

The Hereditarily Finite Sets

Lawrence C. Paulson

February 6, 2026

Abstract

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

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

Contents

1	The Hereditarily Finite Sets	3
1.1	Basic Definitions and Lemmas	3
1.2	Verifying the Axioms of HF	5
1.3	Ordered Pairs, from ZF/ZF.thy	5
1.4	Unions, Comprehensions, Intersections	7
1.4.1	Unions	7
1.4.2	Set comprehensions	7
1.4.3	Union operators	7
1.4.4	Definition 1.8, Intersections	8
1.4.5	Set Difference	9
1.5	Replacement	9
1.6	Subset relation and the Lattice Properties	11
1.6.1	Rules for subsets	11
1.6.2	Lattice properties	12
1.7	Foundation, Cardinality, Powersets	13
1.7.1	Foundation	13
1.7.2	Cardinality	14
1.7.3	Powerset Operator	14
1.8	Bounded Quantifiers	15
1.9	Relations and Functions	17
1.10	Operations on families of sets	18
1.10.1	Rules for Unions and Intersections of families	19
1.10.2	Generalized Cartesian product	20
1.11	Disjoint Sum	21
2	Ordinals, Sequences and Ordinal Recursion	24
2.1	Ordinals	24
2.1.1	Basic Definitions	24
2.1.2	Definition 2.2 (Successor).	24
2.1.3	Induction, Linearity, etc.	26
2.1.4	Supremum and Infimum	27
2.1.5	Converting Between Ordinals and Natural Numbers	28
2.2	Sequences and Ordinal Recursion	29

3	V-Sets, Epsilon Closure, Ranks	33
3.1	V-sets	33
3.2	Least Ordinal Operator	34
3.3	Rank Function	34
3.4	Epsilon Closure	35
3.5	Epsilon-Recursion	36
4	An Application: Finite Automata	38
5	Addition, Sequences and their Concatenation	44
5.1	Generalised Addition — Also for Ordinals	44
5.1.1	Cancellation laws for addition	45
5.1.2	The predecessor function	45
5.2	A Concatenation Operation for Sequences	46
5.3	Nonempty sequences indexed by ordinals	47
5.3.1	Sequence-building operators	49
5.3.2	Showing that Sequences can be Constructed	49
5.3.3	Proving Properties of Given Sequences	50
5.4	A Unique Predecessor for every non-empty set	52

