

# The Hereditarily Finite Sets

Lawrence C. Paulson

September 18, 2020

### **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

<b>1</b>	<b>The Hereditarily Finite Sets</b>	<b>3</b>
1.1	Basic Definitions and Lemmas . . . . .	3
1.2	Verifying the Axioms of HF . . . . .	5
1.3	Ordered Pairs, from <i>ZF/ZF.thy</i> . . . . .	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
<b>2</b>	<b>Ordinals, Sequences and Ordinal Recursion</b>	<b>26</b>
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

<b>3</b>	<b>V-Sets, Epsilon Closure, Ranks</b>	<b>38</b>
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
<b>4</b>	<b>An Application: Finite Automata</b>	<b>46</b>
<b>5</b>	<b>Addition, Sequences and their Concatenation</b>	<b>53</b>
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 Concatenation 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 . . . . .	63
5.4	A Unique Predecessor for every non-empty set . . . . .	65

# Chapter 1

## The Hereditarily Finite Sets

```
theory HF
imports HOL-Library.Nat-Bijection
abbrevs <: = ∈
       and ~<: = ∉
begin
```

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

### 1.1 Basic Definitions and Lemmas

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

```
definition hfset :: hf ⇒ hf set
  where hfset a = Abs-hf ' set-decode (Rep-hf a)
```

```
definition HF :: hf set ⇒ hf
  where HF A = Abs-hf (set-encode (Rep-hf ' A))
```

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

```
definition hmem :: hf ⇒ hf ⇒ bool (infixl ∈ 50)
  where hmem a b ⇔ a ∈ hfset b
```

```
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*]:  $Abs-hf\ 0 = 0$   
**by** (*simp add: HF-def Zero-hf-def*)

HF Set enumerations

**abbreviation** *inserthf* ::  $hf \Rightarrow hf \Rightarrow hf$  (**infixl** < 60)  
**where**  $y < x \equiv hinsert\ x\ y$

**syntax** (*ASCII*)  
*-HFInset* ::  $args \Rightarrow hf$  ( $\{|(-)|\}$ )  
**syntax**  
*-HFInset* ::  $args \Rightarrow hf$  ( $\{\!-\!\}$ )  
**translations**  
 $\{x, y\} \rightleftharpoons \{y\} < x$   
 $\{x\} \rightleftharpoons 0 < x$

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

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

**lemma** *hfset-HF* [*simp*]:  $finite\ A \Longrightarrow hfset\ (HF\ A) = A$   
**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*]:  
 $hmem\ a\ (c < b) \longleftrightarrow a = b \vee a \in c$   
**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*]:  
 $\llbracket finite\ F; F = \{\} \rrbracket \Longrightarrow P; \bigwedge A\ x. \llbracket F = insert\ x\ A; x \notin A; finite\ A \rrbracket \Longrightarrow P \Longrightarrow P$   
**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*:  
 $a \notin b \implies \text{Rep-hf } (\text{hinsert } a \ b) = 2 \wedge (\text{Rep-hf } a) + \text{Rep-hf } b$   
**unfolding** *hinsert-def HF-def hfset-def*  
**apply** (*simp add: image-image Abs-hf-inverse Rep-hf-inverse*)  
**apply** (*subst set-encode-insert, simp*)  
**apply** (*clarsimp simp add: hmem-def hfset-def image-def*  
*Rep-hf-inject [symmetric] Abs-hf-inverse, simp*)  
**done**

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

## 1.2 Verifying the Axioms of HF

HF1

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

HF2

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

HF induction

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

HF3

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

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

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

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

**lemma** *hmem-HF-iff* [*simp*]:  $x \in HF\ A \longleftrightarrow x \in A \wedge finite\ A$   
**apply** (*cases finite A, auto*)  
**apply** (*simp add: hmem-def*)  
**apply** (*simp add: hmem-def*)  
**apply** (*metis HF-def Rep-hf-inject Abs-hf-0 finite-imageD hempty-iff inj-onI set-encode-inj*)  
**done**

### 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*)<sup>+</sup>

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

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

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

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

Ordered Pairs, from ZF/ZF.thy

**nonterminal** *hfs*

**syntax** (*ASCII*)

-*Tuple* ::  $[hf, hfs] \Rightarrow hf$  ( $\langle(-,/ -)\rangle$ )

-*hpattern* ::  $[pttrn, patterns] \Rightarrow pttrn$  ( $\langle(-,/ -)\rangle$ )

**syntax**

::  $hf \Rightarrow hfs$  (-)

-*Enum* ::  $[hf, hfs] \Rightarrow hfs$  ( $(-,/ -)$ )

-*Tuple* ::  $[hf, hfs] \Rightarrow hf$  ( $\langle\langle(-,/ -)\rangle\rangle$ )

-*hpattern* ::  $[pttrn, patterns] \Rightarrow pttrn$  ( $\langle\langle(-,/ -)\rangle\rangle$ )



## translations

$\langle x, y, z \rangle \rightleftharpoons \langle x, \langle y, z \rangle \rangle$   
 $\langle x, y \rangle \rightleftharpoons \text{CONST } \text{hpair } x \ y$   
 $\langle x, y, z \rangle \rightleftharpoons \langle x, \langle y, z \rangle \rangle$   
 $\lambda \langle x, y, zs \rangle. b \rightleftharpoons \text{CONST } \text{hsplit}(\lambda x \langle y, zs \rangle. b)$   
 $\lambda \langle x, y \rangle. b \rightleftharpoons \text{CONST } \text{hsplit}(\lambda x \ y. b)$

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

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

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

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

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

**lemma** *hsplit* [*simp*]:  $\text{hsplit } c \ \langle a, b \rangle = c \ a \ b$   
**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-empty*)

**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-empty)
next
  case (hinsert a b) thus ?case by (metis hmem-hinsert)
qed

```

```

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

```

```

syntax (ASCII)
  -HCollect :: idt  $\Rightarrow$  hf  $\Rightarrow$  bool  $\Rightarrow$  hf ((1  $\{\}$ - <:/ -./ - $\}$ ))
syntax
  -HCollect :: idt  $\Rightarrow$  hf  $\Rightarrow$  bool  $\Rightarrow$  hf ((1  $\{\}$ -  $\in$ / -./ - $\}$ ))
translations
   $\{\{x \in A. P\}\} \Rightarrow$  CONST HCollect ( $\lambda x. P$ ) A

```

```

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

```

```

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

```

```

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

```

```

lemma HCollect-empty [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.  $\forall u. u \in z \longleftrightarrow u \in a \vee u \in b$ )
  instance ..
end

```

```

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

```

```

lemma hunion-iff [iff]: hmem x (a  $\sqcup$  b)  $\longleftrightarrow x \in a \vee x \in b$ 

```

**apply** (*insert binary-union [of a b], clarify*)  
**apply** (*simp add: sup-hf-def*)  
**apply** (*rule theI2*)  
**apply** (*auto simp: hf-ext*)  
**done**

**definition**  $HUnion :: hf \Rightarrow hf$  ( $\sqcup$ -[900] 900)  
**where**  $HUnion A = (THE z. \forall u. u \in z \longleftrightarrow (\exists y. y \in A \wedge u \in y))$

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

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

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

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

#### 1.4.4 Definition 1.8, Intersections

**instantiation**  $hf :: inf$   
**begin**  
**definition**  $inf a b = \{x \in a. x \in b\}$   
**instance ..**  
**end**

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

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

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

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

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

**lemma**  $HInter$ -hinsert [simp]:  $A \neq 0 \Longrightarrow \sqcap (A \triangleleft a) = a \sqcap \sqcap A$

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:  
 $(A \triangleleft x) - B = (if\ x \in B\ then\ A - B\ else\ (A - B) \triangleleft x)$   
 by auto

## 1.5 Replacement

Theorem 1.9 (Replacement Scheme).

**lemma** replacement:

$(\forall u\ v\ v'.\ u \in x \longrightarrow R\ u\ v \longrightarrow R\ u\ v' \longrightarrow v'=v) \implies \exists z.\ \forall v.\ v \in z \longleftrightarrow (\exists u.\ u \in x \wedge R\ u\ v)$

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

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

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

**syntax** (ASCII)

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

**syntax**

-HReplace :: [pttrn, pttrn, hf, bool]  $\Rightarrow hf$  ((1{- ./ - \in -, -|}))  
-HRepFun :: [hf, pttrn, hf]  $\Rightarrow hf$  ((1{- ./ - \in -|}) [51,0,51])  
-HINTER :: [pttrn, hf, hf]  $\Rightarrow hf$  ((3{- \in -./ -) 10)  
-HUNION :: [pttrn, hf, hf]  $\Rightarrow hf$  ((3{- \in -./ -) 10)

**translations**

{y. x \in A, Q}  $\Leftrightarrow$  CONST Replace A ( $\lambda x y. Q$ )  
{b. x \in A}  $\Leftrightarrow$  CONST RepFun A ( $\lambda x. b$ )  
 $\prod x \in A. B$   $\Leftrightarrow$  CONST HInter(CONST RepFun A ( $\lambda x. B$ ))  
 $\sqcup x \in A. B$   $\Leftrightarrow$  CONST HUnion(CONST RepFun A ( $\lambda x. B$ ))

**lemma** PrimReplace-iff:

**assumes** sv:  $\forall u v v'. u \in A \longrightarrow R u v \longrightarrow R u v' \longrightarrow v'=v$   
**shows**  $v \in (PrimReplace A R) \longleftrightarrow (\exists u. u \in A \wedge R u v)$

**apply** (insert replacement [OF sv], clarify)

**apply** (simp add: PrimReplace-def)

**apply** (rule theI2)

**apply** (auto simp: hf-ext)

**done**

**lemma** Replace-iff [iff]:

$v \in Replace A R \longleftrightarrow (\exists u. u \in A \wedge R u v \wedge (\forall y. R u y \longrightarrow y=v))$

**apply** (simp add: Replace-def)

**apply** (subst PrimReplace-iff, auto)

**done**

**lemma** Replace-0 [simp]:  $Replace 0 R = 0$

**by** blast

**lemma** Replace-hunion [simp]:  $Replace (A \sqcup B) R = Replace A R \sqcup Replace B R$

**by** blast

**lemma** Replace-cong [cong]:

$\llbracket A=B; \bigwedge x y. x \in B \Longrightarrow P x y \longleftrightarrow Q x y \rrbracket \Longrightarrow Replace A P = Replace B Q$   
**by** (simp add: hf-ext cong: conj-cong)

**lemma** RepFun-iff [iff]:  $v \in (RepFun A f) \longleftrightarrow (\exists u. u \in A \wedge v = f u)$

**by** (auto simp: RepFun-def)

**lemma** *RepFun-cong* [*cong*]:

$\llbracket A=B; \bigwedge x. x \in B \implies f(x)=g(x) \rrbracket \implies \text{RepFun } A \ f = \text{RepFun } B \ g$   
**by** (*simp add: RepFun-def*)

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

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

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

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

**lemma** *HF-HUnion*:  $\llbracket \text{finite } A; \bigwedge x. x \in A \implies \text{finite } (B \ x) \rrbracket \implies \text{HF } (\bigcup x \in A. B \ x) = (\bigcup x \in \text{HF } A. \text{HF } (B \ x))$   
**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**  $A < B \longleftrightarrow A \leq B \wedge A \neq (B::\text{hf})$

