s1 s2*) (*k1 @+ k2*)
proof −
have *hfunction* (*hrestrict s1 k1* $\sqcup$ *shift s2 k1*)
using *assms*
by (*simp add*: *Ordinal.Seq-def hdomain-shift-disjoint hfunction-hunion hfunction-restr inf.absorb2*)
moreover
have $\bigwedge x.\ [\![x \in k1\ @+\ k2;\ x \notin hdomain\ (shift\ s2\ k1)]\!] \Longrightarrow x \in hdomain\ s1 \wedge x \in k1$
by (*metis Ordinal.Seq-def RepFun-iff assms hdomain-shift-add hmem-hadd-E hsubsetCE*)
ultimately show *?thesis*
by (*auto simp*: *Seq-def seq-append-def*)
qed

lemma *app-hunion1*: $x \notin hdomain\ g \Longrightarrow app\ (f \sqcup g)\ x = app\ f\ x$
by (*auto simp*: *app-def*) (*metis hdomainI*)

lemma *app-hunion2*: $x \notin hdomain\ f \Longrightarrow app\ (f \sqcup g)\ x = app\ g\ x$
by (*auto simp*: *app-def*) (*metis hdomainI*)

lemma *Seq-append-app1*: *Seq s k* $\Longrightarrow$ $l \in k \Longrightarrow app\ (seq\text{-}append\ k\ s\ s')\ l = app\ s$

l
by (*metis app-hrestrict app-hunion1 hdomain-shift-disjoint hemptyE hinter-iff seq-append-def*)

lemma *Seq-append-app2*: *Seq s1 k1* $\Longrightarrow$ *Seq s2 k2* $\Longrightarrow$ *l* = *k1* @+ *j* $\Longrightarrow$ *app* (*seq-append k1 s1 s2*) *l* = *app s2 j*
by (*metis seq-append-def app-hunion2 app-shift hdomain-restr hinter-iff not-add-mem-right*)

5.3 Nonempty sequences indexed by ordinals

definition *OrdDom* **where**
OrdDom r $\equiv \forall\, x\ y.\ \langle x,y\rangle \in r \longrightarrow$ *Ord x*

lemma *OrdDom-insf*: $[\![$*OrdDom s*; *Ord k*$]\!] \Longrightarrow$ *OrdDom* (*insf s* (*succ k*) *y*)
by (*auto simp*: *insf-def OrdDom-def*)

lemma *OrdDom-hunion* [*simp*]: *OrdDom* (*s1* $\sqcup$ *s2*) $\longleftrightarrow$ *OrdDom s1* $\wedge$ *OrdDom s2*
by (*auto simp*: *OrdDom-def*)

lemma *OrdDom-hrestrict*: *OrdDom s* $\Longrightarrow$ *OrdDom* (*hrestrict s A*)
by (*auto simp*: *OrdDom-def*)

lemma *OrdDom-shift*: $[\![$*OrdDom s*; *Ord k*$]\!] \Longrightarrow$ *OrdDom* (*shift s k*)
by (*auto simp*: *OrdDom-def shift-def Ord-hadd*)

A sequence of positive length ending with *y*

definition *LstSeq* :: *hf* $\Rightarrow$ *hf* $\Rightarrow$ *hf* $\Rightarrow$ *bool*
where *LstSeq s k y* $\equiv$ *Seq s* (*succ k*) $\wedge$ *Ord k* $\wedge$ $\langle k,y\rangle \in s \wedge$ *OrdDom s*

lemma *LstSeq-imp-Seq-succ*: *LstSeq s k y* $\Longrightarrow$ *Seq s* (*succ k*)
by (*metis LstSeq-def*)

lemma *LstSeq-imp-Seq-same*: *LstSeq s k y* $\Longrightarrow$ *Seq s k*
by (*metis LstSeq-imp-Seq-succ Seq-succ-D*)

lemma *LstSeq-imp-Ord*: *LstSeq s k y* $\Longrightarrow$ *Ord k*
by (*metis LstSeq-def*)

lemma *LstSeq-trunc*: *LstSeq s k y* $\Longrightarrow$ *l* $\in$ *k* $\Longrightarrow$ *LstSeq s l* (*app s l*)
by (*meson LstSeq-def Ord-in-Ord Seq-Ord-D Seq-iff-app Seq-succ-iff*)