Chapter 1

The Hereditarily Finite Sets

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

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

1.1 Basic Definitions and Lemmas

```
typedef hf = UNIV :: nat set ⟨proof⟩
```

```
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 $\langle proof \rangle$
end

lemma *Abs-hf-0* [*simp*]: $Abs\text{-}hf\ 0 = 0$
 $\langle proof \rangle$

HF Set enumerations

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

syntax (*ASCII*)
 $-HF\inset :: args \Rightarrow hf$ ($\triangleleft\{|(-)|\}\triangleright$)

syntax
 $-HF\inset :: args \Rightarrow hf$ ($\triangleleft\{|-\}\triangleright$)

syntax-consts
 $-HF\inset \rightleftharpoons inserthf$

translations
 $\{|x, y\} \rightleftharpoons \{|y\} \triangleleft x$
 $\{|x\} \rightleftharpoons 0 \triangleleft x$

lemma *finite-hfset* [*simp*]: $finite\ (hfset\ a)$
 $\langle proof \rangle$

lemma *HF-hfset* [*simp*]: $HF\ (hfset\ a) = a$
 $\langle proof \rangle$

lemma *hfset-HF* [*simp*]: $finite\ A \implies hfset\ (HF\ A) = A$
 $\langle proof \rangle$

lemma *inj-on-HF*: $inj\text{-}on\ HF\ (Collect\ finite)$
 $\langle proof \rangle$

lemma *hmem-hempty* [*simp*]: $a \notin 0$
 $\langle proof \rangle$

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

lemma *hmem-hinsert* [*iff*]:
 $hmem\ a\ (c \triangleleft b) \longleftrightarrow a = b \vee a \in c$
 $\langle proof \rangle$

lemma *hf-ext*: $a = b \longleftrightarrow (\forall x. x \in a \longleftrightarrow x \in b)$
 $\langle proof \rangle$

lemma *finite-cases* [*consumes 1, case-names empty insert*]:
 $\llbracket finite\ F; F = \{\} \implies P; \bigwedge A\ x. \llbracket F = insert\ x\ A; x \notin A; finite\ A \rrbracket \implies P \rrbracket \implies P$
 $\langle proof \rangle$

lemma *hf-cases* [*cases type: hf, case-names 0 hinsert*]:
obtains $y = 0 \mid a \triangleleft b$ **where** $y = b \triangleleft a$ **and** $a \notin b$
 ⟨*proof*⟩

lemma *Rep-hf-hinsert*:
assumes $a \notin b$ **shows** $\text{Rep-hf } (\text{hinsert } a \ b) = 2 \hat{\ } (\text{Rep-hf } a) + \text{Rep-hf } b$
 ⟨*proof*⟩

1.2 Verifying the Axioms of HF

HF1

lemma *empty-iff*: $z=0 \iff (\forall x. x \notin z)$
 ⟨*proof*⟩

HF2

lemma *hinsert-iff*: $z = x \triangleleft y \iff (\forall u. u \in z \iff u \in x \vee u = y)$
 ⟨*proof*⟩

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*⟩

HF3

lemma *hf-induct-ax*: $\llbracket P \ 0; \forall x. P \ x \longrightarrow (\forall y. P \ y \longrightarrow P \ (x \triangleleft y)) \rrbracket \implies P \ x$
 ⟨*proof*⟩

lemma *hf-equalityI* [*intro*]: $(\bigwedge x. x \in a \iff x \in b) \implies a = b$
 ⟨*proof*⟩

lemma *hinsert-nonempty* [*simp*]: $A \triangleleft a \neq 0$
 ⟨*proof*⟩

lemma *hinsert-commute*: $(z \triangleleft y) \triangleleft x = (z \triangleleft x) \triangleleft y$
 ⟨*proof*⟩

lemma *hmem-HF-iff* [*simp*]: $x \in \text{HF } A \iff x \in A \wedge \text{finite } A$
 ⟨*proof*⟩

1.3 Ordered Pairs, from ZF/ZF.thy

lemma *singleton-eq-iff* [*iff*]: $\{\!\{a}\!\} = \{\!\{b}\!\} \iff a=b$
 ⟨*proof*⟩

lemma *doubleton-eq-iff*: $\{\!\{a,b}\!\} = \{\!\{c,d}\!\} \iff (a=c \wedge b=d) \vee (a=d \wedge b=c)$

$\langle proof \rangle$

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*)

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

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

syntax

$:: hf \Rightarrow hfs \quad (\langle - \rangle)$

$-Enum :: [hf, hfs] \Rightarrow hfs \quad (\langle -, / - \rangle)$

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

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

syntax-consts

$-Enum\ -Tuple \Rightarrow hpair\ \mathbf{and}$

$-hpattern \Rightarrow hsplit$

translations

$\langle x, y, z \rangle \Rightarrow \langle x, \langle y, z \rangle \rangle$

$\langle x, y \rangle \Rightarrow CONST\ hpair\ x\ y$

$\langle x, y, z \rangle \Rightarrow \langle x, \langle y, z \rangle \rangle$

$\lambda \langle x, y, zs \rangle.\ b \Rightarrow CONST\ hsplit(\lambda x\ \langle y, zs \rangle.\ b)$

$\lambda \langle x, y \rangle.\ b \Rightarrow CONST\ hsplit(\lambda x\ y.\ b)$

lemma $hpair\text{-def}'$: $hpair\ a\ b = \{\{a, a\}, \{a, b\}\}$
 $\langle proof \rangle$

lemma $hpair\text{-iff}$ [*simp*]: $hpair\ a\ b = hpair\ a'\ b' \longleftrightarrow a = a' \wedge b = b'$
 $\langle proof \rangle$

lemmas $hpair\text{-inject} = hpair\text{-iff}$ [*THEN iffD1, THEN conjE, elim!*]

lemma $hfst\text{-conv}$ [*simp*]: $hfst\ \langle a, b \rangle = a$
 $\langle proof \rangle$

lemma $hsnd\text{-conv}$ [*simp*]: $hsnd\ \langle a, b \rangle = b$
 $\langle proof \rangle$

lemma *hsplit* [*simp*]: *hsplit* *c* $\langle a, b \rangle = c\ a\ b$
 \langle *proof* \rangle

1.4 Unions, Comprehensions, Intersections

1.4.1 Unions

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

lemma *binary-union*: $\exists z. \forall u. u \in z \longleftrightarrow u \in x \vee u \in y$
 \langle *proof* \rangle

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

lemma *union-of-set*: $\exists z. \forall u. u \in z \longleftrightarrow (\exists y. y \in x \wedge u \in y)$
 \langle *proof* \rangle

1.4.2 Set comprehensions

Theorem 1.7, comprehension scheme

lemma *comprehension*: $\exists z. \forall u. u \in z \longleftrightarrow u \in x \wedge P\ u$
 \langle *proof* \rangle

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* ($\langle (1\ \{\!-\} \langle \!-\! / \ \!-\! / \ \!-\! \} \rangle)$)

syntax

-HCollect :: *idt* \Rightarrow *hf* \Rightarrow *bool* \Rightarrow *hf* ($\langle (1\ \{\!-\} \in / \ \!-\! / \ \!-\! \} \rangle)$)

syntax-consts

-HCollect \rightleftharpoons *HCollect*

translations

$\{\!x \in A. P\!\} \rightleftharpoons \text{CONST } HCollect\ (\lambda x. P)\ A$

lemma *HCollect-iff* [*iff*]: *hmem* *x* (*HCollect* *P* *A*) $\longleftrightarrow P\ x \wedge x \in A$
 \langle *proof* \rangle

lemma *HCollectI*: $a \in A \Longrightarrow P\ a \Longrightarrow hmem\ a\ \{\!x \in A. P\ x\!\}$
 \langle *proof* \rangle

lemma *HCollectE*:

assumes $a \in \{\!x \in A. P\ x\!\}$ **obtains** $a \in A\ P\ a$
 \langle *proof* \rangle

lemma *HCollect-empty* [*simp*]: *HCollect* *P* *0* = *0*
 \langle *proof* \rangle

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 $\langle proof \rangle$
end

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

lemma $hunion\text{-iff}$ [iff]: $hmem\ x\ (a \sqcup b) \longleftrightarrow x \in a \vee x \in b$
 $\langle proof \rangle$

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

lemma $HUnion\text{-iff}$ [iff]: $hmem\ x\ (\bigsqcup\ A) \longleftrightarrow (\exists y.\ y \in A \wedge x \in y)$
 $\langle proof \rangle$

lemma $HUnion\text{-hempty}$ [simp]: $\bigsqcup\ 0 = 0$
 $\langle proof \rangle$

lemma $HUnion\text{-hinsert}$ [simp]: $\bigsqcup\ (A \triangleleft a) = a \sqcup \bigsqcup\ A$
 $\langle proof \rangle$

lemma $HUnion\text{-hunion}$ [simp]: $\bigsqcup\ (A \sqcup B) = \bigsqcup\ A \sqcup \bigsqcup\ B$
 $\langle proof \rangle$

1.4.4 Definition 1.8, Intersections

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

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

lemma $hinter\text{-iff}$ [iff]: $hmem\ u\ (x \sqcap y) \longleftrightarrow u \in x \wedge u \in y$
 $\langle proof \rangle$

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

lemma $HInter\text{-hempty}$ [iff]: $\sqcap\ 0 = 0$
 $\langle proof \rangle$

lemma $HInter\text{-iff}$ [simp]: $A \neq 0 \implies hmem\ x\ (\sqcap\ A) \longleftrightarrow (\forall y.\ y \in A \longrightarrow x \in y)$
 $\langle proof \rangle$

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

1.4.5 Set Difference

instantiation *hf* :: *minus*

begin

definition $A - B = \{x \in A. x \notin B\}$

instance ⟨*proof*⟩

end

lemma *hdiff-iff* [*iff*]: $hmem\ u\ (x - y) \longleftrightarrow u \in x \wedge u \notin y$
 ⟨*proof*⟩

lemma *hdiff-zero* [*simp*]: **fixes** $x :: hf$ **shows** $(x - 0) = x$
 ⟨*proof*⟩

lemma *zero-hdiff* [*simp*]: **fixes** $x :: hf$ **shows** $(0 - x) = 0$
 ⟨*proof*⟩

lemma *hdiff-insert*: $A - (B \triangleleft a) = A - B - \{a\}$
 ⟨*proof*⟩

lemma *hinsert-hdiff-if*:

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

⟨*proof*⟩

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*⟩

lemma *replacement-fun*: $\exists z. \forall v. v \in z \longleftrightarrow (\exists u. u \in x \wedge v = f\ u)$
 ⟨*proof*⟩

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)