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

**end**

### 1.6.1 Rules for subsets

**lemma** *hsubsetI* [*intro!*]:

$(\bigwedge x. x \in A \implies x \in B) \implies A \leq B$

**by** (*simp add: less-eq-hf-def*)

Classical elimination rule

**lemma** *hsubsetCE* [*elim*]:  $\llbracket A \leq B; c \notin A \implies P; c \in B \implies P \rrbracket \implies P$

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

Rule in Modus Ponens style

**lemma** *hsubsetD* [*elim*]:  $\llbracket A \leq B; c \in A \rrbracket \implies c \in B$

**by** (*simp add: less-eq-hf-def*)

Sometimes useful with premises in this order

**lemma** *rev-hsubsetD*:  $\llbracket c \in A; A \leq B \rrbracket \implies c \in B$   
**by** *blast*

**lemma** *contra-hsubsetD*:  $\llbracket A \leq B; c \notin B \rrbracket \implies c \notin A$   
**by** *blast*

**lemma** *rev-contra-hsubsetD*:  $\llbracket c \notin B; A \leq B \rrbracket \implies c \notin A$   
**by** *blast*

**lemma** *hf-equalityE*:  
**fixes**  $A :: hf$  **shows**  $A = B \implies (A \leq B \implies B \leq A \implies P) \implies P$   
**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*]:  $u \leq 0 \iff u = (0 :: hf)$   
**by** (*metis hempty-iff less-eq-hf-def*)

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

**lemma** *less-eq-insert2-iff*:  
 $z \leq (\text{hinsert } x \ y) \iff z \leq y \vee (\exists u. \text{hinsert } x \ u = z \wedge x \notin u \wedge u \leq y)$   
**proof** (*cases*  $x \in z$ )  
**case** *True*

**hence**  $u: \text{hinsert } x (z - \{x\}) = z$  **by** *auto*  
**show** *?thesis*  
**proof**  
    **assume**  $z \leq (\text{hinsert } x y)$   
    **thus**  $z \leq y \vee (\exists u. \text{hinsert } x u = z \wedge x \notin u \wedge u \leq y)$   
    **by** (*simp add: less-eq-hf-def*) (*metis u hdiff-iff hmem-hinsert*)  
**next**  
    **assume**  $z \leq y \vee (\exists u. \text{hinsert } x u = z \wedge x \notin u \wedge u \leq y)$   
    **thus**  $z \leq (\text{hinsert } x y)$   
    **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:*  $\text{hinsert } x A \sqcup B = \text{hinsert } x (A \sqcup B)$   
**by** *blast*

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

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

**lemma** *hinter-hinsert-right:*  $B \sqcap \text{hinsert } x A = (\text{if } x \in B \text{ then } \text{hinsert } x (B \sqcap A) \text{ else } B \sqcap A)$   
**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:  $x \notin x$ 
  using foundation [of {x}]
  by (metis hinter-iff hmem-hempty hmem-hinsert)

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma zero-notin-hpair:  $0 \notin \langle x, y \rangle$ 
  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:  $hfset\ (b \triangleleft a) = insert\ a\ (hfset\ b)$ 
  by (metis finite-insert hinsert-def HF.finite-hfset hfset-HF)

```

```

lemma hfset-hdiff:  $hfset\ (x - y) = hfset\ x - hfset\ y$ 

```

```

proof (induct x arbitrary: y rule: hf-induct)
  case 0 thus ?case
  by simp

```

```

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  $\Rightarrow$  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  $\in$  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  $\sqcup$  y) + hcard (x  $\sqcap$  y) = hcard x + hcard y
  apply (induct x arbitrary: y rule: hf-induct)
  apply (auto simp: hcard-hinsert-if hunion-hinsert-left hinter-hinsert-left)
  done

lemma hcard-hdiff1-less: x  $\in$  z  $\implies$  hcard (z - {x}) < hcard z
  by (simp add: hcard-def hfset-hdiff hfset-hinsert)
  (metis card-Diff1-less finite-hfset hmem-def)

```

### 1.7.3 Powerset Operator

Theorem 1.11 (Existence of the power set).

```

lemma powerset:  $\exists z. \forall u. u \in z \longleftrightarrow u \leq x$ 
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:  $\forall u. u \in Pb \longleftrightarrow u \leq b$ 
    by auto
  obtain RPb where RPb:  $\forall v. v \in RPb \longleftrightarrow (\exists u. u \in Pb \wedge v = \text{hinsert } a \ u)$ 
    using replacement-fun ..
  thus ?case using Pb binary-union [of Pb RPb]
    apply (simp add: less-eq-insert2-iff, clarify)
    apply (rule-tac x=z in exI)
    apply (metis hinsert.hyps less-eq-hf-def)
  done
qed

```

```

definition HPow :: hf  $\Rightarrow$  hf
  where HPow x = (THE z.  $\forall u. u \in z \longleftrightarrow u \leq x$ )

```