lemma *LstSeq-insf*: *LstSeq s k z* $\Longrightarrow$ *LstSeq* (*insf s* (*succ k*) *y*) (*succ k*) *y*
using *LstSeq-def OrdDom-insf Seq-insf insf-def* **by** *force*

lemma *app-insf-LstSeq*: *LstSeq s k z* $\Longrightarrow$ *app* (*insf s* (*succ k*) *y*) (*succ k*) = *y*
by (*metis LstSeq-imp-Seq-succ app-insf-Seq*)

lemma *app-insf2-LstSeq*: *LstSeq s k z* $\Longrightarrow$ *k′* $\neq$ *succ k* $\Longrightarrow$ *app* (*insf s* (*succ k*) *y*) *k′* = *app s k′*
by (*metis LstSeq-imp-Seq-succ app-insf2-Seq*)

lemma *app-insf-LstSeq-if*: *LstSeq s k z* $\Longrightarrow$ *app* (*insf s* (*succ k*) *y*) *k′* = (*if k′* = *succ k then y else app s k′*)
by (*metis app-insf2-LstSeq app-insf-LstSeq*)

lemma *LstSeq-append-app1*:
LstSeq s k y $\Longrightarrow$ *l* $\in$ *succ k* $\Longrightarrow$ *app* (*seq-append* (*succ k*) *s s′*) *l* = *app s l*
by (*metis LstSeq-imp-Seq-succ Seq-append-app1*)

lemma *LstSeq-append-app2*:
⟦*LstSeq s1 k1 y1*; *LstSeq s2 k2 y2*; *l* = *succ k1* @+ *j*⟧
$\Longrightarrow$ *app* (*seq-append* (*succ k1*) *s1 s2*) *l* = *app s2 j*
by (*metis LstSeq-imp-Seq-succ Seq-append-app2*)

lemma *Seq-append-pair*:
⟦*Seq s1 k1*; *Seq s2* (*succ n*); ⟨*n*, *y*⟩ $\in$ *s2*; *Ord n*⟧ $\Longrightarrow$ ⟨*k1* @+ *n*, *y*⟩ $\in$ (*seq-append k1 s1 s2*)
by (*metis hmem-shift-add-iff hunion-iff seq-append-def*)

lemma *Seq-append-OrdDom*: ⟦*Ord k*; *OrdDom s1*; *OrdDom s2*⟧ $\Longrightarrow$ *OrdDom* (*seq-append k s1 s2*)
by (*auto simp*: *seq-append-def OrdDom-hrestrict OrdDom-shift*)

lemma *LstSeq-append*:
⟦*LstSeq s1 k1 y1*; *LstSeq s2 k2 y2*⟧ $\Longrightarrow$ *LstSeq* (*seq-append* (*succ k1*) *s1 s2*) (*succ* (*k1* @+ *k2*)) *y2*
using *LstSeq-def Ord-hadd Seq-append Seq-append-OrdDom Seq-append-pair* **by** *fastforce*

lemma *LstSeq-app* [*simp*]: *LstSeq s k y* $\Longrightarrow$ *app s k* = *y*
by (*metis LstSeq-def Seq-imp-eq-app*)

5.3.1 Sequence-building operators

definition *Builds* :: (*hf* $\Rightarrow$ *bool*) $\Rightarrow$ (*hf* $\Rightarrow$ *hf* $\Rightarrow$ *hf* $\Rightarrow$ *bool*) $\Rightarrow$ *hf* $\Rightarrow$ *hf* $\Rightarrow$ *bool*
where *Builds B C s l* $\equiv$ *B* (*app s l*) $\vee$ ($\exists$ *m* $\in$ *l*. $\exists$ *n* $\in$ *l*. *C* (*app s l*) (*app s m*) (*app s n*))

definition *BuildSeq* :: (*hf* $\Rightarrow$ *bool*) $\Rightarrow$ (*hf* $\Rightarrow$ *hf* $\Rightarrow$ *hf* $\Rightarrow$ *bool*) $\Rightarrow$ *hf* $\Rightarrow$ *hf* $\Rightarrow$ *hf* $\Rightarrow$ *bool*
where *BuildSeq B C s k y* $\equiv$ *LstSeq s k y* $\wedge$ ($\forall$ *l* $\in$ *succ k*. *Builds B C s l*)