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

syntax

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

syntax-consts

$-HReplace \rightleftharpoons Replace \text{ and}$
 $-HRepFun \rightleftharpoons RepFun \text{ and}$
 $-HINTER \rightleftharpoons HInter \text{ and}$
 $-HUNION \rightleftharpoons HUnion$

translations

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

lemma PrimReplace-iff:

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

lemma Replace-iff [iff]:

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

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma Replace-cong [cong]:

$\llbracket A=B; \bigwedge x y. x \in B \implies P x y \longleftrightarrow Q x y \rrbracket \implies Replace A P = Replace B Q$
 $\langle proof \rangle$

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

$\langle proof \rangle$

lemma RepFun-cong [cong]:

$\llbracket A=B; \bigwedge x. x \in B \implies f(x)=g(x) \rrbracket \implies RepFun A f = RepFun B g$
 $\langle proof \rangle$

lemma triv-RepFun [simp]: RepFun A ($\lambda x. x$) = A

<proof>

lemma *RepFun-0* [*simp*]: $\text{RepFun } 0 \ f = 0$
<proof>

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

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

lemma *HF-HUnion*: $\llbracket \text{finite } A; \bigwedge x. x \in A \implies \text{finite } (B \ x) \rrbracket \implies \text{HF } (\bigcup x \in A. B \ x)$
 $= (\bigsqcup x \in \text{HF } A. \text{HF } (B \ x))$
<proof>

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 *<proof>*

end

1.6.1 Rules for subsets

lemma *hsubsetI* [*intro!*]:
 $(\bigwedge x. x \in A \implies x \in B) \implies A \leq B$
<proof>

Classical elimination rule

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

Rule in Modus Ponens style

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

Sometimes useful with premises in this order

lemma *rev-hsubsetD*: $\llbracket c \in A; A \leq B \rrbracket \implies c \in B$
<proof>

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

<proof>

lemma *rev-contr-hsubsetD*: $\llbracket c \notin B; A \leq B \rrbracket \implies c \notin A$
<proof>

lemma *hf-equalityE*:

fixes $A :: hf$ **shows** $A = B \implies (A \leq B \implies B \leq A \implies P) \implies P$
<proof>

1.6.2 Lattice properties

instantiation *hf* :: *distrib-lattice*

begin

instance *<proof>*

end

instantiation *hf* :: *bounded-lattice-bot*

begin

definition *bot* = $(0::hf)$

instance *<proof>*

end

lemma *hinter-hempty-left* [*simp*]: $0 \sqcap A = 0$
<proof>

lemma *hinter-hempty-right* [*simp*]: $A \sqcap 0 = 0$
<proof>

lemma *hunion-hempty-left* [*simp*]: $0 \sqcup A = A$
<proof>

lemma *hunion-hempty-right* [*simp*]: $A \sqcup 0 = A$
<proof>

lemma *less-eq-hempty* [*simp*]: $u \leq 0 \longleftrightarrow u = (0::hf)$
<proof>

lemma *less-eq-insert1-iff* [*iff*]: $(hinsert\ x\ y) \leq z \longleftrightarrow x \in z \wedge y \leq z$
<proof>

lemma *less-eq-insert2-iff*:

$z \leq (hinsert\ x\ y) \longleftrightarrow z \leq y \vee (\exists u. hinsert\ x\ u = z \wedge x \notin u \wedge u \leq y)$
<proof>

lemma *zero-le* [*simp*]: $0 \leq (x::hf)$
<proof>

lemma *hinsert-eq-sup*: $b \triangleleft a = b \sqcup \{a\}$
<proof>

lemma *hunion-hinsert-left*: $\text{hinsert } x \ A \sqcup B = \text{hinsert } x \ (A \sqcup B)$
<proof>

lemma *hunion-hinsert-right*: $B \sqcup \text{hinsert } x \ A = \text{hinsert } x \ (B \sqcup A)$
<proof>

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

lemma *hinter-hinsert-right*: $B \sqcap \text{hinsert } x \ A = (\text{if } x \in B \text{ then } \text{hinsert } x \ (B \sqcap A) \text{ else } B \sqcap A)$
<proof>

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>

lemma *hmem-not-refl*: $x \notin x$
<proof>

lemma *hmem-not-sym*: $\neg (x \in y \wedge y \in x)$
<proof>

lemma *hmem-ne*: $x \in y \implies x \neq y$
<proof>

lemma *hmem-Sup-ne*: $x \in y \implies \sqcup x \neq y$
<proof>

lemma *hpair-neq-fst*: $\langle a, b \rangle \neq a$
<proof>

lemma *hpair-neq-snd*: $\langle a, b \rangle \neq b$
<proof>

lemma *hpair-nonzero [simp]*: $\langle x, y \rangle \neq 0$
<proof>

lemma *zero-notin-hpair*: $0 \notin \langle x, y \rangle$
<proof>

1.7.2 Cardinality

First we need to hack the underlying representation

lemma *hfset-0* [*simp*]: $hfset\ 0 = \{\}$
<proof>

lemma *hfset-hinsert*: $hfset\ (b \triangleleft a) = insert\ a\ (hfset\ b)$
<proof>

lemma *hfset-hdiff*: $hfset\ (x - y) = hfset\ x - hfset\ y$
<proof>

definition *hcard* :: $hf \Rightarrow nat$
where $hcard\ x = card\ (hfset\ x)$

lemma *hcard-0* [*simp*]: $hcard\ 0 = 0$
<proof>

lemma *hcard-hinsert-if*: $hcard\ (hinsert\ x\ y) = (if\ x \in y\ then\ hcard\ y\ else\ Suc\ (hcard\ y))$
<proof>

lemma *hcard-union-inter*: $hcard\ (x \sqcup y) + hcard\ (x \sqcap y) = hcard\ x + hcard\ y$
<proof>

lemma *hcard-hdiff1-less*: $x \in z \implies hcard\ (z - \{x\}) < hcard\ z$
<proof>

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>

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

lemma *HPow-iff* [*iff*]: $u \in HPow\ x \longleftrightarrow u \leq x$
<proof>

lemma *HPow-mono*: $x \leq y \implies HPow\ x \leq HPow\ y$
<proof>

lemma *HPow-mono-strict*: $x < y \implies HPow\ x < HPow\ y$
<proof>

lemma *HPow-mono-iff* [*simp*]: $HPow\ x \leq HPow\ y \longleftrightarrow x \leq y$
<proof>

lemma *HPow-mono-strict-iff* [*simp*]: $HPow\ x < HPow\ y \longleftrightarrow x < y$
 ⟨*proof*⟩

1.8 Bounded Quantifiers

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

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

syntax (*ASCII*)

-*HBall* :: $pttrn \Rightarrow hf \Rightarrow bool \Rightarrow bool$ ⟨($\exists ALL$ -<:./ -)⟩ [0, 0, 10] 10
 -*HBex* :: $pttrn \Rightarrow hf \Rightarrow bool \Rightarrow bool$ ⟨($\exists EX$ -<:./ -)⟩ [0, 0, 10] 10
 -*HBex1* :: $pttrn \Rightarrow hf \Rightarrow bool \Rightarrow bool$ ⟨($\exists EX!$ -<:./ -)⟩ [0, 0, 10] 10

syntax

-*HBall* :: $pttrn \Rightarrow hf \Rightarrow bool \Rightarrow bool$ ⟨($\exists \forall$ -<:./ -)⟩ [0, 0, 10] 10
 -*HBex* :: $pttrn \Rightarrow hf \Rightarrow bool \Rightarrow bool$ ⟨($\exists \exists$ -<:./ -)⟩ [0, 0, 10] 10
 -*HBex1* :: $pttrn \Rightarrow hf \Rightarrow bool \Rightarrow bool$ ⟨($\exists \exists!$ -<:./ -)⟩ [0, 0, 10] 10

syntax-consts

-*HBall* $\hat{=}$ *HBall* **and**
 -*HBex* $\hat{=}$ *HBex* **and**
 -*HBex1* $\hat{=}$ *Ex1*

translations

$\forall x \in A. P \hat{=} CONST\ HBall\ A\ (\lambda x. P)$
 $\exists x \in A. P \hat{=} CONST\ HBex\ A\ (\lambda x. P)$
 $\exists !x \in A. P \hat{=} \exists !x. x \in A \wedge P$

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

$\llbracket A=A'; \bigwedge x. x \in A' \Longrightarrow P(x) \longleftrightarrow P'(x) \rrbracket \Longrightarrow (\forall x \in A. P(x)) \longleftrightarrow (\forall x \in A'. P'(x))$
 ⟨*proof*⟩

lemma *hballI* [*intro!*]: $(\bigwedge x. x \in A \Longrightarrow P\ x) \Longrightarrow \forall x \in A. P\ x$
 ⟨*proof*⟩

lemma *hbspec* [*dest?*]: $\forall x \in A. P\ x \Longrightarrow x \in A \Longrightarrow P\ x$
 ⟨*proof*⟩

lemma *hballE* [*elim*]: $\forall x \in A. P\ x \Longrightarrow (P\ x \Longrightarrow Q) \Longrightarrow (x \notin A \Longrightarrow Q) \Longrightarrow Q$
 ⟨*proof*⟩

lemma *hbex-cong* [*cong*]:

$\llbracket A=A'; \bigwedge x. x \in A' \Longrightarrow P(x) \longleftrightarrow P'(x) \rrbracket \Longrightarrow (\exists x \in A. P(x)) \longleftrightarrow (\exists x \in A'. P'(x))$
 ⟨*proof*⟩

lemma *hbexI* [*intro*]: $P\ x \Longrightarrow x \in A \Longrightarrow \exists x \in A. P\ x$
and *rev-hbexI* [*intro?*]: $x \in A \Longrightarrow P\ x \Longrightarrow \exists x \in A. P\ x$

and *bexCI*: $(\forall x \in A. \neg P x \implies P a) \implies a \in A \implies \exists x \in A. P x$
and *hbexE* [*elim!*]: $\exists x \in A. P x \implies (\bigwedge x. x \in A \implies P x \implies Q) \implies Q$
<proof>

lemma *hball-triv* [*simp*]: $(\forall x \in A. P) = ((\exists x. x \in A) \longrightarrow P)$
and *hbex-triv* [*simp*]: $(\exists x \in A. P) = ((\exists x. x \in A) \wedge P)$
— Dual form for existentials.
<proof>

lemma *hbex-triv-one-point1* [*simp*]: $(\exists x \in A. x = a) = (a \in A)$
<proof>

lemma *hbex-triv-one-point2* [*simp*]: $(\exists x \in A. a = x) = (a \in A)$
<proof>

lemma *hbex-one-point1* [*simp*]: $(\exists x \in A. x = a \wedge P x) = (a \in A \wedge P a)$
<proof>