```

lemma HPow-iff [iff]: u  $\in$  HPow x  $\longleftrightarrow$  u  $\leq$  x
  apply (insert powerset [of x], clarify)

```

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

**lemma** *HPow-mono*:  $x \leq y \implies \text{HPow } x \leq \text{HPow } y$   
**by** (*metis HPow-iff less-eq-hf-def order-trans*)

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

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

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

## 1.8 Bounded Quantifiers

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

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

**syntax** (*ASCII*)

$\text{-HBall} \quad :: \text{pttrn} \Rightarrow hf \Rightarrow bool \Rightarrow bool \quad ((\exists ALL \text{-<:./ -}) [0, 0, 10] 10)$   
 $\text{-HBex} \quad :: \text{pttrn} \Rightarrow hf \Rightarrow bool \Rightarrow bool \quad ((\exists EX \text{-<:./ -}) [0, 0, 10] 10)$   
 $\text{-HBex1} \quad :: \text{pttrn} \Rightarrow hf \Rightarrow bool \Rightarrow bool \quad ((\exists EX! \text{-<:./ -}) [0, 0, 10] 10)$

**syntax**

$\text{-HBall} \quad :: \text{pttrn} \Rightarrow hf \Rightarrow bool \Rightarrow bool \quad ((\exists \forall \text{-}\in\text{./ -}) [0, 0, 10] 10)$   
 $\text{-HBex} \quad :: \text{pttrn} \Rightarrow hf \Rightarrow bool \Rightarrow bool \quad ((\exists \exists \text{-}\in\text{./ -}) [0, 0, 10] 10)$   
 $\text{-HBex1} \quad :: \text{pttrn} \Rightarrow hf \Rightarrow bool \Rightarrow bool \quad ((\exists \exists! \text{-}\in\text{./ -}) [0, 0, 10] 10)$

**translations**

$\forall x \in A. P \rightleftharpoons \text{CONST HBall } A (\lambda x. P)$   
 $\exists x \in A. P \rightleftharpoons \text{CONST HBex } A (\lambda x. P)$   
 $\exists! x \in A. P \rightarrow \exists! x. x \in A \wedge P$

**lemma** *hball-cong* [*cong*]:

$\llbracket A = A'; \bigwedge x. x \in A' \implies P(x) \longleftrightarrow P'(x) \rrbracket \implies (\forall x \in A. P(x)) \longleftrightarrow (\forall x \in A'. P'(x))$

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

**lemma** *hballI* [*intro!*]:  $(\bigwedge x. x \in A \implies P x) \implies \forall x \in A. P x$

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

**lemma** *hbspec* [*dest?*]:  $\forall x \in A. P x \implies x \in A \implies P x$

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

**lemma** *hballE* [*elim*]:  $\forall x \in A. P x \implies (P x \implies Q) \implies (x \notin A \implies Q) \implies Q$   
**by** (*unfold HBall-def*) *blast*

**lemma** *hbex-cong* [*cong*]:  
 $\llbracket A=A'; \bigwedge x. x \in A' \implies P(x) \longleftrightarrow P'(x) \rrbracket \implies (\exists x \in A. P(x)) \longleftrightarrow (\exists x \in A'. P'(x))$   
**by** (*simp add: HBex-def cong: conj-cong*)

**lemma** *hbexI* [*intro*]:  $P x \implies x \in A \implies \exists x \in A. P x$   
**by** (*unfold HBex-def*) *blast*

**lemma** *rev-hbexI* [*intro?*]:  $x \in A \implies P x \implies \exists x \in A. P x$   
**by** (*unfold HBex-def*) *blast*

**lemma** *bexCI*:  $(\forall x \in A. \neg P x \implies P a) \implies a \in A \implies \exists x \in A. P x$   
**by** (*unfold HBex-def*) *blast*

**lemma** *hbexE* [*elim!*]:  $\exists x \in A. P x \implies (\bigwedge x. x \in A \implies P x \implies Q) \implies Q$   
**by** (*unfold HBex-def*) *blast*

**lemma** *hball-triv* [*simp*]:  $(\forall x \in A. P) = ((\exists x. x \in A) \longrightarrow P)$   
— Trivial rewrite rule.  
**by** (*simp add: HBall-def*)

**lemma** *hbex-triv* [*simp*]:  $(\exists x \in A. P) = ((\exists x. x \in A) \wedge P)$   
— Dual form for existentials.  
**by** (*simp add: HBex-def*)

**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*]:

$$\begin{aligned} \bigwedge A P Q. (\forall x \in A. P x \vee Q) &\longleftrightarrow ((\forall x \in A. P x) \vee Q) \\ \bigwedge A P Q. (\forall x \in A. P \vee Q x) &\longleftrightarrow (P \vee (\forall x \in A. Q x)) \\ \bigwedge A P Q. (\forall x \in A. P \longrightarrow Q x) &\longleftrightarrow (P \longrightarrow (\forall x \in A. Q x)) \\ \bigwedge A P Q. (\forall x \in A. P x \longrightarrow Q) &\longleftrightarrow ((\exists x \in A. P x) \longrightarrow Q) \\ \bigwedge P. (\forall x \in \emptyset. P x) &\longleftrightarrow \text{True} \\ \bigwedge a B P. (\forall x \in B \triangleleft a. P x) &\longleftrightarrow (P a \wedge (\forall x \in B. P x)) \\ \bigwedge P Q. (\forall x \in \text{HCollect } Q A. P x) &\longleftrightarrow (\forall x \in A. Q x \longrightarrow P x) \\ \bigwedge A P. (\neg (\forall x \in A. P x)) &\longleftrightarrow (\exists x \in A. \neg P x) \end{aligned}$$

**by** *auto*

**lemma** *hb-ex-simps* [*simp, no-atp*]:

$$\begin{aligned} \bigwedge A P Q. (\exists x \in A. P x \wedge Q) &\longleftrightarrow ((\exists x \in A. P x) \wedge Q) \\ \bigwedge A P Q. (\exists x \in A. P \wedge Q x) &\longleftrightarrow (P \wedge (\exists x \in A. Q x)) \\ \bigwedge P. (\exists x \in \emptyset. P x) &\longleftrightarrow \text{False} \\ \bigwedge a B P. (\exists x \in B \triangleleft a. P x) &\longleftrightarrow (P a \vee (\exists x \in B. P x)) \\ \bigwedge P Q. (\exists x \in \text{HCollect } Q A. P x) &\longleftrightarrow (\exists x \in A. Q x \wedge P x) \\ \bigwedge A P. (\neg (\exists x \in A. P x)) &\longleftrightarrow (\forall x \in A. \neg P x) \end{aligned}$$

**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* ::  $hf \Rightarrow bool$

**where** *is-hpair*  $z = (\exists x y. z = \langle x, y \rangle)$

**definition** *hconverse* ::  $hf \Rightarrow hf$

**where** *hconverse*( $r$ ) =  $\{z. w \in r, \exists x y. w = \langle x, y \rangle \wedge z = \langle y, x \rangle\}$

**definition** *hdomain* ::  $hf \Rightarrow hf$

**where** *hdomain*( $r$ ) =  $\{x. w \in r, \exists y. w = \langle x, y \rangle\}$

**definition** *hrange* ::  $hf \Rightarrow hf$

**where** *hrange*( $r$ ) = *hdomain*(*hconverse*( $r$ ))

**definition** *hrelation* ::  $hf \Rightarrow bool$

**where** *hrelation*( $r$ ) =  $(\forall z. z \in r \longrightarrow \text{is-hpair } z)$

**definition** *hrestrict* ::  $hf \Rightarrow hf \Rightarrow hf$

— Restrict the relation  $r$  to the domain  $A$

**where** *hrestrict*  $r A = \{z \in r. \exists x \in A. \exists y. z = \langle x, y \rangle\}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**lemma** *hfunction-0* [*simp*]: *hfunction* 0

**by** (*force simp add: hfunction-def*)

**lemma** *hfunction-ins*: *hfunction* *f*  $\implies x \notin \text{hdomain } f \implies \text{hfunction } (f \triangleleft \langle x, y \rangle)$

**by** (*auto simp: hfunction-def hdomain-def*)

**lemma** *hdomainI*:  $\langle x, y \rangle \in f \implies x \in \text{hdomain } f$

**by** (*auto simp: hdomain-def*)

**lemma** *hfunction-hunion*:  $\text{hdomain } f \sqcap \text{hdomain } g = 0$

$\implies \text{hfunction } (f \sqcup g) \longleftrightarrow \text{hfunction } f \wedge \text{hfunction } g$

**by** (*auto simp: hfunction-def*) (*metis hdomainI hinter-iff hmem-hempty*)<sup>+</sup>

**lemma** *app-hrestrict* [*simp*]:  $x \in A \implies \text{app } (\text{hrestrict } f A) x = \text{app } f x$

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

**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*  $\prod$  - $\in$ ./ -) 10)

-*SUM* :: [*pttrn*, *hf*, *hf*]  $\Rightarrow hf$  ((*3*  $\sum$  - $\in$ ./ -) 10)

-*lam* :: [*pttrn*, *hf*, *hf*]  $\Rightarrow hf$  ((*3*  $\lambda$  - $\in$ ./ -) 10)

**translations**

$\prod x \in A. B \Leftrightarrow \text{CONST } \text{HPi } A (\lambda x. B)$

$\sum x \in A. B \Leftrightarrow \text{CONST } \text{HSigma } A (\lambda x. B)$

$\lambda x \in A. f \Leftrightarrow \text{CONST } \text{HLambda } A (\lambda x. f)$

### 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*:  $\llbracket a \in A; b \in B(a) \rrbracket \implies b \in (\bigsqcup x \in A. B(x))$

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

**lemma** *HINT-I*:  $\llbracket \bigwedge x. x \in A \implies b \in B(x); A \neq 0 \rrbracket \implies b \in (\prod x \in A. B(x))$   
by (*simp add: HINT-iff*)

**lemma** *HINT-E*:  $\llbracket b \in (\prod x \in A. B(x)); a \in A \rrbracket \implies b \in B(a)$   
by (*auto simp: HINT-iff*)

### 1.10.2 Generalized Cartesian product

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

**lemma** *HSigmaI* [*intro!*]:  $\llbracket a \in A; b \in B(a) \rrbracket \implies \langle a, b \rangle \in \text{HSigma } A \ B$   
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**  $c \in \text{HSigma } A \ B$   
**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**  $\langle a, b \rangle \in \text{HSigma } A \ B$  **obtains**  $a \in A$  **and**  $b \in B(a)$   
using *assms* by *auto*

**lemma** *HSigma-empty1* [*simp*]:  $\text{HSigma } 0 \ B = 0$   
by *blast*

**instantiation** *hf* :: *times*

**begin**

**definition**  $A * B = \text{HSigma } A \ (\lambda x. B)$

**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!*]:  $\llbracket a \in A; b \in B \rrbracket \implies \langle a, b \rangle \in A * B$   
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*:  $A * B = 0 \iff A = 0 \vee B = (0::hf)$   
**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* ::  $hf \Rightarrow hf$  **where**  
 $Inl(a) \equiv \langle 0, a \rangle$

**definition** *Inr* ::  $hf \Rightarrow hf$  **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* ::  $hf \Rightarrow bool$

**where** *is-hsum*  $z = (\exists x. z = Inl\ x \vee z = Inr\ x)$

**definition** *sum-case* ::  $(hf \Rightarrow 'a) \Rightarrow (hf \Rightarrow 'a) \Rightarrow hf \Rightarrow 'a$

**where**

*sum-case*  $f\ g\ a \equiv$

*THE*  $z. (\forall x. a = Inl\ x \longrightarrow z = f\ x) \wedge (\forall y. a = Inr\ y \longrightarrow z = g\ y) \wedge (\neg is-hsum\ a \longrightarrow z = undefined)$

**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*]:  $\neg is-hsum\ a \Longrightarrow sum-case\ f\ g\ a = undefined$

**by** (*simp* *add: sum-case-def is-hsum-def*)

**lemma** *is-hsum-cases*:  $(\exists x. z = Inl\ x \vee z = Inr\ x) \vee \neg is-hsum\ z$

**by** (*auto simp: is-hsum-def*)

**lemma** *sum-case-split*:

$P\ (sum-case\ f\ g\ a) \longleftrightarrow (\forall x. a = Inl\ x \longrightarrow P(f\ x)) \wedge (\forall y. a = Inr\ y \longrightarrow P(g\ y)) \wedge (\neg is-hsum\ a \longrightarrow P\ undefined)$

**by** (*cases is-hsum a*) (*auto simp: is-hsum-def*)

**lemma** *sum-case-split-asm*:

$P\ (sum-case\ f\ g\ a) \longleftrightarrow \neg ((\exists x. a = Inl\ x \wedge \neg P(f\ x)) \vee (\exists y. a = Inr\ y \wedge \neg P(g\ y)) \vee (\neg is-hsum\ a \wedge \neg P\ undefined))$

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

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

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

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

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

**definition**

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

#### 2.1.2 Definition 2.2 (Successor).

**definition**

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

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

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

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

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

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

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

**lemma** *hbex-succ* [*simp*]:  $(\exists x \in succ\ k. P\ x) \longleftrightarrow P\ k \vee (\exists x \in k. P\ x)$   
**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*]:  $x \in 1 \longleftrightarrow x = 0$   
**by** (*metis One-hf-eq-succ hmem-empty succ-iff*)

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

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

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

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

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

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

**lemma** *hmem-succ-self* [*simp*]:  $k \in succ\ k$

by (metis succ-iff)

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

Theorem 2.3.

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

**lemma** *Ord-succ*:  $\text{Ord}(k) \implies \text{Ord}(\text{succ}(k))$   
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*:  $\text{Ord}(k) \implies j \in k \implies j \leq k$   
by (simp add: Ord-def Transset-def HBall-def)

**lemma** *Ord-trans*:  $\llbracket i \in j; j \in k; \text{Ord}(k) \rrbracket \implies i \in k$   
by (blast dest: OrdmemD)

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

**lemma** *Ord-in-Ord*:  $\llbracket \text{Ord}(k); m \in k \rrbracket \implies \text{Ord}(m)$   
by (auto simp: Ord-def Transset-def)

### 2.1.3 Induction, Linearity, etc.

**lemma** *Ord-induct [consumes 1, case-names step]*:  
assumes  $k: \text{Ord}(k)$   
and *step*:  $\bigwedge x. \llbracket \text{Ord}(x); \bigwedge y. y \in x \implies P(y) \rrbracket \implies P(x)$   
shows  $P(k)$

**proof** –

have  $\forall m \in k. \text{Ord}(m) \longrightarrow P(m)$

**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*:  $\text{Ord}(k) \implies \text{Ord}(l) \implies k \in l \vee k = l \vee l \in k$

**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 ≤ k
by (metis Ord-linear OrdmemD order-eq-refl o)