lemma *BuildSeqI*: *LstSeq s k y* $\Longrightarrow$ ($\bigwedge$*l*. *l* $\in$ *succ k* $\Longrightarrow$ *Builds B C s l*) $\Longrightarrow$ *BuildSeq B C s k y*
by (*simp add*: *BuildSeq-def*)

lemma *BuildSeq-imp-LstSeq*: *BuildSeq B C s k y* $\Longrightarrow$ *LstSeq s k y*
by (*metis BuildSeq-def*)

lemma *BuildSeq-imp-Seq*: *BuildSeq B C s k y* $\Longrightarrow$ *Seq s* (*succ k*)
by (*metis LstSeq-imp-Seq-succ BuildSeq-imp-LstSeq*)

lemma *BuildSeq-conj-distrib*:
BuildSeq (λx. *B x* $\wedge$ *P x*) ($\lambda x\ y\ z$. *C x y z* $\wedge$ *P x*) *s k y* $\longleftrightarrow$
BuildSeq B C s k y $\wedge$ ($\forall\, l \in$ *succ k*. *P* (*app s l*))
by (*auto simp*: *BuildSeq-def Builds-def*)

lemma *BuildSeq-mono*:
assumes *y*: *BuildSeq B C s k y*
and *B*: $\bigwedge x$. *B x* $\Longrightarrow$ *B′ x* **and** *C*: $\bigwedge x\ y\ z$. *C x y z* $\Longrightarrow$ *C′ x y z*
shows *BuildSeq B′ C′ s k y*
using *y*
by (*auto simp*: *BuildSeq-def Builds-def intro*!: *B C*)

lemma *BuildSeq-trunc*:
assumes *b*: *BuildSeq B C s k y*
and *l*: *l* $\in$ *k*
shows *BuildSeq B C s l* (*app s l*)
by (*smt* (*verit*) *BuildSeqI BuildSeq-def LstSeq-def LstSeq-trunc Ord-trans b hballE l succ-iff*)

5.3.2 Showing that Sequences can be Constructed

lemma *Builds-insf*: *Builds B C s l* $\Longrightarrow$ *LstSeq s k z* $\Longrightarrow$ *l* $\in$ *succ k* $\Longrightarrow$ *Builds B C* (*insf s* (*succ k*) *y*) *l*
by (*auto simp*: *HBall-def hmem-not-refl Builds-def app-insf-LstSeq-if simp del*: *succ-iff*)
(*metis hmem-not-sym*)