lemma *hbex-one-point2* [*simp*]: $(\exists x \in A. a = x \wedge P x) = (a \in A \wedge P a)$
<proof>

lemma *hball-one-point1* [*simp*]: $(\forall x \in A. x = a \longrightarrow P x) = (a \in A \longrightarrow P a)$
<proof>

lemma *hball-one-point2* [*simp*]: $(\forall x \in A. a = x \longrightarrow P x) = (a \in A \longrightarrow P a)$
<proof>

lemma *hball-conj-distrib*:
 $(\forall x \in A. P x \wedge Q x) \longleftrightarrow ((\forall x \in A. P x) \wedge (\forall x \in A. Q x))$
<proof>

lemma *hbex-disj-distrib*:
 $(\exists x \in A. P x \vee Q x) \longleftrightarrow ((\exists x \in A. P x) \vee (\exists x \in A. Q x))$
<proof>

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

lemma *hb-ex-simps* [*simp, no-atp*]:
 $\bigwedge A P Q. (\exists x \in A. P x \wedge Q) \longleftrightarrow ((\exists x \in A. P x) \wedge Q)$
 $\bigwedge A P Q. (\exists x \in A. P \wedge Q x) \longleftrightarrow (P \wedge (\exists x \in A. Q x))$

$\bigwedge P. (\exists x \in 0. P x) \longleftrightarrow \text{False}$
 $\bigwedge a B P. (\exists x \in B \triangleleft a. P x) \longleftrightarrow (P a \vee (\exists x \in B. P x))$
 $\bigwedge P Q. (\exists x \in \text{HCollect } Q A. P x) \longleftrightarrow (\exists x \in A. Q x \wedge P x)$
 $\bigwedge A P. (\neg(\exists x \in A. P x)) \longleftrightarrow (\forall x \in A. \neg P x)$
 $\langle \text{proof} \rangle$

lemma *le-HCollect-iff*: $A \leq \{x \in B. P x\} \longleftrightarrow A \leq B \wedge (\forall x \in A. P x)$
 $\langle \text{proof} \rangle$

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* $f x = (\text{THE } y. \langle x, y \rangle \in f)$

lemma *hrestrict-iff* [*iff*]:
 $z \in \text{hrestrict } r A \longleftrightarrow z \in r \wedge (\exists x y. z = \langle x, y \rangle \wedge x \in A)$
 $\langle \text{proof} \rangle$

lemma *hrelation-0* [*simp*]: *hrelation* 0
 $\langle \text{proof} \rangle$

lemma *hrelation-restr* [*iff*]: *hrelation* (*hrestrict* $r x$)
 $\langle \text{proof} \rangle$

lemma *hrelation-hunion* [simp]: $hrelation (f \sqcup g) \longleftrightarrow hrelation f \wedge hrelation g$
 ⟨proof⟩

lemma *hfunction-restr*: $hfunction r \implies hfunction (hrestrict r x)$
 ⟨proof⟩

lemma *hdomain-restr* [simp]: $hdomain (hrestrict r x) = hdomain r \sqcap x$
 ⟨proof⟩

lemma *hdomain-0* [simp]: $hdomain 0 = 0$
 ⟨proof⟩

lemma *hdomain-ins* [simp]: $hdomain (r \triangleleft \langle x, y \rangle) = hdomain r \triangleleft x$
 ⟨proof⟩

lemma *hdomain-hunion* [simp]: $hdomain (f \sqcup g) = hdomain f \sqcup hdomain g$
 ⟨proof⟩

lemma *hdomain-not-mem* [iff]: $\langle hdomain r, a \rangle \notin r$
 ⟨proof⟩

lemma *app-singleton* [simp]: $app \{\langle x, y \rangle\} x = y$
 ⟨proof⟩

lemma *app-equality*: $hfunction f \implies \langle x, y \rangle \in f \implies app f x = y$
 ⟨proof⟩

lemma *app-ins2*: $x' \neq x \implies app (f \triangleleft \langle x, y \rangle) x' = app f x'$
 ⟨proof⟩

lemma *hfunction-0* [simp]: $hfunction 0$
 ⟨proof⟩

lemma *hfunction-ins*: $hfunction f \implies x \notin hdomain f \implies hfunction (f \triangleleft \langle x, y \rangle)$
 ⟨proof⟩

lemma *hdomainI*: $\langle x, y \rangle \in f \implies x \in hdomain f$
 ⟨proof⟩

lemma *hfunction-hunion*: $hdomain f \sqcap hdomain g = 0$
 $\implies hfunction (f \sqcup g) \longleftrightarrow hfunction f \wedge hfunction g$
 ⟨proof⟩

lemma *app-hrestrict* [simp]: $x \in A \implies app (hrestrict f A) x = app f x$
 ⟨proof⟩

1.10 Operations on families of sets

definition *HLambda* :: $hf \Rightarrow (hf \Rightarrow hf) \Rightarrow hf$

where $HLambda A b = RepFun A (\lambda x. \langle x, b x \rangle)$

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

definition $HPi :: hf \Rightarrow (hf \Rightarrow hf) \Rightarrow hf$
 where $HPi A B = \{f \in HPow(HSigma A B). A \leq hdomain(f) \wedge hfunction(f)\}$

syntax (*ASCII*)

-*PROD* :: $[pttrn, hf, hf] \Rightarrow hf$ $\langle (\exists PROD \text{ -<:./ -}) \rangle 10$
 -*SUM* :: $[pttrn, hf, hf] \Rightarrow hf$ $\langle (\exists SUM \text{ -<:./ -}) \rangle 10$
 -*lam* :: $[pttrn, hf, hf] \Rightarrow hf$ $\langle (\exists lam \text{ -<:./ -}) \rangle 10$

syntax

-*PROD* :: $[pttrn, hf, hf] \Rightarrow hf$ $\langle (\exists \prod \text{ -}\in\text{./ -}) \rangle 10$
 -*SUM* :: $[pttrn, hf, hf] \Rightarrow hf$ $\langle (\exists \sum \text{ -}\in\text{./ -}) \rangle 10$
 -*lam* :: $[pttrn, hf, hf] \Rightarrow hf$ $\langle (\exists \lambda \text{ -}\in\text{./ -}) \rangle 10$

syntax-consts

-*PROD* $\hat{=}$ *HPi* **and**
 -*SUM* $\hat{=}$ *HSigma* **and**
 -*lam* $\hat{=}$ *HLambda*

translations

$\prod x \in A. B \hat{=}$ *CONST HPi* $A (\lambda x. B)$
 $\sum x \in A. B \hat{=}$ *CONST HSigma* $A (\lambda x. B)$
 $\lambda x \in A. f \hat{=}$ *CONST HLambda* $A (\lambda x. f)$

1.10.1 Rules for Unions and Intersections of families

lemma *HUN-iff* [*simp*]: $b \in (\bigsqcup x \in A. B(x)) \longleftrightarrow (\exists x \in A. b \in B(x))$
 $\langle proof \rangle$

lemma *HUN-I*: $\llbracket a \in A; b \in B(a) \rrbracket \Longrightarrow b \in (\bigsqcup x \in A. B(x))$
 $\langle proof \rangle$

lemma *HUN-E* [*elim!*]: **assumes** $b \in (\bigsqcup x \in A. B(x))$ **obtains** x **where** $x \in A$ $b \in B(x)$
 $\langle proof \rangle$

lemma *HINT-iff*: $b \in (\prod x \in A. B(x)) \longleftrightarrow (\forall x \in A. b \in B(x)) \wedge A \neq 0$
 $\langle proof \rangle$

lemma *HINT-I*: $\llbracket \bigwedge x. x \in A \Longrightarrow b \in B(x); A \neq 0 \rrbracket \Longrightarrow b \in (\prod x \in A. B(x))$
 $\langle proof \rangle$

lemma *HINT-E*: $\llbracket b \in (\prod x \in A. B(x)); a \in A \rrbracket \Longrightarrow b \in B(a)$
 $\langle proof \rangle$

1.10.2 Generalized Cartesian product

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

lemma *HSigmaI* [*intro!*]: $\llbracket a \in A; b \in B(a) \rrbracket \implies \langle a, b \rangle \in \text{HSigma } A \ B$
 $\langle \text{proof} \rangle$

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** $x \in A \ y \in B(x) \ c = \langle x, y \rangle$
 $\langle \text{proof} \rangle$

lemma *HSigmaE2* [*elim!*]:
assumes $\langle a, b \rangle \in \text{HSigma } A \ B$ **obtains** $a \in A$ **and** $b \in B(a)$
 $\langle \text{proof} \rangle$

lemma *HSigma-empty1* [*simp*]: $\text{HSigma } 0 \ B = 0$
 $\langle \text{proof} \rangle$

instantiation *hf* :: *times*

begin

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

instance $\langle \text{proof} \rangle$

end

lemma *times-iff* [*simp*]: $\langle a, b \rangle \in A * B \longleftrightarrow a \in A \wedge b \in B$
 $\langle \text{proof} \rangle$

lemma *timesI* [*intro!*]: $\llbracket a \in A; b \in B \rrbracket \implies \langle a, b \rangle \in A * B$
 $\langle \text{proof} \rangle$

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** $x \in A \ y \in B \ c = \langle x, y \rangle$ $\langle \text{proof} \rangle$

...and a specific one

lemma *timesE2* [*elim!*]:
assumes $\langle a, b \rangle \in A * B$ **obtains** $a \in A$ **and** $b \in B$
 $\langle \text{proof} \rangle$

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

<proof>

lemma *times-empty2* [simp]: $A * 0 = (0 :: hf)$
<proof>

lemma *times-empty-iff*: $A * B = 0 \longleftrightarrow A = 0 \vee B = (0 :: hf)$
<proof>

instantiation *hf* :: *mult-zero*
begin
 instance *<proof>*
end

1.11 Disjoint Sum

instantiation *hf* :: *zero-neq-one*
begin
 definition *One-hf-def*: $1 = \{0\}$
 instance *<proof>*
end

instantiation *hf* :: *plus*
begin
 definition $A + B = (\{0\} * A) \sqcup (\{1\} * B)$
 instance *<proof>*
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]: $Inl\ x \neq 0$
<proof>

lemma *Inr-nonzero* [simp]: $Inr\ x \neq 0$
<proof>

Introduction rules for the injections (as equivalences)

lemma *Inl-in-sum-iff* [iff]: $Inl(a) \in A + B \longleftrightarrow a \in A$
<proof>

lemma *Inr-in-sum-iff* [iff]: $Inr(b) \in A + B \longleftrightarrow b \in B$
<proof>

Elimination rule

lemma *sumE* [*elim!*]:

assumes $u: u \in A+B$

obtains x **where** $x \in A \ u=Inl(x) \mid y$ **where** $y \in B \ u=Inr(y)$ *<proof>*

Injection and freeness equivalences, for rewriting

lemma *Inl-iff* [*iff*]: $Inl(a)=Inl(b) \longleftrightarrow a=b$

<proof>

lemma *Inr-iff* [*iff*]: $Inr(a)=Inr(b) \longleftrightarrow a=b$

<proof>

lemma *Inl-Inr-iff* [*iff*]: $Inl(a)=Inr(b) \longleftrightarrow False$

<proof>