```

```

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

```

```

lemma hunion-less-iff [simp]: [[Ord i; Ord j]] ⇒ i ⊔ j < k ⇔ i < k ∧ 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 ≤ 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 ≤ 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 ∈ 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*]:  $\llbracket \bigwedge i. i \in A \implies \text{Ord}(i) \rrbracket \implies \text{Ord}(\bigsqcup A)$   
**by** (*auto simp: Ord-def Transset-def*) *blast*

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

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

**lemma** *hmem-Sup-Ords*:  $\llbracket A \neq 0; \bigwedge i. i \in A \implies \text{Ord}(i) \rrbracket \implies \bigsqcup A \in A$

**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.prem(2) hinsert-nonempty hmem-hinsert*)

**hence**  $\bigsqcup A \leq x \vee x \leq \bigsqcup A$

**by** (*metis Ord-linear2 OrdmemD hinsert.prem(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*:  $\llbracket A \neq 0; \bigwedge i. i \in A \implies \text{Ord}(i) \rrbracket \implies \bigsqcap A \in A$

**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.prem(2) hinsert-nonempty hmem-hinsert*)

**hence**  $\bigsqcap A \leq x \vee x \leq \bigsqcap A$

**by** (*metis Ord-linear2 OrdmemD hinsert.prem(2) hmem-hinsert*)

**thus** *?thesis*

**by** (*metis HInter-hinsert IA hmem-hempty hmem-hinsert inf-absorb2 le-iff-inf*)

**qed**

**qed**



**lemma** *Ord-pred*:  $\llbracket \text{Ord}(k); k \neq 0 \rrbracket \implies \text{succ}(\lfloor k) = k$   
**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*:  $\text{Ord}(k)$   
**obtains**  $k = 0 \mid l$  **where**  $\text{Ord } l \text{ succ } l = k$   
**by** (*metis* *Ok* *Ord-in-Ord* *Ord-pred succ-iff*)

**lemma** *Ord-induct2* [*consumes 1, case-names 0 succ, induct type: hf*]:  
**assumes** *k*:  $\text{Ord}(k)$   
**and**  $P: P \ 0 \wedge k. \text{Ord } k \implies P \ k \implies P \ (\text{succ } k)$   
**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*]:  $\text{Ord} \ (\text{succ } k) = \text{Ord } k$   
**by** (*metis* *Ord-in-Ord* *Ord-succ less-eq-insert1-iff* *order-refl succ-def*)

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

**lemma** *Ord-Sup-succ-eq* [*simp*]:  $\text{Ord } k \implies \lfloor (\text{succ } k) = k$   
**by** (*metis* *Ord-pred* *Ord-succ-iff succ-inject-iff* *hinsert-nonempty succ-def*)

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

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

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

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

**lemma** *Not-Ord-hpair* [*iff*]:  $\neg \text{Ord } \langle x, y \rangle$   
**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*:  $\text{Ord } x \implies \neg \text{is-hpair } x$   
**by** (*metis* *Not-Ord-hpair is-hpair-def*)

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

## 2.1.5 Converting Between Ordinals and Natural Numbers

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

**lemma** *Ord-ord-of* [*simp*]:  $\text{Ord } (\text{ord-of } k)$   
**by** (*induct k, auto*)

**lemma** *ord-of-inject* [*iff*]:  $\text{ord-of } i = \text{ord-of } j \iff i=j$   
**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*:  $n > 0 \implies \text{ord-of } n = \text{succ } (\text{ord-of } (n - 1))$   
**by** (*metis Suc-diff-1 ord-of.simps(2)*)

**definition** *nat-of-ord* ::  $\text{hf} \Rightarrow \text{nat}$   
**where** *nat-of-ord* x = (*THE* n. x = *ord-of* n)

**lemma** *nat-of-ord-ord-of* [*simp*]:  $\text{nat-of-ord } (\text{ord-of } n) = n$   
**by** (*auto simp: nat-of-ord-def*)

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

**lemma** *ord-of-nat-of-ord* [*simp*]:  $\text{Ord } x \implies \text{ord-of } (\text{nat-of-ord } x) = x$   
**apply** (*erule Ord-induct2, simp*)  
**apply** (*metis nat-of-ord-ord-of ord-of.simps(2)*)  
**done**

**lemma** *nat-of-ord-inject*:  $\text{Ord } x \implies \text{Ord } y \implies \text{nat-of-ord } x = \text{nat-of-ord } y \iff x = y$   
**by** (*metis ord-of-nat-of-ord*)

**lemma** *nat-of-ord-succ* [*simp*]:  $\text{Ord } x \implies \text{nat-of-ord } (\text{succ } x) = \text{Suc } (\text{nat-of-ord } x)$   
**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*:  $hfset (ord\text{-}of\ n) = ord\text{-}of\ \{0..<n\}$   
**by** (*induct n*) (*auto simp: hfset-hinsert succ-def*)

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

**lemma** *bij-betw-ord-ofI*:  $bij\text{-}betw\ h\ A\ \{0..<n\} \implies bij\text{-}betw\ (ord\text{-}of\ \circ\ h)\ A\ (hfset\ (ord\text{-}of\ n))$   
**by** (*blast intro: bij-betw-ord-of bij-betw-trans*)

## 2.2 Sequences and Ordinal Recursion

Definition 3.2 (Sequence).

**definition** *Seq* ::  $hf \Rightarrow hf \Rightarrow bool$   
**where**  $Seq\ s\ k \iff hrelation\ s \wedge hfunction\ s \wedge k \leq hdomain\ s$

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

**lemma** *Seq-succ-D*:  $Seq\ s\ (succ\ k) \implies Seq\ s\ k$   
**by** (*simp add: Seq-def succ-def*)

**lemma** *Seq-Ord-D*:  $Seq\ s\ k \implies l \in k \implies Ord\ k \implies Seq\ s\ l$   
**by** (*auto simp: Seq-def intro: Ord-trans*)

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

**lemma** *Seq-Ord-restr*:  $\llbracket Seq\ s\ k; l \in k; Ord\ k \rrbracket \implies Seq\ (hrestrict\ s\ l)\ l$   
**by** (*auto simp: Seq-def hfunction-restr intro: Ord-trans*)

**lemma** *Seq-ins*:  $\llbracket Seq\ s\ k; k \notin hdomain\ s \rrbracket \implies Seq\ (s \triangleleft \langle k, y \rangle)\ (succ\ k)$   
**by** (*auto simp: Seq-def hrelation-def succ-def hfunction-def hdomainI*)

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

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

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

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

**apply** (*metis hfst-conv, blast*)  
**done**

**lemma** *nonrestrictD*:  $a \in \text{nonrestrict } s \ X \implies a \in s$   
**by** (*auto simp: nonrestrict-def*)

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

**lemma** *app-nonrestrict-Seq*:  $\text{Seq } s \ k \implies z \notin X \implies \text{app } (\text{nonrestrict } s \ X) \ z = \text{app } s \ z$   
**by** (*auto simp: Seq-def nonrestrict-def app-def HBall-def*) (*metis*)

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

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

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

**lemma** *Seq-imp-eq-app*:  $\llbracket \text{Seq } s \ d; \langle x, y \rangle \in s \rrbracket \implies \text{app } s \ x = y$   
**by** (*metis Seq-def app-equality*)

**lemma** *Seq-iff-app*:  $\llbracket \text{Seq } s \ d; x \in d \rrbracket \implies \langle x, y \rangle \in s \iff \text{app } s \ x = y$   
**by** (*auto simp: Seq-def hdomain-def app-equality*)

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

**lemma** *Ord-trans2*:  $\llbracket i2 \in i; i \in j; j \in k; \text{Ord } k \rrbracket \implies i2 \in k$   
**by** (*metis Ord-trans*)

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

**lemma** *Seq-succ-insf*:  
**assumes**  $s: \text{Seq } s \ (\text{succ } k)$  **shows**  $\exists y. s = \text{insf } s \ k \ y$   
**proof** –  
**obtain**  $y$  **where**  $y: \langle k, y \rangle \in s$  **by** (*metis Seq-succ-iff s*)  
**hence**  $y \text{uniq}: \forall y'. \langle k, y' \rangle \in s \longrightarrow y' = y$  **using**  $s$   
**by** (*simp add: Seq-def hfunction-def*)

```

{ fix z
  assume z: z ∈ s
  then obtain u v where uv: z = ⟨u, v⟩ using s
    by (metis Seq-def hrelation-def is-hpair-def)
  hence z ∈ insf s k y
    by (metis hemptyE hmem-hinsert hpair-in-nonrestrict-iff insf-def yuniq z)
}
note left2right = this
show ?thesis
proof
  show s = insf s k y
    by (rule hf-equalityI) (metis hmem-hinsert insf-def left2right nonrestrictD
y)
qed
qed

```

**lemma** *ord-rec-Seq-succ-iff*:

```

assumes k: Ord k and knz: k ≠ 0
shows ord-rec-Seq T G s (succ k) z ⟷ (∃ s' y. ord-rec-Seq T G s' k y ∧ z =
G y ∧ s = insf s' k y)

```

**proof**

```

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

```

**next**

```

assume ok: ∃ 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  $\Rightarrow$  (hf  $\Rightarrow$  hf)  $\Rightarrow$  (hf  $\Rightarrow$  hf)  $\Rightarrow$  hf  $\Rightarrow$  hf  $\Rightarrow$  bool

where

ord-recp T G H x y =  
(if x=0 then y = T  
else  
if Ord(x) then  $\exists$  s. ord-rec-Seq T G s x y  
else y = H x)

**lemma** ord-recp-functional: ord-recp T G H x y  $\Longrightarrow$  ord-recp T G H x y'  $\Longrightarrow$  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  $\longleftrightarrow$  ( $\exists$  y. z = G y  $\wedge$  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  $\Rightarrow$  (hf  $\Rightarrow$  hf)  $\Rightarrow$  (hf  $\Rightarrow$  hf)  $\Rightarrow$  hf  $\Rightarrow$  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:  $\exists$  y. ord-recp T G H x y

**proof** (cases Ord x)

case True thus ?thesis

**proof** (induct x rule: Ord-induct2)

case 0 thus ?case

by (simp add: ord-recp-def)

next

case (succ x) thus ?case

by (metis ord-recp-succ-iff)

qed

next

case False thus ?thesis

by (auto simp: ord-recp-def)

qed

```

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

lemma ord-rec-non [simp]:  $\neg \text{Ord } x \implies \text{ord-rec } T \ G \ H \ x = H \ x$ 
  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* ::  $hf \Rightarrow hf$   
**where**  $Vset\ x = ord-rec\ 0\ HPow\ (\lambda z. 0)\ x$

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

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

**lemma** *Vset-non* [*simp*]:  $\neg Ord\ x \Longrightarrow Vset\ x = 0$   
**by** (*simp add: Vset-def*)

Theorem 4.2(a)

**lemma** *Vset-mono-strict*:

**assumes**  $Ord\ m\ n \in m$  **shows**  $Vset\ n < Vset\ m$

**proof** –

**have**  $n: Ord\ n$

**by** (*metis Ord-in-Ord assms*)

**hence**  $Ord\ m \Longrightarrow n \in m \Longrightarrow Vset\ n < Vset\ m$

**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*:  $\llbracket \text{Ord } m; n \leq m \rrbracket \implies \text{Vset } n \leq \text{Vset } m$   
**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*:  $\text{Ord } m \implies \text{Transset } (\text{Vset } m)$   
**by** (induct rule: Ord-induct2) (auto simp: Transset-def)

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

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

Theorem 4.3

**lemma** *Vset-universal*:  $\exists n. \text{Ord } n \wedge x \in \text{Vset } n$   
**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*:  $\text{Ord } na \wedge a \in \text{Vset } na \wedge \text{Ord } nb \wedge b \in \text{Vset } nb$   
**by** *blast*  
**hence**  $b \leq \text{Vset } nb$  **using** *Vset-Transset* [of *nb*]  
**by** (auto simp: Transset-def)  
**also have**  $\dots \leq \text{Vset } (na \sqcup nb)$  **using** *nab*  
**by** (metis Ord-sup Vset-mono sup-ge2)  
**finally have**  $b \triangleleft a \in \text{Vset } (\text{succ } (na \sqcup nb))$  **using** *nab*  
**by** *simp* (metis Ord-sup Vset-mono sup-ge1 rev-hsubsetD)  
**thus** *?case using* *nab*  
**by** (metis Ord-succ Ord-sup)  
**qed**

## 3.2 Least Ordinal Operator

Definition 4.4. For every  $x$ , let  $\text{rank}(x)$  be the least ordinal  $n$  such that...

**lemma** *Ord-minimal*:  
 $\text{Ord } k \implies P k \implies \exists n. \text{Ord } n \wedge P n \wedge (\forall m. \text{Ord } m \wedge P m \longrightarrow n \leq m)$   
**by** (induct k rule: Ord-induct) (metis Ord-linear2)

**lemma** *OrdLeastI*:  $\text{Ord } k \implies P k \implies P(\text{LEAST } n. \text{Ord } n \wedge P n)$   
**by** (metis (lifting, no-types) Least-equality Ord-minimal)

**lemma** *OrdLeast-le*:  $Ord\ k \implies P\ k \implies (LEAST\ n.\ Ord\ n \wedge P\ n) \leq k$   
**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*:  $succ(rank\ a) \leq k \implies Ord\ k \implies a \in Vset\ k$   
**by** (*metis* *Vset-mono* *hsubsetCE* *in-Vset-rank*)

**lemma** *Vset-succ-rank-le*:  $Ord\ k \implies a \in Vset\ (succ\ k) \implies rank\ a \leq k$   
**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 emptyE
in-Vset-rank)
  qed
}
thus ?thesis using a
by (metis Vset-non emptyE)
qed