lemma *BuildSeq-insf*:
assumes *b*: *BuildSeq B C s k z*
and *m*: *m* $\in$ *succ k*
and *n*: *n* $\in$ *succ k*
and *y*: *B y* $\vee$ *C y* (*app s m*) (*app s n*)
shows *BuildSeq B C* (*insf s* (*succ k*) *y*) (*succ k*) *y*
proof (*rule BuildSeqI*)
show *LstSeq* (*insf s* (*succ k*) *y*) (*succ k*) *y*
by (*metis BuildSeq-imp-LstSeq LstSeq-insf b*)
next
fix *l*
assume *l*: *l* $\in$ *succ* (*succ k*)
thus *Builds B C* (*insf s* (*succ k*) *y*) *l*
proof
assume *l*: *l* = *succ k*
have *B* (*app* (*insf s l y*) *l*) $\vee$ *C* (*app* (*insf s l y*) *l*) (*app* (*insf s l y*) *m*) (*app*

```
(insf s l y) n)
      by (metis BuildSeq-imp-Seq app-insf-Seq-if b hmem-not-refl l m n y)
    thus Builds B C (insf s (succ k) y) l using m n
      by (auto simp: Builds-def l)
  next
    assume l: l ∈ succ k
    thus Builds B C (insf s (succ k) y) l using b l
      by (metis hballE Builds-insf BuildSeq-def)
  qed
qed

lemma BuildSeq-insf1:
  assumes b: BuildSeq B C s k z
      and y: B y
  shows BuildSeq B C (insf s (succ k) y) (succ k) y
by (metis BuildSeq-insf b succ-iff y)

lemma BuildSeq-insf2:
  assumes b: BuildSeq B C s k z
      and m: m ∈ k
      and n: n ∈ k
      and y: C y (app s m) (app s n)
  shows BuildSeq B C (insf s (succ k) y) (succ k) y
  by (metis BuildSeq-insf b m n succ-iff y)

lemma BuildSeq-append:
  assumes s1: BuildSeq B C s1 k1 y1 and s2: BuildSeq B C s2 k2 y2
  shows BuildSeq B C (seq-append (succ k1) s1 s2) (succ (k1 @+ k2)) y2
proof (rule BuildSeqI)
  show LstSeq (seq-append (succ k1) s1 s2) (succ (k1 @+ k2)) y2
    using assms
    by (metis BuildSeq-imp-LstSeq LstSeq-append)
next
  fix l
  have s1L: LstSeq s1 k1 y1
   and s1BC: ⋀l. l ∈ succ k1 ⟹ Builds B C s1 l
   and s2L: LstSeq s2 k2 y2
   and s2BC: ⋀l. l ∈ succ k2 ⟹ Builds B C s2 l
    using s1 s2 by (auto simp: BuildSeq-def)
  assume l: l ∈ succ (succ (k1 @+ k2))
  hence  l ∈ succ k1 @+ succ k2
    by (metis LstSeq-imp-Ord hadd-succ-left hadd-succ-right s2L)
  thus Builds B C (seq-append (succ k1) s1 s2) l
  proof (rule hmem-hadd-E)
    assume l1: l ∈ succ k1
   hence B (app s1 l) ∨ (∃m∈l. ∃n∈l. C (app s1 l) (app s1 m) (app s1 n)) using
s1BC
      by (simp add: Builds-def)
    thus ?thesis
```

```
    proof
      assume B (app s1 l)
      thus ?thesis
        by (metis Builds-def LstSeq-append-app1 l1 s1L)
    next
      assume ∃ m∈l. ∃ n∈l. C (app s1 l) (app s1 m) (app s1 n)
      then obtain m n where mn: m ∈ l n ∈ l and C: C (app s1 l) (app s1 m)
(app s1 n)
        by blast
      moreover have m ∈ succ k1 n ∈ succ k1
        by (metis LstSeq-def Ord-trans l1 mn s1L succ-iff)+
      ultimately have C (app (seq-append (succ k1) s1 s2) l)
                        (app (seq-append (succ k1) s1 s2) m)
                        (app (seq-append (succ k1) s1 s2) n)
        using s1L l1
        by (simp add: LstSeq-append-app1)
      thus Builds B C (seq-append (succ k1) s1 s2) l using mn
        by (auto simp: Builds-def)
    qed
  next
    fix z
    assume z: z ∈ succ k2 and l2: l = succ k1 @+ z
     hence B (app s2 z) ∨ (∃ m∈z. ∃ n∈z. C (app s2 z) (app s2 m) (app s2 n))
using s2BC
      by (simp add: Builds-def)
    thus ?thesis
    proof
      assume B (app s2 z)
      thus ?thesis
        by (metis Builds-def LstSeq-append-app2 l2 s1L s2L)
    next
      assume ∃ m∈z. ∃ n∈z. C (app s2 z) (app s2 m) (app s2 n)
      then obtain m n where mn: m ∈ z n ∈ z and C: C (app s2 z) (app s2 m)
(app s2 n)
        by blast
      also have m ∈ succ k2 n ∈ succ k2 using mn
        by (metis LstSeq-def Ord-trans z s2L succ-iff)+
      ultimately have C (app (seq-append (succ k1) s1 s2) l)
                        (app (seq-append (succ k1) s1 s2) (succ k1 @+ m))
                        (app (seq-append (succ k1) s1 s2) (succ k1 @+ n))
        using s1L s2L l2 z
        by (simp add: LstSeq-append-app2)
      thus Builds B C (seq-append (succ k1) s1 s2) l using mn l2
        by (auto simp: Builds-def HBall-def)
    qed
  qed
qed

lemma BuildSeq-combine:
```

```
 assumes b1: BuildSeq B C s1 k1 y1 and b2: BuildSeq B C s2 k2 y2
     and y: C y y1 y2
 shows BuildSeq B C (insf (seq-append (succ k1) s1 s2) (succ (succ (k1 @+ k2)))
y) (succ (succ (k1 @+ k2))) y
proof −
 have k2: Ord k2  using b2
  by (auto simp: BuildSeq-def LstSeq-def)
 show ?thesis
 proof (rule BuildSeq-insf [where m=k1 and n=succ(k1@+k2)])
  show BuildSeq B C (seq-append (succ k1) s1 s2) (succ (k1 @+ k2)) y2
   by (rule BuildSeq-append [OF b1 b2])
 next
  show k1 ∈ succ (succ (k1 @+ k2)) using k2
   by (metis hadd-0-right hmem-0-Ord hmem-self-hadd succ-iff)
 next
  show succ (k1 @+ k2) ∈ succ (succ (k1 @+ k2))
   by (metis succ-iff)
 next
  have [simp]: app (seq-append (succ k1) s1 s2) k1 = y1
   by (metis b1 BuildSeq-imp-LstSeq LstSeq-app LstSeq-append-app1 succ-iff)
  have [simp]: app (seq-append (succ k1) s1 s2) (succ (k1 @+ k2)) = y2
      by (metis b1 b2 k2 BuildSeq-imp-LstSeq LstSeq-app LstSeq-append-app2
hadd-succ-left)
  show B y ∨
       C y (app (seq-append (succ k1) s1 s2) k1)
          (app (seq-append (succ k1) s1 s2) (succ (k1 @+ k2)))
   using y by simp
 qed
qed

lemma LstSeq-1: LstSeq {|⟨0, y⟩|} 0 y
 by (auto simp: LstSeq-def One-hf-eq-succ Seq-ins OrdDom-def)

lemma BuildSeq-1: B y ⟹ BuildSeq B C {|⟨0, y⟩|} 0 y
 by (auto simp: BuildSeq-def Builds-def LstSeq-1)

lemma BuildSeq-exI: B t ⟹ ∃ s k. BuildSeq B C s k t
 by (metis BuildSeq-1)
```