lemma *Inr-Inl-iff* [*iff*]: $Inr(b)=Inl(a) \longleftrightarrow False$

<proof>

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

<proof>

lemma *sum-iff*: $u \in A+B \longleftrightarrow (\exists x. x \in A \wedge u=Inl(x)) \vee (\exists y. y \in B \wedge u=Inr(y))$

<proof>

lemma *sum-subset-iff*:

fixes $A :: hf$ **shows** $A+B \leq C+D \longleftrightarrow A \leq C \wedge B \leq D$

<proof>

lemma *sum-equal-iff*:

fixes $A :: hf$ **shows** $A+B = C+D \longleftrightarrow A=C \wedge B=D$

<proof>

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*]: $sum-case\ f\ g\ (Inl\ x) = f\ x$

<proof>

lemma *sum-case-Inr* [*simp*]: $sum-case\ f\ g\ (Inr\ y) = g\ y$

<proof>

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

<proof>

lemma *is-hsum-cases*: $(\exists x. z = \text{Inl } x \vee z = \text{Inr } x) \vee \neg \text{is-hsum } z$
<proof>

lemma *sum-case-split*:

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

lemma *sum-case-split-asm*:

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

Expose standard *Inl* and *Inr* on sum types:

hide-const (**open**) *HF.Inl HF.Inr*

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*: $Transset\ x \Longrightarrow Transset\ y \Longrightarrow Transset\ (x \sqcup y)$
<proof>

lemma *Transset-inf*: $Transset\ x \Longrightarrow Transset\ y \Longrightarrow Transset\ (x \sqcap y)$
<proof>

lemma *Transset-hinsert*: $Transset\ x \Longrightarrow y \leq x \Longrightarrow Transset\ (x \triangleleft y)$
<proof>

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

lemma *succ-iff* [*simp*]: $x \in \text{succ } y \longleftrightarrow x=y \vee x \in y$
 ⟨*proof*⟩

lemma *succ-ne-self* [*simp*]: $i \neq \text{succ } i$
 ⟨*proof*⟩

lemma *succ-notin-self*: $\text{succ } i \notin i$
 ⟨*proof*⟩

lemma *succE* [*elim?*]: **assumes** $x \in \text{succ } y$ **obtains** $x=y \mid x \in y$
 ⟨*proof*⟩

lemma *hmem-succ-ne*: $\text{succ } x \in y \implies x \neq y$
 ⟨*proof*⟩

lemma *hball-succ* [*simp*]: $(\forall x \in \text{succ } k. P x) \longleftrightarrow P k \wedge (\forall x \in k. P x)$
 ⟨*proof*⟩

lemma *hbex-succ* [*simp*]: $(\exists x \in \text{succ } k. P x) \longleftrightarrow P k \vee (\exists x \in k. P x)$
 ⟨*proof*⟩

lemma *One-hf-eq-succ*: $1 = \text{succ } 0$
 ⟨*proof*⟩

lemma *zero-hmem-one* [*iff*]: $x \in 1 \longleftrightarrow x = 0$
 ⟨*proof*⟩

lemma *hball-One* [*simp*]: $(\forall x \in 1. P x) = P 0$
 ⟨*proof*⟩

lemma *hbex-One* [*simp*]: $(\exists x \in 1. P x) = P 0$
 ⟨*proof*⟩

lemma *hpair-neq-succ* [*simp*]: $\langle x, y \rangle \neq \text{succ } k$
 ⟨*proof*⟩

lemma *succ-neq-hpair* [*simp*]: $\text{succ } k \neq \langle x, y \rangle$
 ⟨*proof*⟩

lemma *hpair-neq-one* [*simp*]: $\langle x, y \rangle \neq 1$
 ⟨*proof*⟩

lemma *one-neq-hpair* [*simp*]: $1 \neq \langle x, y \rangle$
 ⟨*proof*⟩

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

<proof>

lemma *hmem-succ*: $l \in k \implies l \in \text{succ } k$

<proof>

Theorem 2.3.

lemma *Ord-0 [iff]*: *Ord 0*

<proof>

lemma *Ord-succ*: $\text{Ord}(k) \implies \text{Ord}(\text{succ}(k))$

<proof>

lemma *Ord-1 [iff]*: *Ord 1*

<proof>

lemma *OrdmemD*: $\text{Ord}(k) \implies j \in k \implies j \leq k$

<proof>

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

<proof>

lemma *hmem-0-Ord*:

assumes *k*: $\text{Ord}(k)$ **and** *knz*: $k \neq 0$ **shows** $0 \in k$

<proof>

lemma *Ord-in-Ord*: $\llbracket \text{Ord}(k); m \in k \rrbracket \implies \text{Ord}(m)$

<proof>

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>

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>

The trichotomy law for ordinals

lemma *Ord-linear-lt*:

assumes *o*: $\text{Ord}(k)$ $\text{Ord}(l)$

obtains (*lt*) $k \in l$ | (*eq*) $k=l$ | (*gt*) $l \in k$

<proof>

lemma *Ord-linear2*:

assumes *o*: $\text{Ord}(k)$ $\text{Ord}(l)$

obtains (*lt*) $k \in l$ | (*ge*) $l \leq k$

<proof>

lemma *Ord-linear-le*:

assumes $o: \text{Ord}(k) \ \text{Ord}(l)$

obtains $(le) \ k \leq l \mid (ge) \ l \leq k$

<proof>

lemma *hunion-less-iff* [simp]: $[[\text{Ord } i; \text{Ord } j]] \implies i \sqcup j < k \longleftrightarrow i < k \wedge j < k$

<proof>

Theorem 2.5

lemma *Ord-mem-iff-lt*: $\text{Ord}(k) \implies \text{Ord}(l) \implies k \in l \longleftrightarrow k < l$

<proof>

lemma *le-succE*: $\text{succ } i \leq \text{succ } j \implies i \leq j$

<proof>

lemma *le-succ-iff*: $\text{Ord } i \implies \text{Ord } j \implies \text{succ } i \leq \text{succ } j \longleftrightarrow i \leq j$

<proof>

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

<proof>

lemma *mem-succ-iff* [simp]: $\text{Ord } j \implies \text{succ } i \in \text{succ } j \longleftrightarrow i \in j$

<proof>

lemma *Ord-mem-succ-cases*:

assumes $\text{Ord}(k) \ l \in k$

shows $\text{succ } l = k \vee \text{succ } l \in k$

<proof>

2.1.4 Supremum and Infimum

lemma *Ord-Union* [intro,simp]: $[[\bigwedge i. i \in A \implies \text{Ord}(i)]] \implies \text{Ord}(\bigsqcup A)$

<proof>

lemma *Ord-Inter* [intro,simp]:

assumes $\bigwedge i. i \in A \implies \text{Ord}(i)$ **shows** $\text{Ord}(\bigsqcap A)$

<proof>

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

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

<proof>

lemma *hmem-Inf-Ords*: $[[A \neq 0; \bigwedge i. i \in A \implies \text{Ord}(i)]] \implies \bigsqcap A \in A$

<proof>

lemma *Ord-pred*: $[[\text{Ord}(k); k \neq 0]] \implies \text{succ}(\bigsqcup k) = k$

<proof>

lemma *Ord-cases* [*cases type: hf, case-names 0 succ*]:
 assumes *Ok: Ord(k)*
 obtains $k = 0 \mid l$ **where** *Ord l succ l = k*
<proof>

lemma *Ord-induct2* [*consumes 1, case-names 0 succ, induct type: hf*]:
 assumes *k: Ord(k)*
 and $P: P\ 0 \wedge k. Ord\ k \implies P\ k \implies P\ (succ\ k)$
 shows $P\ k$
<proof>

lemma *Ord-succ-iff* [*iff*]: $Ord\ (succ\ k) = Ord\ k$
<proof>

lemma [*simp*]: $succ\ k \neq 0$
<proof>

lemma *Ord-Sup-succ-eq* [*simp*]: $Ord\ k \implies \bigsqcup (succ\ k) = k$
<proof>

lemma *Ord-lt-succ-iff-le*: $Ord\ k \implies Ord\ l \implies k < succ\ l \iff k \leq l$
<proof>

lemma *zero-in-Ord*: $Ord\ k \implies k=0 \vee 0 \in k$
<proof>

lemma *hpair-neq-Ord*: $Ord\ k \implies \langle x,y \rangle \neq k$
<proof>

lemma *hpair-neq-Ord'*: **assumes** *k: Ord k* **shows** $k \neq \langle x,y \rangle$
<proof>

lemma *Not-Ord-hpair* [*iff*]: $\neg Ord\ \langle x,y \rangle$
<proof>

lemma *is-hpair* [*simp*]: $is-hpair\ \langle x,y \rangle$
<proof>

lemma *Ord-not-hpair*: $Ord\ x \implies \neg is-hpair\ x$
<proof>

lemma *zero-in-succ* [*simp,intro*]: $Ord\ i \implies 0 \in succ\ i$
<proof>

2.1.5 Converting Between Ordinals and Natural Numbers

fun *ord-of* :: $nat \Rightarrow hf$

where

$ord\text{-of } 0 = 0$
| $ord\text{-of } (Suc\ k) = succ\ (ord\text{-of } k)$

lemma *Ord-ord-of [simp]: Ord (ord-of k)*
<proof>

lemma *ord-of-inject [iff]: ord-of i = ord-of j \longleftrightarrow i=j*
<proof>

lemma *ord-of-minus-1: n > 0 \implies ord-of n = succ (ord-of (n - 1))*
<proof>

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

lemma *nat-of-ord-ord-of [simp]: nat-of-ord (ord-of n) = n*
<proof>

lemma *nat-of-ord-0 [simp]: nat-of-ord 0 = 0*
<proof>

lemma *ord-of-nat-of-ord [simp]: Ord x \implies ord-of (nat-of-ord x) = x*
<proof>

lemma *nat-of-ord-inject: Ord x \implies Ord y \implies nat-of-ord x = nat-of-ord y \longleftrightarrow x = y*
<proof>

lemma *nat-of-ord-succ [simp]: Ord x \implies nat-of-ord (succ x) = Suc (nat-of-ord x)*
<proof>

lemma *inj-ord-of: inj-on ord-of A*
<proof>

lemma *hfset-ord-of: hfset (ord-of n) = ord-of ‘ {0.. $<n$ }*
<proof>

lemma *bij-betw-ord-of: bij-betw ord-of {0.. $<n$ } (hfset (ord-of n))*
<proof>

lemma *bij-betw-ord-ofI:*
bij-betw h A {0.. $<n$ } \implies bij-betw (ord-of \circ h) A (hfset (ord-of n))
<proof>

2.2 Sequences and Ordinal Recursion

Definition 3.2 (Sequence).

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

lemma $Seq-0$ [iff]: $Seq\ 0\ 0$
 ⟨proof⟩

lemma $Seq-succ-D$: $Seq\ s\ (succ\ k) \Longrightarrow Seq\ s\ k$
 ⟨proof⟩

lemma $Seq-Ord-D$: $Seq\ s\ k \Longrightarrow l \in k \Longrightarrow Ord\ k \Longrightarrow Seq\ s\ l$
 ⟨proof⟩

lemma $Seq-restr$: $Seq\ s\ (succ\ k) \Longrightarrow Seq\ (hrestrict\ s\ k)\ k$
 ⟨proof⟩

lemma $Seq-Ord-restr$: $\llbracket Seq\ s\ k; l \in k; Ord\ k \rrbracket \Longrightarrow Seq\ (hrestrict\ s\ l)\ l$
 ⟨proof⟩