```

Theorem 4.5

```

theorem rank-lt:  $a \in b \implies \text{rank}(a) < \text{rank}(b)$ 
  by (metis Vset-rank-lt hsubsetD le-Vset-rank)

```

```

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

```

```

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

```

```

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

```

```

lemma rank-hinsert [simp]:  $\text{rank } (b \triangleleft a) = \text{rank } b \sqcup \text{succ}(\text{rank } a)$ 
  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**

$eclose \quad :: \text{ hf } \Rightarrow \text{ hf } \text{ where}$   
 $eclose \ X = \sqcap \{ Y \in \text{HPow}(Vset \ (\text{rank } X)). \text{Transset } Y \wedge X \leq Y \}$

**lemma** *eclose-facts*:

**shows** *Transset-eclose*:  $\text{Transset} \ (eclose \ X)$   
**and** *le-eclose*:  $X \leq eclose \ X$

**proof** –

**have** *nz*:  $\{ Y \in \text{HPow}(Vset \ (\text{rank } X)). \text{Transset } Y \wedge X \leq Y \} \neq 0$   
**by** (*simp add: eclose-def empty-iff*) (*metis Ord-rank Vset-Transset le-Vset-rank order-refl*)  
**show**  $\text{Transset} \ (eclose \ X) \ X \leq eclose \ X$  **using** *HInter-iff* [*OF nz*]  
**by** (*auto simp: eclose-def Transset-def*)  
**qed**

**lemma** *eclose-minimal*:

**assumes**  $Y: \text{Transset } Y \ X \leq Y$  **shows**  $eclose \ X \leq Y$

**proof** –

**have**  $\{ Y \in \text{HPow}(Vset \ (\text{rank } X)). \text{Transset } Y \wedge X \leq Y \} \neq 0$   
**by** (*simp add: eclose-def empty-iff*) (*metis Ord-rank Vset-Transset le-Vset-rank order-refl*)  
**moreover have**  $\text{Transset} \ (Y \sqcap Vset \ (\text{rank } X))$   
**by** (*metis Ord-rank Transset-inf Vset-Transset Y(1)*)  
**moreover have**  $X \leq Y \sqcap Vset \ (\text{rank } X)$   
**by** (*metis Y(2) le-Vset-rank le-inf-iff*)  
**ultimately show**  $eclose \ X \leq Y$   
**apply** (*auto simp: eclose-def*)  
**apply** (*metis hinter-iff le-inf-iff order-refl*)  
**done**  
**qed**

**lemma** *eclose-0* [*simp*]:  $eclose \ 0 = 0$

**by** (*metis Ord-0 Vset-0 Vset-Transset eclose-minimal less-eq-empty*)

**lemma** *eclose-sup* [*simp*]:  $eclose \ (a \sqcup b) = eclose \ a \sqcup eclose \ b$

**proof** (*rule order-antisym*)

**show**  $eclose \ (a \sqcup b) \leq eclose \ a \sqcup eclose \ b$

**by** (*metis Transset-eclose Transset-sup eclose-minimal le-eclose sup-mono*)

**next**

**show**  $eclose \ a \sqcup eclose \ b \leq eclose \ (a \sqcup b)$

**by** (*metis Transset-eclose eclose-minimal le-eclose le-sup-iff*)

**qed**

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

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

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

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

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

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

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

### 3.5 Epsilon-Recursion

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

**lemma** *hmem-induct* [*case-names step*]:  
  **assumes** *ih*:  $\bigwedge x. (\bigwedge y. y \in x \implies P\ y) \implies P\ x$  **shows**  $P\ x$   
**proof** –  
  **have**  $\bigwedge y. y \in x \implies P\ y$   
  **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*  $\{(x,y). x \in y\}$

**lemma** *wf-hmem-rel*: *wf hmem-rel*

**proof** –

**have** *wf*  $\{(x,y). x \in y\}$   
**by** (*metis* (*full-types*) *hmem-induct wfPUNIVI wfP-def*)  
**thus** *?thesis*  
**by** (*metis hmem-rel-def wf-trancl*)

**qed**

**lemma** *hmem-eclose-le*:  $y \in x \implies \text{eclose } y \leq \text{eclose } x$

**by** (*metis Transset-def Transset-eclose eclose-minimal hsubsetD le-eclose*)

**lemma** *hmem-rel-iff-hmem-eclose*:  $(x,y) \in \text{hmem-rel} \longleftrightarrow x \in \text{eclose } y$

**proof** (*unfold hmem-rel-def, rule iffI*)

**assume**  $(x, y) \in \text{trancl } \{(x, y). x \in y\}$

**thus**  $x \in \text{eclose } y$

**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*  $\{x \in \text{eclose } y. (x, y) \in \text{hmem-rel}\}$  **using** *Transset-eclose*

**by** (*auto simp: Transset-def hmem-rel-def intro: trancl-trans*)

**hence**  $\text{eclose } y \leq \{x \in \text{eclose } y. (x, y) \in \text{hmem-rel}\}$

**by** (*rule eclose-minimal*) (*auto simp: le-HCollect-iff le-eclose hmem-rel-def*)

**moreover assume**  $x \in \text{eclose } y$

**ultimately show**  $(x, y) \in \text{trancl } \{(x, y). x \in y\}$

**by** (*metis HCollect-iff hmem-rel-def hsubsetD*)

**qed**

**definition** *hmemrec* ::  $((hf \Rightarrow 'a) \Rightarrow hf \Rightarrow 'a) \Rightarrow hf \Rightarrow 'a$  **where**

*hmemrec* *G*  $\equiv \text{wfrec } \text{hmem-rel } G$

**definition** *ecut* ::  $(hf \Rightarrow 'a) \Rightarrow hf \Rightarrow hf \Rightarrow 'a$  **where**

*ecut* *f* *x*  $\equiv (\lambda y. \text{if } y \in \text{eclose } x \text{ then } f y \text{ else undefined})$

**lemma** *hmemrec*: *hmemrec* *G* *a* = *G* (*ecut* (*hmemrec* *G*) *a*) *a*

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

This form avoids giant explosions in proofs.

**lemma** *def-hmemrec*:  $f \equiv \text{hmemrec } G \implies f a = G (\text{ecut } (\text{hmemrec } G) a) a$

**by** (*metis hmemrec*)

**lemma** *ecut-apply*:  $y \in \text{eclose } x \implies \text{ecut } f \ x \ y = f \ y$   
**by** (*metis ecut-def*)

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

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

**lemma** *hmem-rel-induct* [*case-names step*]:  
**assumes** *ih*:  $\bigwedge x. (\bigwedge y. (y, x) \in \text{hmem-rel} \implies P \ y) \implies P \ x$  **shows**  $P \ x$   
**proof** –  
**have**  $\bigwedge y. (y, x) \in \text{hmem-rel} \implies P \ y$   
**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*:  $x \neq 0 \implies \text{rank } (\bigsqcup x) < \text{rank } x$   
**apply** (*induct x rule: hf-induct, auto*)  
**apply** (*metis hmem-hinsert rank-hinsert rank-lt*)  
**apply** (*metis HUnion-hempty Ord-lt-succ-iff-le Ord-rank hunion-hempty-right less-supI1 less-supI2 rank-sup sup.cobounded2*)  
**done**

**corollary** *Sup-ne*:  $x \neq 0 \implies \bigsqcup x \neq x$   
**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  $\Rightarrow$  'a  $\Rightarrow$  hf  $\Rightarrow$  bool
```

```
inductive reaches :: ['a fsm, hf, 'a list, hf]  $\Rightarrow$  bool
```

```
where
```

```
  Nil: st  $\in$  states fsm  $\implies$  reaches fsm st [] st  
  | Cons: [next fsm st x st''; reaches fsm st'' xs st'; st  $\in$  states fsm]  $\implies$  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'  $\implies$  st  $\in$  states fsm  $\wedge$  st'  $\in$  states  
fsm
```

```
  by (induct xs arbitrary: st st', auto)
```

```
lemma reaches-append-iff:
```

```
  reaches fsm st (xs@ys) st'  $\longleftrightarrow$  ( $\exists$  st''. reaches fsm st xs st''  $\wedge$  reaches fsm st''  
ys st')
```

```
  by (induct xs arbitrary: ys st st') (auto simp: reaches-imp-states)
```

```
definition accepts :: 'a fsm  $\Rightarrow$  'a list  $\Rightarrow$  bool where
```



$accepts\ fsm\ xs \equiv \exists st\ st'.\ reaches\ fsm\ st\ xs\ st' \wedge st \in init\ fsm \wedge st' \in final\ fsm$

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

**definition** *Null* **where**  
 $Null = (\{states = 0, init = 0, final = 0, next = \lambda st\ x\ st'.\ False\})$

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

**abbreviation** *NullStr* **where**  
 $NullStr \equiv (\{states = 1, init = 1, final = 1, next = \lambda st\ x\ st'.\ False\})$

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

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

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

**definition** *Reverse* **where**  
 $Reverse\ fsm = (\{states = states\ fsm, init = final\ fsm, final = init\ fsm, next = \lambda st\ x\ st'.\ next\ fsm\ st'\ x\ st\})$

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

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

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

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

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

**theorem** regular-rev:  $regular S \implies regular (rev \text{ ` } S)$   
**apply** (auto simp: regular-def accepts-def)  
**apply** (rule-tac x=Reverse fsm in exI, force+)  
**done**

**definition** Times **where**

$$\begin{aligned} Times fsm1 fsm2 = (&states = states fsm1 * states fsm2, \\ &init = init fsm1 * init fsm2, \\ &final = final fsm1 * final fsm2, \\ &next = \lambda st x st'. (\exists st1 st2 st1' st2'. st = \langle st1, st2 \rangle \wedge st' = \\ &\langle st1', st2' \rangle \wedge \\ &\quad next fsm1 st1 x st1' \wedge next fsm2 st2 x st2')) \end{aligned}$$

**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:  $next (Times fsm1 fsm2) \langle st1, st2 \rangle x st' \longleftrightarrow$   
 $(\exists st1' st2'. st' = \langle st1', st2' \rangle \wedge next fsm1 st1 x st1' \wedge next fsm2 st2 x st2')$   
**by** (simp add: Times-def)

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

$$\begin{aligned} reaches (Times fsm1 fsm2) \langle st1, st2 \rangle xs \langle st1', st2' \rangle \longleftrightarrow \\ reaches fsm1 st1 xs st1' \wedge reaches fsm2 st2 xs st2' \end{aligned}$$

**apply** (induct xs arbitrary: st1 st2 st1' st2', force)

**apply** (force simp add: next-Times Times-def reaches.Cons)

**done**

**lemma** accepts-Times-iff [simp]:

$$\begin{aligned} accepts (Times fsm1 fsm2) xs \longleftrightarrow \\ accepts fsm1 xs \wedge accepts fsm2 xs \end{aligned}$$