5.3.3 Proving Properties of Given Sequences

```
lemma BuildSeq-succ-E:
   assumes s: BuildSeq B C s k y
     obtains B y |  m n where m ∈ k n ∈ k C y (app s m) (app s n)
proof −
 have Bs: Builds B C s k and apps: app s k = y using s
  by (auto simp: BuildSeq-def)
 hence B y ∨ (∃ m ∈ k. ∃ n ∈ k. C y (app s m) (app s n))
  by (metis Builds-def apps Bs)
```

thus *?thesis* **using** *that*
  by *auto*
qed

lemma *BuildSeq-induct* [*consumes 1*, *case-names B C*]:
  assumes *major*: *BuildSeq B C s k a*
    and *B*: $\bigwedge x.\ B\ x \Longrightarrow P\ x$
    and *C*: $\bigwedge x\ y\ z.\ C\ x\ y\ z \Longrightarrow P\ y \Longrightarrow P\ z \Longrightarrow P\ x$
  shows *P a*
proof −
  have *Ord k* **using** *assms*
    by (*auto simp*: *BuildSeq-def LstSeq-def*)
  hence $\bigwedge a\ s.$ *BuildSeq B C s k a* $\Longrightarrow$ *P a*
    by (*induction k rule*: *Ord-induct*) (*metis BuildSeq-trunc BuildSeq-succ-E B C*)
  thus *?thesis*
    by (*metis major*)
qed

definition *BuildSeq2* :: [[*hf*,*hf*] ⇒ *bool*, [*hf*,*hf*,*hf*,*hf*,*hf*,*hf*] ⇒ *bool*, *hf*, *hf*, *hf*, *hf*] ⇒ *bool*
  where *BuildSeq2 B C s k y y′* ≡
        BuildSeq (λ*p*. ∃ *x x′*. *p* = ⟨*x*,*x′*⟩ ∧ *B x x′*)
                (λ*p q r*. ∃ *x x′ y y′ z z′*. *p* = ⟨*x*,*x′*⟩ ∧ *q* = ⟨*y*,*y′*⟩ ∧ *r* = ⟨*z*,*z′*⟩ ∧ *C x x′ y y′ z z′*)
                s k ⟨*y*,*y′*⟩