lemma $Seq-ins$: $\llbracket Seq\ s\ k; k \notin hdomain\ s \rrbracket \Longrightarrow Seq\ (s \triangleleft \langle k, y \rangle)\ (succ\ k)$
 ⟨proof⟩

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

lemma $hrelation-insf$: $hrelation\ s \Longrightarrow hrelation\ (insf\ s\ k\ y)$
 ⟨proof⟩

lemma $hfunction-insf$: $hfunction\ s \Longrightarrow hfunction\ (insf\ s\ k\ y)$
 ⟨proof⟩

lemma $hdomain-insf$: $k \leq hdomain\ s \Longrightarrow succ\ k \leq hdomain\ (insf\ s\ k\ y)$
 ⟨proof⟩

lemma $Seq-insf$: $Seq\ s\ k \Longrightarrow Seq\ (insf\ s\ k\ y)\ (succ\ k)$
 ⟨proof⟩

lemma $Seq-succ-iff$: $Seq\ s\ (succ\ k) \longleftrightarrow Seq\ s\ k \wedge (\exists y. \langle k, y \rangle \in s)$
 ⟨proof⟩

lemma $nonrestrictD$: $a \in nonrestrict\ s\ X \Longrightarrow a \in s$
 ⟨proof⟩

lemma $hpair-in-nonrestrict-iff$ [simp]:
 $\langle a, b \rangle \in nonrestrict\ s\ X \longleftrightarrow \langle a, b \rangle \in s \wedge \neg a \in X$
 ⟨proof⟩

lemma $app-nonrestrict-Seq$: $Seq\ s\ k \Longrightarrow z \notin X \Longrightarrow app\ (nonrestrict\ s\ X)\ z = app\ s\ z$
 ⟨proof⟩

lemma *app-insf-Seq*: $Seq\ s\ k \implies app\ (insf\ s\ k\ y)\ k = y$
 ⟨proof⟩

lemma *app-insf2-Seq*: $Seq\ s\ k \implies k' \neq k \implies app\ (insf\ s\ k\ y)\ k' = app\ s\ k'$
 ⟨proof⟩

lemma *app-insf-Seq-if*: $Seq\ s\ k \implies app\ (insf\ s\ k\ y)\ k' = (if\ k' = k\ then\ y\ else\ app\ s\ k')$
 ⟨proof⟩

lemma *Seq-imp-eq-app*: $\llbracket Seq\ s\ d; \langle x, y \rangle \in s \rrbracket \implies app\ s\ x = y$
 ⟨proof⟩

lemma *Seq-iff-app*: $\llbracket Seq\ s\ d; x \in d \rrbracket \implies \langle x, y \rangle \in s \longleftrightarrow app\ s\ x = y$
 ⟨proof⟩

lemma *Exists-iff-app*: $Seq\ s\ d \implies x \in d \implies (\exists y. \langle x, y \rangle \in s \wedge P\ y) = P\ (app\ s\ x)$
 ⟨proof⟩

lemma *Ord-trans2*: $\llbracket i2 \in i; i \in j; j \in k; Ord\ k \rrbracket \implies i2 \in k$
 ⟨proof⟩

definition *ord-rec-Seq* :: $hf \Rightarrow (hf \Rightarrow hf) \Rightarrow hf \Rightarrow hf \Rightarrow hf \Rightarrow bool$
where

$ord-rec-Seq\ T\ G\ s\ k\ y \longleftrightarrow$
 $(Seq\ s\ k \wedge y = G\ (app\ s\ (\bigsqcup k)) \wedge app\ s\ 0 = T \wedge$
 $(\forall n. succ\ n \in k \longrightarrow app\ s\ (succ\ n) = G\ (app\ s\ n)))$

lemma *Seq-succ-insf*:

assumes s : $Seq\ s\ (succ\ k)$ **shows** $\exists y. s = insf\ s\ k\ y$
 ⟨proof⟩

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

assumes k : $Ord\ k$ **and** knz : $k \neq 0$
shows $ord-rec-Seq\ T\ G\ s\ (succ\ k)\ z \longleftrightarrow (\exists s'\ y. ord-rec-Seq\ T\ G\ s'\ k\ y \wedge z = G\ y \wedge s = insf\ s'\ k\ y)$
 ⟨proof⟩

lemma *ord-rec-Seq-functional*:

$Ord\ k \implies k \neq 0 \implies ord-rec-Seq\ T\ G\ s\ k\ y \implies ord-rec-Seq\ T\ G\ s'\ k\ y' \implies y' = y$
 ⟨proof⟩

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 $\exists s. ord-rec-Seq\ T\ G\ H\ s\ x\ y$
 else $y = H\ x$

lemma *ord-rec-functional*: $ord-rec\ T\ G\ H\ x\ y \implies ord-rec\ T\ G\ H\ x\ y' \implies y' = y$
 <proof>

lemma *ord-rec-succ-iff*:
assumes $k: Ord\ k$ **shows** $ord-rec\ T\ G\ H\ (succ\ k)\ z \longleftrightarrow (\exists y. z = G\ y \wedge ord-rec\ T\ G\ H\ k\ y)$
 <proof>

definition *ord-rec* :: $hf \Rightarrow (hf \Rightarrow hf) \Rightarrow (hf \Rightarrow hf) \Rightarrow hf \Rightarrow hf$
where
 $ord-rec\ T\ G\ H\ x = (THE\ y. ord-rec\ T\ G\ H\ x\ y)$

lemma *ord-rec-0* [*simp*]: $ord-rec\ T\ G\ H\ 0 = T$
 <proof>

lemma *ord-rec-total*: $\exists y. ord-rec\ T\ G\ H\ x\ y$
 <proof>

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>

lemma *ord-rec-non* [*simp*]: $\neg Ord\ x \implies ord-rec\ T\ G\ H\ x = H\ x$
 <proof>

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
<proof>

lemma *Vset-succ* [*simp*]: *Ord* *k* \Longrightarrow *Vset* (*succ* *k*) = *HPow* (*Vset* *k*)
<proof>

lemma *Vset-non* [*simp*]: \neg *Ord* *x* \Longrightarrow *Vset* *x* = 0
<proof>

Theorem 4.2(a)

lemma *Vset-mono-strict*:
assumes *Ord* *m* *n* \in *m* **shows** *Vset* *n* < *Vset* *m*
<proof>

lemma *Vset-mono*: \llbracket *Ord* *m*; *n* \leq *m* $\rrbracket \Longrightarrow$ *Vset* *n* \leq *Vset* *m*
<proof>

Theorem 4.2(b)

lemma *Vset-Transset*: *Ord* *m* \Longrightarrow *Transset* (*Vset* *m*)
<proof>

lemma *Ord-sup* [*simp*]: *Ord* *k* \Longrightarrow *Ord* *l* \Longrightarrow *Ord* (*k* \sqcup *l*)

<proof>

lemma *Ord-inf [simp]*: $Ord\ k \implies Ord\ l \implies Ord\ (k \sqcap l)$
<proof>

Theorem 4.3

lemma *Vset-universal*: $\exists n. Ord\ n \wedge x \in Vset\ n$
<proof>

3.2 Least Ordinal Operator

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

lemma *Ord-minimal*:
 $Ord\ k \implies P\ k \implies \exists n. Ord\ n \wedge P\ n \wedge (\forall m. Ord\ m \wedge P\ m \implies n \leq m)$
<proof>

lemma *OrdLeastI*: $Ord\ k \implies P\ k \implies P(LEAST\ n. Ord\ n \wedge P\ n)$
<proof>

lemma *OrdLeast-le*: $Ord\ k \implies P\ k \implies (LEAST\ n. Ord\ n \wedge P\ n) \leq k$
<proof>

lemma *OrdLeast-Ord*:
assumes $Ord\ k\ P\ k$ **shows** $Ord(LEAST\ n. Ord\ n \wedge P\ n)$
<proof>

3.3 Rank Function

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

lemma *[simp]*: $rank\ 0 = 0$
<proof>

lemma *in-Vset-rank*: $a \in Vset(succ(rank\ a))$
<proof>

lemma *Ord-rank [simp]*: $Ord\ (rank\ a)$
<proof>

lemma *le-Vset-rank*: $a \leq Vset(rank\ a)$
<proof>

lemma *VsetI*: $succ(rank\ a) \leq k \implies Ord\ k \implies a \in Vset\ k$
<proof>

lemma *Vset-succ-rank-le*: $Ord\ k \implies a \in Vset\ (succ\ k) \implies rank\ a \leq k$
<proof>

lemma *Vset-rank-lt*: **assumes** $a: a \in Vset\ k$ **shows** $rank\ a < k$
 ⟨*proof*⟩

Theorem 4.5

theorem *rank-lt*: $a \in b \implies rank(a) < rank(b)$
 ⟨*proof*⟩

lemma *rank-mono*: $x \leq y \implies rank\ x \leq rank\ y$
 ⟨*proof*⟩

lemma *rank-sup* [*simp*]: $rank\ (a \sqcup b) = rank\ a \sqcup rank\ b$
 ⟨*proof*⟩

lemma *rank-singleton* [*simp*]: $rank\ \{a\} = succ(rank\ a)$
 ⟨*proof*⟩

lemma *rank-hinsert* [*simp*]: $rank\ (b \triangleleft a) = rank\ b \sqcup succ(rank\ a)$
 ⟨*proof*⟩

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

3.4 Epsilon Closure

definition

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

lemma *eclose-facts*:

shows *Transset-eclose*: $Transset\ (eclose\ X)$

and *le-eclose*: $X \leq eclose\ X$

⟨*proof*⟩

lemma *eclose-minimal*:

assumes $Y: Transset\ Y\ X \leq Y$ **shows** $eclose\ X \leq Y$

⟨*proof*⟩

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

⟨*proof*⟩

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

⟨*proof*⟩

lemma *eclose-singleton* [*simp*]: $eclose\ \{a\} = (eclose\ a) \triangleleft a$

⟨*proof*⟩

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

⟨*proof*⟩

lemma *eclose-succ* [simp]: $eclose (succ a) = eclose a \triangleleft a$
 ⟨proof⟩

lemma *fst-in-eclose* [simp]: $x \in eclose \langle x, y \rangle$
 ⟨proof⟩

lemma *snd-in-eclose* [simp]: $y \in eclose \langle x, y \rangle$
 ⟨proof⟩

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

lemma *rank-eclose* [simp]: $rank (eclose x) = rank x$
 ⟨proof⟩

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⟩

definition
hmem-rel :: $(hf * hf)$ set **where**
hmem-rel = $transcl \{(x,y). x \in y\}$

lemma *wf-hmem-rel*: *wf hmem-rel*
 ⟨proof⟩

lemma *hmem-eclose-le*: $y \in x \implies eclose y \leq eclose x$
 ⟨proof⟩

lemma *hmem-rel-iff-hmem-eclose*: $(x,y) \in hmem-rel \iff x \in eclose y$
 ⟨proof⟩

definition *hmemrec* :: $((hf \Rightarrow 'a) \Rightarrow hf \Rightarrow 'a) \Rightarrow hf \Rightarrow 'a$ **where**
hmemrec $G \equiv wfrec hmem-rel G$

definition *ecut* :: $(hf \Rightarrow 'a) \Rightarrow hf \Rightarrow hf \Rightarrow 'a$ **where**
ecut $f x \equiv (\lambda y. \text{if } y \in eclose x \text{ then } f y \text{ else undefined})$

lemma *hmemrec*: $hmemrec G a = G (ecut (hmemrec G) a) a$
 ⟨proof⟩

This form avoids giant explosions in proofs.

lemma *def-hmemrec*: $f \equiv hmemrec G \implies f a = G (ecut (hmemrec G) a) a$
 ⟨proof⟩

lemma *ecut-apply*: $y \in eclose x \implies ecut f x y = f y$

<proof>

lemma *RepFun-ecut*: $y \leq z \implies \text{RepFun } y (\text{ecut } f z) = \text{RepFun } y f$
<proof>

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>

lemma *rank-HUnion-less*: $x \neq 0 \implies \text{rank } (\bigsqcup x) < \text{rank } x$
<proof>

corollary *Sup-ne*: $x \neq 0 \implies \bigsqcup x \neq x$
<proof>

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
```

```
  <proof>
```