**by** (force simp add: accepts-def)

**theorem** regular-Int:

**assumes** S:  $regular S$  **and** T:  $regular T$  **shows**  $regular (S \cap T)$

**proof** –

**obtain** fsmS fsmT **where**  $S = \{xs. accepts fsmS xs\}$   $T = \{xs. accepts fsmT xs\}$

**using** S T

**by** (auto simp: regular-def)

**hence**  $S \cap T = \{xs. accepts (Times fsmS fsmT) xs\}$

by (auto simp: accepts-Times-iff [of fsmS fsmT])  
 thus ?thesis  
 by (metis regular-def)  
 qed

**definition** *Plus* where

$$\begin{aligned}
 \text{Plus fsm1 fsm2} = & (\text{states} = \text{states fsm1} + \text{states fsm2}, \\
 & \text{init} = \text{init fsm1} + \text{init fsm2}, \\
 & \text{final} = \text{final fsm1} + \text{final fsm2}, \\
 & \text{next} = \lambda st\ x\ st'. (\exists st1\ st1'. st = \text{Inl } st1 \wedge st' = \text{Inl } st1' \wedge \text{next} \\
 & \text{fsm1 } st1\ x\ st1') \vee \\
 & (\exists st2\ st2'. st = \text{Inr } st2 \wedge st' = \text{Inr } st2' \wedge \text{next} \\
 & \text{fsm2 } st2\ x\ st2'))
 \end{aligned}$$

**lemma** *states-Plus* [simp]:  $\text{states } (\text{Plus fsm1 fsm2}) = \text{states fsm1} + \text{states fsm2}$   
 by (simp add: Plus-def)

**lemma** *init-Plus* [simp]:  $\text{init } (\text{Plus fsm1 fsm2}) = \text{init fsm1} + \text{init fsm2}$   
 by (simp add: Plus-def)

**lemma** *final-Plus* [simp]:  $\text{final } (\text{Plus fsm1 fsm2}) = \text{final fsm1} + \text{final fsm2}$   
 by (simp add: Plus-def)

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

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

**lemma** *reaches-Plus-iff1* [simp]:  
 $\text{reaches } (\text{Plus fsm1 fsm2}) (\text{Inl } st1)\ xs\ st' \longleftrightarrow$   
 $(\exists st1'. st' = \text{Inl } st1' \wedge \text{reaches fsm1 } st1\ xs\ st1')$

**apply** (induct xs arbitrary: st1, force)  
**apply** (force simp add: next-Plus1 reaches.Cons)  
**done**

**lemma** *reaches-Plus-iff2* [simp]:  
 $\text{reaches } (\text{Plus fsm1 fsm2}) (\text{Inr } st2)\ xs\ st' \longleftrightarrow$   
 $(\exists st2'. st' = \text{Inr } st2' \wedge \text{reaches fsm2 } st2\ xs\ st2')$

**apply** (induct xs arbitrary: st2, force)  
**apply** (force simp add: next-Plus2 reaches.Cons)  
**done**

**lemma** *reaches-Plus-iff* [simp]:  
 $\text{reaches } (\text{Plus fsm1 fsm2}) st\ xs\ st' \longleftrightarrow$   
 $(\exists st1\ st1'. st = \text{Inl } st1 \wedge st' = \text{Inl } st1' \wedge \text{reaches fsm1 } st1\ xs\ st1') \vee$   
 $(\exists st2\ st2'. st = \text{Inr } st2 \wedge st' = \text{Inr } st2' \wedge \text{reaches fsm2 } st2\ xs\ st2')$

```

apply (induct xs arbitrary: st st', auto)
apply (force simp add: next-Plus1 next-Plus2 Plus-def reaches.Cons)
apply (auto simp: Plus-def)
done

```

```

lemma accepts-Plus-iff [simp]:
  accepts (Plus fsm1 fsm2) xs  $\longleftrightarrow$  accepts fsm1 xs  $\vee$  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  $\cup$  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  $\cup$  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  $\Rightarrow$  hf
  assumes inj: inj hf-of
begin
  lemma hf-of-eq-iff [simp]: hf-of x = hf-of y  $\longleftrightarrow$  x=y
  using inj by (auto simp: inj-on-def)
end