lemma *BuildSeq2-combine*:
  assumes *b1*: *BuildSeq2 B C s1 k1 y1 y1′* **and** *b2*: *BuildSeq2 B C s2 k2 y2 y2′*
    and *y*: *C y y′ y1 y1′ y2 y2′*
  shows *BuildSeq2 B C* (*insf* (*seq-append* (*succ k1*) *s1 s2*) (*succ* (*succ* (*k1* @+ *k2*))) ⟨*y*, *y′*⟩)
                (*succ* (*succ* (*k1* @+ *k2*))) *y y′*
  using *BuildSeq2-def BuildSeq-combine b1 b2 y* **by** *force*

lemma *BuildSeq2-1*: *B y y′* ⟹ *BuildSeq2 B C* {|⟨*0*, *y*, *y′*⟩|} *0 y y′*
  by (*auto simp*: *BuildSeq2-def BuildSeq-1*)

lemma *BuildSeq2-exI*: *B t t′* ⟹ ∃ *s k*. *BuildSeq2 B C s k t t′*
  by (*metis BuildSeq2-1*)

lemma *BuildSeq2-induct* [*consumes 1*, *case-names B C*]:
  assumes *BuildSeq2 B C s k a a′*
    and *B*: $\bigwedge x\ x'.\ B\ x\ x' \Longrightarrow P\ x\ x'$
    and *C*: $\bigwedge x\ x'\ y\ y'\ z\ z'.\ C\ x\ x'\ y\ y'\ z\ z' \Longrightarrow P\ y\ y' \Longrightarrow P\ z\ z' \Longrightarrow P\ x\ x'$
  shows *P a a′*
  using *assms BuildSeq-induct* [**where** *P* = λ⟨*x*,*x′*⟩. *P x x′*]
  by (*smt* (*verit*, *del-insts*) *BuildSeq2-def hsplit*)

definition *BuildSeq3*

:: [[*hf*,*hf*,*hf*] ⇒ *bool*, [*hf*,*hf*,*hf*,*hf*,*hf*,*hf*,*hf*,*hf*,*hf*] ⇒ *bool*, *hf*, *hf*, *hf*, *hf*, *hf*] ⇒ *bool*
where *BuildSeq3 B C s k y y′ y″* ≡
BuildSeq (λ*p*. ∃ *x x′ x″*. *p* = ⟨*x*,*x′*,*x″*⟩ ∧ *B x x′ x″*)
(λ*p q r*. ∃ *x x′ x″ y y′ y″ z z′ z″*.
p = ⟨*x*,*x′*,*x″*⟩ ∧ *q* = ⟨*y*,*y′*,*y″*⟩ ∧ *r* = ⟨*z*,*z′*,*z″*⟩ ∧
C x x′ x″ y y′ y″ z z′ z″)
s k ⟨*y*,*y′*,*y″*⟩

lemma *BuildSeq3-combine*:
assumes *b1*: *BuildSeq3 B C s1 k1 y1 y1′ y1″* **and** *b2*: *BuildSeq3 B C s2 k2 y2 y2′ y2″*
and *y*: *C y y′ y″ y1 y1′ y1″ y2 y2′ y2″*
shows *BuildSeq3 B C* (*insf* (*seq-append* (*succ k1*) *s1 s2*) (*succ* (*succ* (*k1* @+ *k2*))) ⟨*y*, *y′*, *y″*⟩)
(*succ* (*succ* (*k1* @+ *k2*))) *y y′ y″*
using *assms*
unfolding *BuildSeq3-def* **by** (*blast intro*: *BuildSeq-combine*)

lemma *BuildSeq3-1*: *B y y′ y″* ⟹ *BuildSeq3 B C* {|⟨*0*, *y*, *y′*, *y″*⟩|} *0 y y′ y″*
by (*auto simp*: *BuildSeq3-def BuildSeq-1*)

lemma *BuildSeq3-exI*: *B t t′ t″* ⟹ ∃ *s k*. *BuildSeq3 B C s k t t′ t″*
by (*metis BuildSeq3-1*)

lemma *BuildSeq3-induct* [*consumes 1*, *case-names B C*]:
assumes *BuildSeq3 B C s k a a′ a″*
and *B*: ⋀*x x′ x″*. *B x x′ x″* ⟹ *P x x′ x″*
and *C*: ⋀*x x′ x″ y y′ y″ z z′ z″*. *C x x′ x″ y y′ y″ z z′ z″* ⟹ *P y y′ y″* ⟹ *P z z′ z″* ⟹ *P x x′ x″*
shows *P a a′ a″*
using *assms BuildSeq-induct* [**where** *P* = λ⟨*x*,*x′*,*x″*⟩. *P x x′ x″*]
by (*smt* (*verit*, *del-insts*) *BuildSeq3-def hsplit*)