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

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

```
  <proof>
```

```
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*: $regular\ \{\}$
 $\langle proof \rangle$

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

theorem *regular-emptystr*: $regular\ \{\}\}$
 $\langle proof \rangle$

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*: $regular\ \{[a]\}$
 $\langle proof \rangle$

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*]: $Reverse\ (Reverse\ fsm) = fsm$
 $\langle proof \rangle$

lemma *reaches-Reverse-iff* [*simp*]:
 $reaches\ (Reverse\ fsm)\ st\ (rev\ xs)\ st' \longleftrightarrow reaches\ fsm\ st'\ xs\ st$
 $\langle proof \rangle$

lemma *reaches-Reverse-iff2* [*simp*]:
 $reaches\ (Reverse\ fsm)\ st'\ xs\ st \longleftrightarrow reaches\ fsm\ st\ (rev\ xs)\ st'$
 $\langle proof \rangle$

lemma [*simp*]: $init\ (Reverse\ fsm) = final\ fsm$
 $\langle proof \rangle$

lemma [*simp*]: $final\ (Reverse\ fsm) = init\ fsm$
 $\langle proof \rangle$

lemma *accepts-Reverse*: $rev\ \{xs.\ accepts\ fsm\ xs\} = \{xs.\ accepts\ (Reverse\ fsm)\ xs\}$
 $\langle proof \rangle$

theorem *regular-rev*: $\text{regular } S \implies \text{regular } (\text{rev } S)$
 ⟨proof⟩

definition *Times where*

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

lemma *states-Times [simp]*: $\text{states } (\text{Times fsm1 fsm2}) = \text{states fsm1} * \text{states fsm2}$
 ⟨proof⟩

lemma *init-Times [simp]*: $\text{init } (\text{Times fsm1 fsm2}) = \text{init fsm1} * \text{init fsm2}$
 ⟨proof⟩

lemma *final-Times [simp]*: $\text{final } (\text{Times fsm1 fsm2}) = \text{final fsm1} * \text{final fsm2}$
 ⟨proof⟩

lemma *next-Times*: $\text{next } (\text{Times fsm1 fsm2}) \langle st1, st2 \rangle x st' \longleftrightarrow$
 $(\exists st1' st2'. st' = \langle st1', st2' \rangle \wedge \text{next fsm1 st1 x st1}' \wedge \text{next fsm2 st2 x st2}')$
 ⟨proof⟩

lemma *reaches-Times-iff [simp]*:
 $\text{reaches } (\text{Times fsm1 fsm2}) \langle st1, st2 \rangle xs \langle st1', st2' \rangle \longleftrightarrow$
 $\text{reaches fsm1 st1 xs st1}' \wedge \text{reaches fsm2 st2 xs st2}'$
 ⟨proof⟩

lemma *accepts-Times-iff [simp]*:
 $\text{accepts } (\text{Times fsm1 fsm2}) xs \longleftrightarrow \text{accepts fsm1 xs} \wedge \text{accepts fsm2 xs}$
 ⟨proof⟩

theorem *regular-Int*:

assumes *S*: $\text{regular } S$ **and** *T*: $\text{regular } T$ **shows** $\text{regular } (S \cap T)$
 ⟨proof⟩

definition *Plus where*

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

lemma *states-Plus [simp]*: $\text{states } (\text{Plus fsm1 fsm2}) = \text{states fsm1} + \text{states fsm2}$
 ⟨proof⟩

lemma *init-Plus* [*simp*]: $\text{init } (\text{Plus fsm1 fsm2}) = \text{init fsm1} + \text{init fsm2}$
<proof>

lemma *final-Plus* [*simp*]: $\text{final } (\text{Plus fsm1 fsm2}) = \text{final fsm1} + \text{final fsm2}$
<proof>

lemma *next-Plus1*: $\text{next } (\text{Plus fsm1 fsm2}) (\text{HF.Inl st1}) x st' \longleftrightarrow (\exists st1'. st' = \text{HF.Inl st1}' \wedge \text{next fsm1 st1 } x st1')$
<proof>

lemma *next-Plus2*: $\text{next } (\text{Plus fsm1 fsm2}) (\text{HF.Inr st2}) x st' \longleftrightarrow (\exists st2'. st' = \text{HF.Inr st2}' \wedge \text{next fsm2 st2 } x st2')$
<proof>

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

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

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

lemma *accepts-Plus-iff* [*simp*]:
 $\text{accepts } (\text{Plus fsm1 fsm2}) xs \longleftrightarrow \text{accepts fsm1 } xs \vee \text{accepts fsm2 } xs$
<proof>

lemma *regular-Un*:
assumes *S*: *regular S* **and** *T*: *regular T* **shows** *regular (S ∪ T)*
<proof>

end

theory *Finitary*

imports *Ordinal*

begin

class *finitary* =

fixes *hf-of* :: 'a ⇒ hf

assumes *inj*: *inj hf-of*

begin

lemma *hf-of-eq-iff* [*simp*]: $\text{hf-of } x = \text{hf-of } y \longleftrightarrow x=y$

<proof>

end

instantiation *unit* :: *finitary*

begin

definition *hf-of-unit-def*: $hf\text{-of } (u::unit) \equiv 0$

instance

<proof>

end

instantiation *bool* :: *finitary*

begin

definition *hf-of-bool-def*: $hf\text{-of } b \equiv \text{if } b \text{ then } 1 \text{ else } 0$

instance

<proof>

end

instantiation *nat* :: *finitary*

begin

definition *hf-of-nat-def*: $hf\text{-of } \equiv \text{ord-of}$

instance

<proof>

end

instantiation *int* :: *finitary*

begin

definition *hf-of-int-def*:

$hf\text{-of } i \equiv \text{if } i \geq (0::int) \text{ then } \langle 0, hf\text{-of } (nat\ i) \rangle \text{ else } \langle 1, hf\text{-of } (nat\ (-i)) \rangle$

instance

<proof>

end

Strings are char lists, and are not considered separately.

instantiation *char* :: *finitary*

begin

definition *hf-of-char-def*:

$hf\text{-of } x \equiv hf\text{-of } (of\text{-char } x :: nat)$

instance

<proof>

end

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

begin

definition *hf-of-prod-def*:

$hf\text{-of } \equiv \lambda(x,y). \langle hf\text{-of } x, hf\text{-of } y \rangle$

instance

<proof>

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
    <proof>
end

instantiation option :: (finitary) finitary
begin
  definition hf-of-option-def:
    hf-of  $\equiv$  case-option 0 ( $\lambda x.$  {hf-of x})
  instance
    <proof>
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
  <proof>

instance
  <proof>
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

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

lemma *hadd*: $x @+ y = x \sqcup \text{RepFun } y (\lambda z. x @+ z)$
<proof>

lemma *hmem-hadd-E*:

assumes $l \in x @+ y$
obtains $l \in x \mid z \text{ where } z \in y \ l = x @+ z$
<proof>

lemma *hadd-0-right* [*simp*]: $x @+ 0 = x$
<proof>

lemma *hadd-hinsert-right*: $x @+ \text{hinsert } y z = \text{hinsert } (x @+ y) (x @+ z)$
<proof>

lemma *hadd-succ-right* [*simp*]: $x @+ \text{succ } y = \text{succ } (x @+ y)$
<proof>

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

lemma *not-add-mem-right*: $\neg (x @+ y \in x)$
 ⟨proof⟩

lemma *hadd-0-left [simp]*: $0 @+ x = x$
 ⟨proof⟩

lemma *hadd-succ-left [simp]*: $Ord\ y \implies succ\ x @+ y = succ\ (x @+ y)$
 ⟨proof⟩

lemma *hadd-assoc*: $(x @+ y) @+ z = x @+ (y @+ z)$
 ⟨proof⟩

lemma *RepFun-hadd-disjoint*: $x \sqcap RepFun\ y\ ((@+) x) = 0$
 ⟨proof⟩

5.1.1 Cancellation laws for addition

lemma *Rep-le-Cancel*: $x \sqcup RepFun\ y\ ((@+) x) \leq x \sqcup RepFun\ z\ ((@+) x)$
 $\implies RepFun\ y\ ((@+) x) \leq RepFun\ z\ ((@+) x)$
 ⟨proof⟩

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

lemma *RepFun-hadd-cancel*: $RepFun\ y\ (\lambda z. x @+ z) = RepFun\ z\ (\lambda z. x @+ z)$
 $\longleftrightarrow y = z$
 ⟨proof⟩

lemma *hadd-hmem-cancel [simp]*: $x @+ y \in x @+ z \longleftrightarrow y \in z$
 ⟨proof⟩

lemma *ord-of-add*: $ord\text{-of}\ (i+j) = ord\text{-of}\ i @+ ord\text{-of}\ j$
 ⟨proof⟩

lemma *Ord-hadd*: $Ord\ x \implies Ord\ y \implies Ord\ (x @+ y)$
 ⟨proof⟩

lemma *hmem-self-hadd [simp]*: $k1 \in k1 @+ k2 \longleftrightarrow 0 \in k2$
 ⟨proof⟩

lemma *hadd-commute*: $Ord\ x \implies Ord\ y \implies x @+ y = y @+ x$
 ⟨proof⟩

lemma *hadd-cancel-left [simp]*: $Ord\ x \implies y @+ x = z @+ x \longleftrightarrow y = z$
 ⟨proof⟩

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]: $\text{pred } (\text{succ } x) = x$
 ⟨proof⟩

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

lemma *succ-pred* [simp]: $\text{Ord } x \implies x \neq 0 \implies \text{succ } (\text{pred } x) = x$
 ⟨proof⟩

lemma *pred-mem* [simp]: $\text{Ord } x \implies x \neq 0 \implies \text{pred } x \in x$
 ⟨proof⟩

lemma *Ord-pred* [simp]: $\text{Ord } x \implies \text{Ord } (\text{pred } x)$
 ⟨proof⟩

lemma *hadd-pred-right*: $\text{Ord } y \implies y \neq 0 \implies x @+ \text{pred } y = \text{pred } (x @+ y)$
 ⟨proof⟩

lemma *Ord-pred-HUnion*: $\text{Ord}(k) \implies \text{pred } k = \bigsqcup k$
 ⟨proof⟩

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*: $x \in \text{shift } f \text{ delta} \implies \exists u. u \in f \wedge x = \langle \text{delta } @+ \text{hfst } u, \text{hsnd } u \rangle$
 ⟨proof⟩

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

lemma *hmem-shift-add-iff* [simp]: $\langle \text{delta } @+ n, y \rangle \in \text{shift } f \text{ delta} \longleftrightarrow \langle n, y \rangle \in f$
 ⟨proof⟩

lemma *hrelation-shift* [simp]: $\text{hrelation } (\text{shift } f \text{ delta})$
 ⟨proof⟩

lemma *app-shift* [simp]: $\text{app } (\text{shift } f \text{ delta}) (k @+ j) = \text{app } f j$
 ⟨proof⟩

lemma *hfunction-shift-iff* [simp]: $\text{hfunction } (\text{shift } f \text{ delta}) = \text{hfunction } f$
 ⟨proof⟩

lemma *hdomain-shift-add*: $\text{hdomain } (\text{shift } f \text{ delta}) = \{\text{delta } @+ n . n \in \text{hdomain } f\}$

<proof>

lemma *hdomain-shift-disjoint*: $\text{delta} \sqcap \text{hdomain} (\text{shift } f \text{ delta}) = 0$
<proof>

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

lemma *hrelation-seq-append* [*simp*]: *hrelation* (*seq-append* $k f g$)
<proof>

lemma *Seq-append*:
assumes *Seq* $s1 k1$ *Seq* $s2 k2$
shows *Seq* (*seq-append* $k1 s1 s2$) ($k1 @+ k2$)
<proof>

lemma *app-hunion1*: $x \notin \text{hdomain } g \Longrightarrow \text{app } (f \sqcup g) x = \text{app } f x$
<proof>

lemma *app-hunion2*: $x \notin \text{hdomain } f \Longrightarrow \text{app } (f \sqcup g) x = \text{app } g x$
<proof>

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

lemma *Seq-append-app2*: *Seq* $s1 k1 \Longrightarrow \text{Seq } s2 k2 \Longrightarrow l = k1 @+ j \Longrightarrow \text{app } (\text{seq-append } k1 s1 s2) l = \text{app } s2 j$
<proof>

5.3 Nonempty sequences indexed by ordinals

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

lemma *OrdDom-insf*: $\llbracket \text{OrdDom } s; \text{Ord } k \rrbracket \Longrightarrow \text{OrdDom } (\text{insf } s (\text{succ } k) y)$
<proof>

lemma *OrdDom-hunion* [*simp*]: $\text{OrdDom } (s1 \sqcup s2) \longleftrightarrow \text{OrdDom } s1 \wedge \text{OrdDom } s2$
<proof>

lemma *OrdDom-hrestrict*: $\text{OrdDom } s \Longrightarrow \text{OrdDom } (\text{hrestrict } s A)$
<proof>

lemma *OrdDom-shift*: $\llbracket \text{OrdDom } s; \text{Ord } k \rrbracket \Longrightarrow \text{OrdDom } (\text{shift } s k)$
<proof>

A sequence of positive length ending with y

definition $LstSeq :: hf \Rightarrow hf \Rightarrow hf \Rightarrow bool$

where $LstSeq\ s\ k\ y \equiv Seq\ s\ (succ\ k) \wedge Ord\ k \wedge \langle k, y \rangle \in s \wedge OrdDom\ s$

lemma $LstSeq\ imp\ Seq\ succ$: $LstSeq\ s\ k\ y \Longrightarrow Seq\ s\ (succ\ k)$

$\langle proof \rangle$

lemma $LstSeq\ imp\ Seq\ same$: $LstSeq\ s\ k\ y \Longrightarrow Seq\ s\ k$

$\langle proof \rangle$

lemma $LstSeq\ imp\ Ord$: $LstSeq\ s\ k\ y \Longrightarrow Ord\ k$

$\langle proof \rangle$

lemma $LstSeq\ trunc$: $LstSeq\ s\ k\ y \Longrightarrow l \in k \Longrightarrow LstSeq\ s\ l\ (app\ s\ l)$

$\langle proof \rangle$

lemma $LstSeq\ insf$: $LstSeq\ s\ k\ z \Longrightarrow LstSeq\ (insf\ s\ (succ\ k)\ y)\ (succ\ k)\ y$

$\langle proof \rangle$

lemma $app\ insf\ LstSeq$: $LstSeq\ s\ k\ z \Longrightarrow app\ (insf\ s\ (succ\ k)\ y)\ (succ\ k) = y$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma $LstSeq\ append\ app1$:

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

$\langle proof \rangle$

lemma $LstSeq\ append\ app2$:

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

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

$\langle proof \rangle$

lemma $Seq\ append\ pair$:

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma $LstSeq\ append$:

$\llbracket LstSeq\ s1\ k1\ y1; LstSeq\ s2\ k2\ y2 \rrbracket \Longrightarrow LstSeq\ (seq\ append\ (succ\ k1)\ s1\ s2)\ (succ$

$(k1 \text{ @+ } k2)) \ y2$
 $\langle \text{proof} \rangle$

lemma *LstSeq-app [simp]*: $LstSeq \ s \ k \ y \implies \text{app } s \ k = y$
 $\langle \text{proof} \rangle$

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 $Builds \ B \ C \ s \ l \equiv B \ (\text{app } s \ l) \vee (\exists m \in l. \exists n \in l. C \ (\text{app } s \ l) \ (\text{app } s \ m) \ (\text{app } s \ n))$

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

lemma *BuildSeqI*: $LstSeq \ s \ k \ y \implies (\bigwedge l. l \in \text{succ } k \implies Builds \ B \ C \ s \ l) \implies BuildSeq \ B \ C \ s \ k \ y$
 $\langle \text{proof} \rangle$

lemma *BuildSeq-imp-LstSeq*: $BuildSeq \ B \ C \ s \ k \ y \implies LstSeq \ s \ k \ y$
 $\langle \text{proof} \rangle$

lemma *BuildSeq-imp-Seq*: $BuildSeq \ B \ C \ s \ k \ y \implies Seq \ s \ (\text{succ } k)$
 $\langle \text{proof} \rangle$

lemma *BuildSeq-conj-distrib*:
 $BuildSeq \ (\lambda x. B \ x \wedge P \ x) \ (\lambda x \ y \ z. C \ x \ y \ z \wedge P \ x) \ s \ k \ y \longleftrightarrow BuildSeq \ B \ C \ s \ k \ y \wedge (\forall l \in \text{succ } k. P \ (\text{app } s \ l))$
 $\langle \text{proof} \rangle$

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 $BuildSeq \ B' \ C' \ s \ k \ y$
 $\langle \text{proof} \rangle$

lemma *BuildSeq-trunc*:
assumes $b: BuildSeq \ B \ C \ s \ k \ y$
and $l: l \in k$
shows $BuildSeq \ B \ C \ s \ l \ (\text{app } s \ l)$
 $\langle \text{proof} \rangle$

5.3.2 Showing that Sequences can be Constructed

lemma *Builds-insf*: $Builds \ B \ C \ s \ l \implies LstSeq \ s \ k \ z \implies l \in \text{succ } k \implies Builds \ B \ C \ (\text{insf } s \ (\text{succ } k) \ y) \ l$
 $\langle \text{proof} \rangle$

lemma *BuildSeq-insf*:

assumes $b: \text{BuildSeq } B \ C \ s \ k \ z$
and $m: m \in \text{succ } k$
and $n: n \in \text{succ } k$
and $y: B \ y \ \vee \ C \ y \ (\text{app } s \ m) \ (\text{app } s \ n)$
shows $\text{BuildSeq } B \ C \ (\text{insf } s \ (\text{succ } k) \ y) \ (\text{succ } k) \ y$
 $\langle \text{proof} \rangle$

lemma *BuildSeq-insf1*:
assumes $b: \text{BuildSeq } B \ C \ s \ k \ z$
and $y: B \ y$
shows $\text{BuildSeq } B \ C \ (\text{insf } s \ (\text{succ } k) \ y) \ (\text{succ } k) \ y$
 $\langle \text{proof} \rangle$

lemma *BuildSeq-insf2*:
assumes $b: \text{BuildSeq } B \ C \ s \ k \ z$
and $m: m \in k$
and $n: n \in k$
and $y: C \ y \ (\text{app } s \ m) \ (\text{app } s \ n)$
shows $\text{BuildSeq } B \ C \ (\text{insf } s \ (\text{succ } k) \ y) \ (\text{succ } k) \ y$
 $\langle \text{proof} \rangle$

lemma *BuildSeq-append*:
assumes $s1: \text{BuildSeq } B \ C \ s1 \ k1 \ y1$ **and** $s2: \text{BuildSeq } B \ C \ s2 \ k2 \ y2$
shows $\text{BuildSeq } B \ C \ (\text{seq-append } (\text{succ } k1) \ s1 \ s2) \ (\text{succ } (k1 \ @+ \ k2)) \ y2$
 $\langle \text{proof} \rangle$

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 $\text{BuildSeq } B \ C \ (\text{insf } (\text{seq-append } (\text{succ } k1) \ s1 \ s2) \ (\text{succ } (\text{succ } (k1 \ @+ \ k2))))$
 $y \ (\text{succ } (\text{succ } (k1 \ @+ \ k2))) \ y$
 $\langle \text{proof} \rangle$

lemma *LstSeq-1*: $\text{LstSeq } \{\langle 0, y \rangle\} \ 0 \ y$
 $\langle \text{proof} \rangle$

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

lemma *BuildSeq-exI*: $B \ t \ \Longrightarrow \ \exists \ s \ k. \ \text{BuildSeq } B \ C \ s \ k \ t$
 $\langle \text{proof} \rangle$

5.3.3 Proving Properties of Given Sequences

lemma *BuildSeq-succ-E*:
assumes $s: \text{BuildSeq } B \ C \ s \ k \ y$
obtains $B \ y \ | \ m \ n$ **where** $m \in k \ n \in k \ C \ y \ (\text{app } s \ m) \ (\text{app } s \ n)$
 $\langle \text{proof} \rangle$

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

assumes major: *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>

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

$\Rightarrow bool$

where *BuildSeq2* B C s k y y' \equiv

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: *BuildSeq2* B C s1 k1 y1 y1' and b2: *BuildSeq2* B C s2 k2 y2 y2'

and y: C y y' y1 y1' y2 y2'

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

(succ (succ (k1 @+ k2))) y y'

<proof>

lemma *BuildSeq2-1*: B y y' \implies *BuildSeq2* B C $\{\langle 0, y, y' \rangle\}$ 0 y y'

<proof>

lemma *BuildSeq2-exI*: B t t' $\implies \exists s k. \text{BuildSeq2 } B C s k t t'$

<proof>

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'

<proof>

definition *BuildSeq3*

:: $[[hf, hf, hf] \Rightarrow bool, [hf, hf, hf, hf, hf, hf, hf, hf, hf] \Rightarrow bool, hf, hf, hf, hf, hf] \Rightarrow bool$

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 $y: C\ y\ y'\ y''\ y_1\ y_1'\ y_1''\ y_2\ y_2'\ y_2''$
shows $BuildSeq3\ B\ C\ (insf\ (seq-append\ (succ\ k1)\ s1\ s2)\ (succ\ (succ\ (k1\ @+\ k2))))\ \langle y,\ y',\ y'' \rangle$
 $(succ\ (succ\ (k1\ @+\ k2)))\ y\ y'\ y''$
 $\langle proof \rangle$

lemma $BuildSeq3-1: B\ y\ y'\ y'' \implies BuildSeq3\ B\ C\ \{\{0,\ y,\ y',\ y''\}\}\ 0\ y\ y'\ y''$
 $\langle proof \rangle$

lemma $BuildSeq3-exI: B\ t\ t'\ t'' \implies \exists s\ k.\ BuildSeq3\ B\ C\ s\ k\ t\ t'\ t''$
 $\langle proof \rangle$

lemma $BuildSeq3-induct$ [consumes 1, case-names B C]:
assumes $BuildSeq3\ B\ C\ s\ k\ a\ a''$
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 $P\ a\ a'\ a''$
 $\langle proof \rangle$

5.4 A Unique Predecessor for every non-empty set

lemma $Rep-hf-0$ [simp]: $Rep-hf\ 0 = 0$
 $\langle proof \rangle$

lemma $hmem-imp-less: x \in y \implies Rep-hf\ x < Rep-hf\ y$
 $\langle proof \rangle$

lemma $hsubset-imp-le:$
assumes $x \leq y$ **shows** $Rep-hf\ x \leq Rep-hf\ y$
 $\langle proof \rangle$

lemma $diff-hmem-imp-less: \text{assumes } x \in y \text{ shows } Rep-hf\ (y - \{x\}) < Rep-hf\ y$
 $\langle proof \rangle$

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

lemma $least-equality:$
assumes $x \in a$ **and** $\bigwedge y.\ y \in a \implies Rep-hf\ x \leq Rep-hf\ y$
shows $least\ a = x$
 $\langle proof \rangle$

lemma $leastI2-order:$
assumes $x \in a$
and $\bigwedge y.\ y \in a \implies Rep-hf\ x \leq Rep-hf\ y$
and $\bigwedge z.\ z \in a \implies \forall y.\ y \in a \longrightarrow Rep-hf\ z \leq Rep-hf\ y \implies Q\ z$
shows $Q\ (least\ a)$
 $\langle proof \rangle$

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>

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

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>.