```

```

instantiation unit :: finitary
begin
  definition hf-of-unit-def: hf-of (u::unit)  $\equiv$  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  $\equiv$  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  $\equiv$  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  $\equiv$  if  $i \geq (0 :: int)$  then  $\langle 0, hf-of (nat i) \rangle$  else  $\langle 1, hf-of (nat (-i)) \rangle$ 
  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  $\equiv$  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  $\equiv$   $\lambda(x,y). \langle hf-of x, hf-of y \rangle$ 
  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  $\equiv$  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  $\equiv$  case-option 0 ( $\lambda x. \{ hf-of x \}$ )
  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  $\implies$  x = y
  apply (induct x arbitrary: y, auto)
  apply (metis (mono-tags) hf-of-list.simps(2) hpair-nonzero neq-Nil-conv)
  apply (rename-tac y)
  apply (case-tac y, auto)
  done
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**

$\text{hadd} \quad :: \text{hf} \Rightarrow \text{hf} \Rightarrow \text{hf} \quad (\text{infixl } @+ \ 65) \quad \text{where}$   
 $\text{hadd } x \equiv \text{hmemrec } (\lambda f z. x \sqcup \text{RepFun } z f)$

**lemma** *hadd*:  $x @+ y = x \sqcup \text{RepFun } y (\lambda z. x @+ z)$   
**by** (*metis def-hmemrec RepFun-ecut hadd-def order-refl*)

**lemma** *hmem-hadd-E*:

**assumes**  $l \in x @+ y$   
**obtains**  $l \in x \mid z \text{ where } z \in y \ l = x @+ z$   
**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*:  $x @+ \text{hinsert } y z = \text{hinsert } (x @+ y) (x @+ z)$   
**by** (*metis hadd hunion-hinsert-right RepFun-hinsert*)

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

**lemma** *not-add-less-right*:  $\neg (x @+ y < x)$

**apply** (*induct y, auto*)  
**apply** (*metis less-supI1 hadd order-less-le*)  
**done**

**lemma** *not-add-mem-right*:  $\neg (x @+ y \in x)$   
**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*]:  $\text{Ord } y \implies \text{succ } x @+ y = \text{succ } (x @+ y)$   
**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*:  $x \sqcap \text{RepFun } y ((@+) x) = 0$   
**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*:  $x \sqcup \text{RepFun } y ((@+) x) \leq x \sqcup \text{RepFun } z ((@+) x)$   
 $\implies \text{RepFun } y ((@+) x) \leq \text{RepFun } z ((@+) x)$   
**by** (*auto simp add: not-add-mem-right*)

**lemma** *hadd-cancel-right* [*simp*]:  $x @+ y = x @+ z \longleftrightarrow y = z$

**proof** (*induct y arbitrary: z rule: hmem-induct*)

**case** (*step y z*) **show** ?*case*

**proof** *auto*

**assume** *eq*:  $x @+ y = x @+ z$

**hence**  $\text{RepFun } y ((@+) x) = \text{RepFun } z ((@+) x)$

**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*:  $\text{RepFun } y (\lambda z. x @+ z) = \text{RepFun } z (\lambda z. x @+ z)$   
 $\longleftrightarrow y = z$

**by** (*metis hadd hadd-cancel-right*)

**lemma** *hadd-hmem-cancel* [*simp*]:  $x @+ y \in x @+ z \longleftrightarrow y \in z$   
**apply** (*auto simp: hadd [of - y] hadd [of - z] not-add-mem-right*)  
**apply** (*metis hmem-not-refl hunion-iff*)  
**apply** (*metis hadd hadd-cancel-right*)  
**done**

**lemma** *ord-of-add*:  $\text{ord-of } (i+j) = \text{ord-of } i @+ \text{ord-of } j$   
**by** (*induct j*) *auto*



**lemma** *Ord-hadd*:  $\text{Ord } x \implies \text{Ord } y \implies \text{Ord } (x @+ y)$   
**by** (*induct x rule: Ord-induct2*) *auto*

**lemma** *hmem-self-hadd* [*simp*]:  $k1 \in k1 @+ k2 \longleftrightarrow 0 \in k2$   
**by** (*metis hadd-0-right hadd-hmem-cancel*)

**lemma** *hadd-commute*:  $\text{Ord } x \implies \text{Ord } y \implies x @+ y = y @+ x$   
**by** (*induct x rule: Ord-induct2*) *auto*

**lemma** *hadd-cancel-left* [*simp*]:  $\text{Ord } x \implies y @+ x = z @+ x \longleftrightarrow y=z$   
**by** (*induct x rule: Ord-induct2*) *auto*

### 5.1.2 The predecessor function

**definition** *pred* ::  $hf \Rightarrow hf$   
**where**  $\text{pred } x \equiv (\text{THE } y. \text{succ } y = x \vee x=0 \wedge y=0)$

**lemma** *pred-succ* [*simp*]:  $\text{pred } (\text{succ } x) = x$   
**by** (*simp add: pred-def*)

**lemma** *pred-0* [*simp*]:  $\text{pred } 0 = 0$   
**by** (*simp add: pred-def*)

**lemma** *succ-pred* [*simp*]:  $\text{Ord } x \implies x \neq 0 \implies \text{succ } (\text{pred } x) = x$   
**by** (*metis Ord-cases pred-succ*)

**lemma** *pred-mem* [*simp*]:  $\text{Ord } x \implies x \neq 0 \implies \text{pred } x \in x$   
**by** (*metis succ-iff succ-pred*)

**lemma** *Ord-pred* [*simp*]:  $\text{Ord } x \implies \text{Ord } (\text{pred } x)$   
**by** (*metis Ord-in-Ord pred-0 pred-mem*)

**lemma** *hadd-pred-right*:  $\text{Ord } y \implies y \neq 0 \implies x @+ \text{pred } y = \text{pred } (x @+ y)$   
**by** (*metis hadd-succ-right pred-succ succ-pred*)

**lemma** *Ord-pred-HUnion*:  $\text{Ord}(k) \implies \text{pred } k = \bigsqcup k$   
**by** (*metis HUnion-hempty Ordinal.Ordinal-pred pred-0 pred-succ*)

## 5.2 A Concatentation Operation for Sequences

**definition** *shift* ::  $hf \Rightarrow hf \Rightarrow hf$   
**where**  $\text{shift } f \text{ delta} = \{v . u \in f, \exists n y. u = \langle n, y \rangle \wedge v = \langle \text{delta } @+ n, y \rangle\}$

**lemma** *shiftD*:  $x \in \text{shift } f \text{ delta} \implies \exists u. u \in f \wedge x = \langle \text{delta } @+ \text{hfst } u, \text{hsnd } u \rangle$   
**by** (*auto simp: shift-def hsplit-def*)

**lemma** *hmem-shift-iff*:  $\langle m, y \rangle \in \text{shift } f \text{ delta} \longleftrightarrow (\exists n. m = \text{delta } @+ n \wedge \langle n, y \rangle \in f)$

by (auto simp: shift-def hrelation-def is-hpair-def)

**lemma** hmem-shift-add-iff [simp]:  $\langle \text{delta } @+ \ n, \ y \rangle \in \text{shift } f \ \text{delta} \iff \langle n, \ y \rangle \in f$   
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]:  $\text{app } (\text{shift } f \ k) \ (k \ @+ \ j) = \text{app } f \ j$   
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:  $\text{hdomain } (\text{shift } f \ \text{delta}) = \{\text{delta } @+ \ n \mid n \in \text{hdomain } f\}$   
by (rule hf-equalityI) (force simp add: hdomain-def hmem-shift-iff)

**lemma** hdomain-shift-disjoint:  $\text{delta} \sqcap \text{hdomain } (\text{shift } f \ \text{delta}) = 0$   
by (auto simp: hdomain-def intro!: hf-equalityI) (metis shiftD hpair-inject not-add-mem-right)

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

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

**lemma** Seq-append:  $\text{Seq } s1 \ k1 \implies \text{Seq } s2 \ k2 \implies \text{Seq } (\text{seq-append } k1 \ s1 \ s2) \ (k1 \ @+ \ k2)$   
**apply** (auto simp: Seq-def seq-append-def)  
**apply** (metis hdomain-restr hdomain-shift-disjoint hfunction-hunion hfunction-restr hfunction-shift-iff inf-absorb2 seq-append-def)  
**apply** (simp add: hdomain-shift-add)  
**apply** (metis hmem-hadd-E rev-hsubsetD)  
**apply** (erule hmem-hadd-E, assumption, auto)  
**apply** (metis Seq-def Seq-iff-app hdomainI hmem-shift-add-iff)  
**done**

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

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

**lemma** Seq-append-app1:  $\text{Seq } s \ k \implies l \in k \implies \text{app } (\text{seq-append } k \ s \ s') \ l = \text{app } s \ l$   
**apply** (auto simp: Seq-def seq-append-def)  
**apply** (metis app-hunion1 hdomain-shift-disjoint hemptyE hinter-iff app-hrestrict)

done

**lemma** *Seq-append-app2*:  $Seq\ s1\ k1 \implies Seq\ s2\ k2 \implies l = k1\ @+ j \implies app$   
(*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*:  $\llbracket OrdDom\ s; Ord\ k \rrbracket \implies OrdDom\ (insf\ s\ (succ\ k)\ y)$   
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*:  $OrdDom\ s \implies OrdDom\ (hrestrict\ s\ A)$   
by (*auto simp: OrdDom-def*)

**lemma** *OrdDom-shift*:  $\llbracket OrdDom\ s; Ord\ k \rrbracket \implies OrdDom\ (shift\ s\ k)$   
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*:  $LstSeq\ s\ k\ y \implies Seq\ s\ (succ\ k)$   
by (*metis LstSeq-def*)

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

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

**lemma** *LstSeq-trunc*:  $LstSeq\ s\ k\ y \implies l \in k \implies LstSeq\ s\ l\ (app\ s\ l)$   
apply (*auto simp: LstSeq-def Seq-iff-app*)  
apply (*metis Ord-succ Seq-Ord-D mem-succ-iff*)  
apply (*metis Ord-in-Ord*)  
done

**lemma** *LstSeq-insf*:  $LstSeq\ s\ k\ z \implies LstSeq\ (insf\ s\ (succ\ k)\ y)\ (succ\ k)\ y$   
by (*metis OrdDom-insf LstSeq-def Ord-succ-iff Seq-imp-eq-app Seq-insf Seq-succ-iff app-insf-Seq*)

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

**lemma** *app-insf2-LstSeq*:  $LstSeq\ s\ k\ z \implies k' \neq succ\ k \implies app\ (insf\ s\ (succ\ k)\ y)\ k' = app\ s\ k'$

**by** (*metis LstSeq-imp-Seq-succ app-insf2-Seq*)

**lemma** *app-insf-LstSeq-if*:  $LstSeq\ s\ k\ z \implies app\ (insf\ s\ (succ\ k)\ y)\ k' = (if\ k' = succ\ k\ then\ y\ else\ app\ s\ k')$

**by** (*metis app-insf2-LstSeq app-insf-LstSeq*)

**lemma** *LstSeq-append-app1*:

$LstSeq\ s\ k\ y \implies l \in succ\ k \implies app\ (seq-append\ (succ\ k)\ s\ s')\ l = app\ s\ l$

**by** (*metis LstSeq-imp-Seq-succ Seq-append-app1*)

**lemma** *LstSeq-append-app2*:

$\llbracket LstSeq\ s1\ k1\ y1; LstSeq\ s2\ k2\ y2; l = succ\ k1\ @+ j \rrbracket$

$\implies app\ (seq-append\ (succ\ k1)\ s1\ s2)\ l = app\ s2\ j$

**by** (*metis LstSeq-imp-Seq-succ Seq-append-app2*)

**lemma** *Seq-append-pair*:

$\llbracket Seq\ s1\ k1; Seq\ s2\ (succ\ n); \langle n, y \rangle \in s2; Ord\ n \rrbracket \implies \langle k1\ @+ n, y \rangle \in (seq-append\ k1\ s1\ s2)$

**by** (*metis hmem-shift-add-iff hunion-iff seq-append-def*)

**lemma** *Seq-append-OrdDom*:  $\llbracket Ord\ k; OrdDom\ s1; OrdDom\ s2 \rrbracket \implies OrdDom\ (seq-append\ k\ s1\ s2)$

**by** (*auto simp: seq-append-def OrdDom-hrestrict OrdDom-shift*)

**lemma** *LstSeq-append*:

$\llbracket LstSeq\ s1\ k1\ y1; LstSeq\ s2\ k2\ y2 \rrbracket \implies LstSeq\ (seq-append\ (succ\ k1)\ s1\ s2)\ (succ\ (k1\ @+ k2))\ y2$

**apply** (*auto simp: LstSeq-def Seq-append Ord-hadd Seq-append-pair*)

**apply** (*metis Seq-append hadd-succ-left hadd-succ-right*)

**apply** (*metis Seq-append-pair hadd-succ-left*)

**apply** (*metis Ord-succ Seq-append-OrdDom*)

**done**

**lemma** *LstSeq-app [simp]*:  $LstSeq\ s\ k\ y \implies app\ s\ k = y$

**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*:  $LstSeq\ s\ k\ y \implies (\bigwedge l. l \in succ\ k \implies Builds\ B\ C\ s\ l) \implies BuildSeq\ B\ C\ s\ k\ y$

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

**lemma** *BuildSeq-imp-LstSeq*:  $BuildSeq\ B\ C\ s\ k\ y \implies LstSeq\ s\ k\ y$

**by** (*metis BuildSeq-def*)

**lemma** *BuildSeq-imp-Seq*:  $BuildSeq\ B\ C\ s\ k\ y \implies Seq\ s\ (succ\ k)$

**by** (*metis LstSeq-imp-Seq-succ BuildSeq-imp-LstSeq*)

**lemma** *BuildSeq-conj-distrib*:

$BuildSeq\ (\lambda x. B\ x \wedge P\ x)\ (\lambda x\ y\ z. C\ x\ y\ z \wedge P\ x)\ s\ k\ y \longleftrightarrow$

$BuildSeq\ B\ C\ s\ k\ y \wedge (\forall l \in succ\ k. P\ (app\ s\ l))$

**by** (*auto simp: BuildSeq-def Builds-def*)

**lemma** *BuildSeq-mono*:

**assumes**  $y: BuildSeq\ B\ C\ s\ k\ y$

**and**  $B: \bigwedge x. B\ x \implies B'\ x$  **and**  $C: \bigwedge x\ y\ z. C\ x\ y\ z \implies C'\ x\ y\ z$

**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**  $BuildSeq\ B\ C\ s\ l\ (app\ s\ l)$

**proof** –

{ **fix**  $j$

**assume**  $j: j \in succ\ l$

**have**  $k: Ord\ k$

**by** (*metis BuildSeq-imp-LstSeq LstSeq-def b*)

**hence**  $Built\ B\ C\ s\ j$

**by** (*metis BuildSeq-def OrdmemD b hballe hsubsetD j l succ-iff*)

}

**thus** *?thesis* **using**  $b\ l$

**by** (*auto simp: BuildSeq-def LstSeq-trunc*)

**qed**

### 5.3.2 Showing that Sequences can be Constructed

**lemma** *Built-insf*:  $Built\ B\ C\ s\ l \implies LstSeq\ s\ k\ z \implies l \in succ\ k \implies Built\ B\ C\ (insf\ s\ (succ\ k)\ y)\ l$

**by** (*auto simp: HBall-def hmem-not-refl Built-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 ∈ succ k
    and y: B y ∨ 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 ∈ 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) ∨ 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
have LstSeq s k z
  by (metis BuildSeq-imp-LstSeq b)
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:  $\bigwedge l. l \in \text{succ } k1 \implies \text{Builds } B \ C \ s1 \ l$ 
and s2L: LstSeq s2 k2 y2
and s2BC:  $\bigwedge l. l \in \text{succ } k2 \implies \text{Builds } B \ C \ s2 \ l$ 
  using s1 s2 by (auto simp: BuildSeq-def)
assume l:  $l \in \text{succ } (\text{succ } (k1 \ @+ \ k2))$ 
hence  $l \in \text{succ } k1 \ @+ \ \text{succ } k2$ 
  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:  $l \in \text{succ } k1$ 
  hence  $B \ (\text{app } s1 \ l) \vee (\exists m \in l. \exists n \in l. C \ (\text{app } s1 \ l) \ (\text{app } s1 \ m) \ (\text{app } s1 \ n))$ 
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  $\exists m \in l. \exists n \in l. C \ (\text{app } s1 \ l) \ (\text{app } s1 \ m) \ (\text{app } s1 \ n)$ 
  then obtain m n where mn:  $m \in l \ n \in l$  and C:  $C \ (\text{app } s1 \ l) \ (\text{app } s1 \ m)$ 
  (app s1 n)
  by blast
  also have  $m \in \text{succ } k1 \ n \in \text{succ } k1$ 
  by (metis LstSeq-def Ord-trans l1 mn s1L succ-iff)+
  ultimately have  $C \ (\text{app } (\text{seq-append } (\text{succ } k1) \ s1 \ s2) \ l)$ 
     $(\text{app } (\text{seq-append } (\text{succ } k1) \ s1 \ s2) \ m)$ 
     $(\text{app } (\text{seq-append } (\text{succ } k1) \ s1 \ s2) \ n)$ 
  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:  $z \in \text{succ } k2$  and l2:  $l = \text{succ } k1 \ @+ \ z$ 
hence  $B \ (\text{app } s2 \ z) \vee (\exists m \in z. \exists n \in z. C \ (\text{app } s2 \ z) \ (\text{app } s2 \ m) \ (\text{app } s2 \ n))$ 
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  $\exists m \in z. \exists n \in z. C \ (\text{app } s2 \ z) \ (\text{app } s2 \ m) \ (\text{app } s2 \ n)$ 
  then obtain m n where mn:  $m \in z \ n \in z$  and C:  $C \ (\text{app } s2 \ z) \ (\text{app } s2 \ m)$ 
  (app s2 n)