5.4 A Unique Predecessor for every non-empty set

lemma *Rep-hf-0* [*simp*]: *Rep-hf 0* = *0*
by (*metis Abs-hf-inverse HF.HF-def UNIV-I Zero-hf-def image-empty set-encode-empty*)

lemma *hmem-imp-less*: *x* ∈ *y* ⟹ *Rep-hf x* < *Rep-hf y*
unfolding *hmem-def hfset-def image-iff*
apply (*clarsimp simp*: *hmem-def hfset-def set-decode-def Abs-hf-inverse*)
apply (*metis div-less even-zero le-less-trans less-exp not-less*)
done

lemma *hsubset-imp-le*:
assumes *x* ≤ *y* **shows** *Rep-hf x* ≤ *Rep-hf y*
proof −
have ⋀*u v*. ⟦∀ *x*. *x* ∈ *Abs-hf* ‘ *set-decode* (*Rep-hf* (*Abs-hf u*)) ⟶

$x \in$ *Abs-hf* ‘ *set-decode* (*Rep-hf* (*Abs-hf v*))⟧
$\Longrightarrow u \leq v$
by (*metis Abs-hf-inverse UNIV-I imageE image-eqI subsetI subset-decode-imp-le*)
then show *?thesis*
by (*metis Rep-hf-inverse assms hfset-def hmem-def hsubsetCE*)
qed

lemma *diff-hmem-imp-less*: **assumes** $x \in y$ **shows** *Rep-hf* $(y - \{\!|x|\!\}) <$ *Rep-hf y*
proof −
have *Rep-hf* $(y - \{\!|x|\!\}) \leq$ *Rep-hf y*
by (*metis hdiff-iff hsubsetI hsubset-imp-le*)
moreover
have *Rep-hf* $(y - \{\!|x|\!\}) \neq$ *Rep-hf y* **using** *assms*
by (*metis Rep-hf-inject hdiff-iff hinsert-iff*)
ultimately show *?thesis*
by (*metis le-neq-implies-less*)
qed

definition *least* :: *hf* ⇒ *hf*
where *least a* ≡ (*THE x*. $x \in a \wedge (\forall y.\ y \in a \longrightarrow$ *Rep-hf x* ≤ *Rep-hf y*))

lemma *least-equality*:
assumes $x \in a$ **and** $\bigwedge y.\ y \in a \Longrightarrow$ *Rep-hf x* ≤ *Rep-hf y*
shows *least a* = *x*
unfolding *least-def*
using *Rep-hf-inject assms order-antisym-conv* **by** *blast*

lemma *leastI2-order*:
assumes $x \in a$
and $\bigwedge y.\ y \in a \Longrightarrow$ *Rep-hf x* ≤ *Rep-hf y*
and $\bigwedge z.\ z \in a \Longrightarrow \forall y.\ y \in a \longrightarrow$ *Rep-hf z* ≤ *Rep-hf y* $\Longrightarrow Q\ z$
shows *Q* (*least a*)
by (*metis assms least-equality*)

lemma *nonempty-imp-ex-least*: $a \neq 0 \Longrightarrow \exists x.\ x \in a \wedge (\forall y.\ y \in a \longrightarrow$ *Rep-hf x* ≤ *Rep-hf y*)
proof (*induction a rule*: *hf-induct*)
case *0* **thus** *?case* **by** *simp*
next
case (*hinsert u v*)
show *?case*
proof (*cases v=0*)
case *True* **thus** *?thesis*
by (*rule-tac x=u* **in** *exI*, *simp*)
next
case *False*
thus *?thesis*
by (*metis order.trans hinsert.IH*(*2*) *hmem-hinsert linorder-le-cases*)
qed

qed

lemma *least-hmem*: $a \neq 0 \Longrightarrow least\ a \in a$
  by (*metis least-equality nonempty-imp-ex-least*)

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.