```

```

    by blast
  also have  $m \in \text{succ } k2$   $n \in \text{succ } k2$  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  $b1: \text{BuildSeq } B \ C \ s1 \ k1 \ y1$  and  $b2: \text{BuildSeq } B \ C \ s2 \ k2 \ y2$ 
    and  $y: C \ y \ y1 \ y2$ 
  shows  $\text{BuildSeq } B \ C$  (insf (seq-append (succ  $k1$ )  $s1$   $s2$ ) (succ (succ ( $k1$  @+
 $k2$ )))  $y$ ) (succ (succ ( $k1$  @+  $k2$ )))  $y$ 
  proof -
    have  $k2: \text{Ord } k2$  using  $b2$ 
      by (auto simp: BuildSeq-def LstSeq-def)
    show ?thesis
  proof (rule BuildSeq-insf [where  $m=k1$  and  $n=\text{succ}(k1@+k2)$ ])
    show  $\text{BuildSeq } B \ C$  (seq-append (succ  $k1$ )  $s1$   $s2$ ) (succ ( $k1$  @+  $k2$ ))  $y2$ 
      by (rule BuildSeq-append [OF  $b1$   $b2$ ])
    next
      show  $k1 \in \text{succ} (\text{succ} (k1 @+ k2))$  using  $k2$ 
        by (metis hadd-0-right hmem-0-Ord hmem-self-hadd succ-iff)
    next
      show  $\text{succ} (k1 @+ k2) \in \text{succ} (\text{succ} (k1 @+ k2))$ 
        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  $B \ y \vee$ 
         $C \ y$  (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:  $LstSeq \ \{\langle 0, y \rangle\} \ 0 \ y$

by (auto simp: LstSeq-def One-hf-eq-succ Seq-ins OrdDom-def)

lemma BuildSeq-1:  $B \ y \implies \text{BuildSeq } B \ C \ \{\langle 0, y \rangle\} \ 0 \ y$



by (auto simp: BuildSeq-def Builds-def LstSeq-1)

**lemma** *BuildSeq-exI*:  $B\ t \implies \exists\ s\ k. \text{BuildSeq}\ B\ C\ s\ k\ t$   
 by (metis BuildSeq-1)

### 5.3.3 Proving Properties of Given Sequences

**lemma** *BuildSeq-succ-E*:

assumes  $s: \text{BuildSeq}\ B\ C\ s\ k\ y$

obtains  $B\ y \mid m\ n$  where  $m \in k\ n \in k\ C\ y\ (\text{app}\ s\ m)\ (\text{app}\ s\ n)$

**proof** –

have  $Bs: \text{Builds}\ B\ C\ s\ k$  and  $\text{apps}: \text{app}\ s\ k = y$  using  $s$

by (auto simp: BuildSeq-def)

hence  $B\ y \vee (\exists\ m \in k. \exists\ n \in k. C\ y\ (\text{app}\ s\ m)\ (\text{app}\ s\ n))$

by (metis Builds-def apps Bs)

thus ?thesis using that

by auto

qed

**lemma** *BuildSeq-induct* [consumes 1, case-names  $B\ C$ ]:

assumes major:  $\text{BuildSeq}\ B\ C\ s\ k\ a$

and  $B: \bigwedge x. B\ x \implies P\ x$

and  $C: \bigwedge x\ y\ z. C\ x\ y\ z \implies P\ y \implies P\ z \implies P\ x$

shows  $P\ a$

**proof** –

have  $\text{Ord}\ k$  using *assms*

by (auto simp: BuildSeq-def LstSeq-def)

hence  $\bigwedge a\ s. \text{BuildSeq}\ B\ C\ s\ k\ a \implies P\ a$

by (induction  $k$  rule: Ord-induct) (metis BuildSeq-trunc BuildSeq-succ-E  $B\ C$ )

thus ?thesis

by (metis major)

qed

**definition** *BuildSeq2* ::  $[[hf, hf] \Rightarrow \text{bool}, [hf, hf, hf, hf, hf, hf] \Rightarrow \text{bool}, hf, hf, hf, hf] \Rightarrow \text{bool}$

where  $\text{BuildSeq2}\ B\ C\ s\ k\ y\ y' \equiv$

$\text{BuildSeq}\ (\lambda p. \exists x\ x'. p = \langle x, x' \rangle \wedge B\ x\ x')$

$(\lambda p\ q\ r. \exists x\ x'\ y\ y'\ z\ z'. p = \langle x, x' \rangle \wedge q = \langle y, y' \rangle \wedge r = \langle z, z' \rangle \wedge C\ x\ x'\ y\ y'\ z\ z')$

$s\ k\ \langle y, y' \rangle$

**lemma** *BuildSeq2-combine*:

assumes  $b1: \text{BuildSeq2}\ B\ C\ s1\ k1\ y1\ y1'$  and  $b2: \text{BuildSeq2}\ B\ C\ s2\ k2\ y2\ y2'$

and  $y: C\ y\ y'\ y1\ y1'\ y2\ y2'$

shows  $\text{BuildSeq2}\ B\ C\ (\text{insf}\ (\text{seq-append}\ (\text{succ}\ k1)\ s1\ s2))\ (\text{succ}\ (\text{succ}\ (k1\ @+ k2)))\ \langle y, y' \rangle$

$(\text{succ}\ (\text{succ}\ (k1\ @+ k2)))\ y\ y'$

using *assms*

apply (unfold BuildSeq2-def)

```

apply (blast intro: BuildSeq-combine)
done

lemma BuildSeq2-1:  $B\ y\ y' \implies \text{BuildSeq2}\ B\ C\ \{\langle 0, y, y' \rangle\}\ 0\ y\ y'$ 
by (auto simp: BuildSeq2-def BuildSeq-1)

lemma BuildSeq2-exI:  $B\ t\ t' \implies \exists s\ k. \text{BuildSeq2}\ B\ C\ s\ k\ t\ t'$ 
by (metis BuildSeq2-1)

lemma BuildSeq2-induct [consumes 1, case-names B C]:
  assumes BuildSeq2 B C s k a a'
    and B:  $\bigwedge x\ x'. B\ x\ x' \implies P\ x\ x'$ 
    and C:  $\bigwedge x\ x'\ y\ y'\ z\ z'. C\ x\ x'\ y\ y'\ z\ z' \implies P\ y\ y' \implies P\ z\ z' \implies P\ x\ x'$ 
  shows P a a'
using assms
apply (simp add: BuildSeq2-def)
apply (drule BuildSeq-induct [where P =  $\lambda(x,x'). P\ x\ x'$ ])
apply (auto intro: B C)
done

definition BuildSeq3
  :: [[hf,hf,hf]  $\Rightarrow$  bool, [hf,hf,hf,hf,hf,hf,hf,hf,hf,hf]  $\Rightarrow$  bool, hf, hf, hf, hf, hf]  $\Rightarrow$ 
  bool
  where BuildSeq3 B C s k y y' y''  $\equiv$ 
    BuildSeq ( $\lambda p. \exists x\ x'\ x''. p = \langle x, x', x'' \rangle \wedge B\ x\ x'\ x''$ )
      ( $\lambda p\ q\ r. \exists x\ x'\ x''\ y\ y'\ y''\ z\ z'\ z''.$ 
         $p = \langle x, x', x'' \rangle \wedge q = \langle y, y', y'' \rangle \wedge r = \langle z, z', z'' \rangle \wedge$ 
         $C\ x\ x'\ x''\ y\ y'\ y''\ z\ z'\ z''$ )
      s k  $\langle y, y', y'' \rangle$ 

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))))  $\langle y, y', y'' \rangle$ 
    (succ (succ (k1 @+ k2))) y y' y''
  using assms
  apply (unfold BuildSeq3-def)
  apply (blast intro: BuildSeq-combine)
  done

lemma BuildSeq3-1:  $B\ y\ y'\ y'' \implies \text{BuildSeq3}\ B\ C\ \{\langle 0, y, y', y'' \rangle\}\ 0\ y\ y'\ y''$ 
by (auto simp: BuildSeq3-def BuildSeq-1)

lemma BuildSeq3-exI:  $B\ t\ t'\ t'' \implies \exists s\ k. \text{BuildSeq3}\ B\ C\ s\ k\ t\ t'\ t''$ 
by (metis BuildSeq3-1)

lemma BuildSeq3-induct [consumes 1, case-names B C]:

```

```

assumes BuildSeq3 B C s k a a' a''
  and B:  $\bigwedge x x' x''. B x x' x'' \implies P x x' x''$ 
  and C:  $\bigwedge x x' x'' y y' y'' z z' z''. C x x' x'' y y' y'' z z' z'' \implies P y y' y''$ 
 $\implies P z z' z'' \implies P x x' x''$ 
  shows P a a' a''
using assms
apply (simp add: BuildSeq3-def)
apply (drule BuildSeq-induct [where P =  $\lambda(x,x',x''). P x x' x''$ ])
apply (auto intro: B C)
done

```

## 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:  $x \in y \implies \text{Rep-hf } x < \text{Rep-hf } y$ 
apply (auto simp: hmem-def hfset-def set-decode-def Abs-hf-inverse)
apply (metis div-less even-zero le-less-trans less-two-power not-less)
done

```

```

lemma hsubset-imp-le:  $x \leq y \implies \text{Rep-hf } x \leq \text{Rep-hf } y$ 
  apply (auto simp: less-eq-hf-def hmem-def hfset-def Abs-hf-inverse)
  apply (cases x rule: Abs-hf-cases)
  apply (cases y rule: Abs-hf-cases, auto)
  apply (rule subset-decode-imp-le)
  apply (auto simp: Abs-hf-inverse [OF UNIV-I])
  apply (metis Abs-hf-inverse UNIV-I imageE imageI)
done

```

```

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

```

```

definition least :: hf  $\Rightarrow$  hf
  where least a  $\equiv$  (THE x.  $x \in a \wedge (\forall y. y \in a \longrightarrow \text{Rep-hf } x \leq \text{Rep-hf } y)$ )

```

```

lemma least-equality:
  assumes  $x \in a$  and  $\bigwedge y. y \in a \implies \text{Rep-hf } x \leq \text{Rep-hf } y$ 
  shows least a = x
unfolding least-def
apply (rule the-equality)

```

```

apply (metis assms)
apply (metis Rep-hf-inverse assms eq-iff)
done

```

**lemma** *leastI2-order*:

```

assumes  $x \in a$ 
and  $\bigwedge y. y \in a \implies \text{Rep-hf } x \leq \text{Rep-hf } y$ 
and  $\bigwedge z. z \in a \implies \forall y. y \in a \longrightarrow \text{Rep-hf } z \leq \text{Rep-hf } y \implies Q z$ 
shows  $Q (\text{least } a)$ 
by (metis assms least-equality)

```

**lemma** *nonempty-imp-ex-least*:  $a \neq 0 \implies \exists x. x \in a \wedge (\forall y. y \in a \longrightarrow \text{Rep-hf } x \leq \text{Rep-hf } y)$

**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 dual-order.trans eq-iff hinsert.IH(2) hmem-hinsert less-eq-insert1-iff linear*)

**qed**

**qed**

**lemma** *least-hmem*:  $a \neq 0 \implies \text{least } a \in a$

**apply** (*frule nonempty-imp-ex-least, clarify*)

**apply** (*rule leastI2-order, auto*)

**done**

**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>.