

HOTG

Kevin Kappelmann

March 21, 2024

Abstract

TODO

Contents

1	Setup for Higher-Order Tarski-Grothendieck Set Theory.	3
2	Axioms of Tarski-Grothendieck Set Theory embedded in HOL.	3
3	Basic Lemmas	6
4	Subset	7
4.1	Strict Subsets	8
5	Transitive Sets	8
5.1	Order on Sets	9
6	Powerset	10
7	Bounded Quantifiers	10
8	Bounded definite description	14
9	Set Equality	14
10	Replacement	16
10.1	Image	17
11	Unordered Pairs	18
12	Finite Sets	20
12.1	Replacement	21
13	Restricted Comprehension	21

14 Union and Intersection	22
14.1 Indexed Union and Intersection:	24
14.2 Binary Union and Intersection	26
15 Well-Foundedness of Sets	33
16 Transfinite Recursion	35
17 Transitive Closure With Respect To Membership	36
18 Less-Than and Less-Than or Equal Orders	38
19 Generalised Addition	43
20 Ordinals	50
21 Generalised Multiplication	53
22 Pairs (Σ-types)	58
22.1 Set-Theoretic Dependent Pair Type	59
22.2 Monotonicity	61
22.3 Functions on Dependent Pairs	61
23 Coproduct (\coprod-types)	62
24 Cardinals	63
24.1 Antisymmetric	68
24.2 Connected	69
25 Replacement on Function-Like Predicates	69
25.1 Functions on Relations	71
25.1.1 Inverse	71
25.1.2 Extensions and Restricts	72
25.1.3 Domain and Range	76
25.1.4 Composition	78
25.1.5 Diagonal	78
25.2 Injective	79
25.3 Irreflexive	80
25.4 Left Total	80
25.5 Reflexive	82
25.5.1 Right Unique	82
25.6 Surjective	83
25.7 Symmetric	85
25.8 Transitive	85
25.9 Basic Properties	86

26 Set-Theoretic Binary Relations	86
26.1 Evaluation of Functions	86
26.1.1 Dependent Functions	88
26.2 Lambda Abstractions	93
26.3 Composition	96
26.4 Extending Functions	97
26.5 Gluing	98
26.6 Restriction	99
27 Functions	100
28 Set-Theoretic Orders	100
29 Empty Set	101
30 Set Difference	101
31 Universes	103

1 Setup for Higher-Order Tarski-Grothendieck Set Theory.

```

theory Setup
  imports Transport.HOL-Syntax-Bundles-Base
begin

  Remove conflicting HOL-specific syntax.

  unbundle no-HOL-ascii-syntax

  Additional logical rules

  lemma or-if-not-imp:  $(\neg A \implies B) \implies A \vee B$  by blast

end

```

2 Axioms of Tarski-Grothendieck Set Theory embedded in HOL.

```

theory Axioms
  imports Setup
begin

```

Summary We follow the axiomatisation as described in [1], who also describe the existence of a model if a 2-inaccessible cardinal exists.

The primitive set type.

typed decl *set*

The first four axioms.

axiomatization

mem :: $\langle \text{set} \Rightarrow \text{set} \Rightarrow \text{bool} \rangle$ **and**
emptyset :: $\langle \text{set} \rangle$ **and**
union :: $\langle \text{set} \Rightarrow \text{set} \rangle$ **and**
repl :: $\langle \text{set} \Rightarrow (\text{set} \Rightarrow \text{set}) \Rightarrow \text{set} \rangle$

where

mem-induction: $(\forall X. (\forall x. \text{mem } x X \longrightarrow P x) \longrightarrow P X) \longrightarrow (\forall X. P X)$ **and**
emptyset: $\neg(\exists x. \text{mem } x \text{ emptyset})$ **and**
union: $\forall X x. \text{mem } x (\text{union } X) \longleftrightarrow (\exists Y. \text{mem } Y X \wedge \text{mem } x Y)$ **and**
replacement: $\forall X y. \text{mem } y (\text{repl } X f) \longleftrightarrow (\exists x. \text{mem } x X \wedge y = f x)$

Note: axioms $(\forall X. (\forall x. \text{mem } x X \longrightarrow ?P x) \longrightarrow ?P X) \longrightarrow (\forall X. ?P X)$ and $\forall X y. \text{mem } y (\text{repl } X ?f) = (\exists x. \text{mem } x X \wedge y = ?f x)$ are axiom schemas in first-order logic. Moreover, $\forall X y. \text{mem } y (\text{repl } X ?f) = (\exists x. \text{mem } x X \wedge y = ?f x)$ takes a meta-level function F .

Let us define some expected notation.

bundle *hotg-mem-syntax* **begin notation** *mem* (infixl \in 50) **end**
bundle *no-hotg-mem-syntax* **begin no-notation** *mem* (infixl \in 50) **end**

bundle *hotg-emptyset-zero-syntax* **begin notation** *emptyset* (\emptyset) **end**
bundle *no-hotg-emptyset-zero-syntax* **begin no-notation** *emptyset* (\emptyset) **end**

bundle *hotg-emptyset-braces-syntax* **begin notation** *emptyset* ($\{\}$) **end**
bundle *no-hotg-emptyset-braces-syntax* **begin no-notation** *emptyset* ($\{\}$) **end**

bundle *hotg-emptyset-syntax*
begin
 unbundle *hotg-emptyset-zero-syntax* *hotg-emptyset-braces-syntax*
end
bundle *no-hotg-emptyset-syntax*
begin
 unbundle *no-hotg-emptyset-zero-syntax* *no-hotg-emptyset-braces-syntax*
end

bundle *hotg-union-syntax* **begin notation** *union* (\bigcup - [90] 90) **end**
bundle *no-hotg-union-syntax* **begin no-notation** *union* (\bigcup - [90] 90) **end**

unbundle *hotg-mem-syntax* *hotg-emptyset-syntax* *hotg-union-syntax*

abbreviation (*input*) *mem-of* A $x \equiv x \in A$

abbreviation *not-mem* $x y \equiv \neg(x \in y)$

bundle *hotg-not-mem-syntax* **begin notation** *not-mem* (**infixl** \notin 50) **end**
bundle *no-hotg-not-mem-syntax* **begin no-notation** *not-mem* (**infixl** \notin 50) **end**

unbundle *hotg-not-mem-syntax*

Based on the membership relation, we can define the subset relation.

definition *subset* :: $\langle \text{set} \Rightarrow \text{set} \Rightarrow \text{bool} \rangle$
where *subset* $A\ B \equiv \forall x. x \in A \longrightarrow x \in B$

Again, we define some notation.

bundle *hotg-subset-syntax* **begin notation** *subset* (**infixl** \subseteq 50) **end**
bundle *no-hotg-subset-syntax* **begin no-notation** *subset* (**infixl** \subseteq 50) **end**

unbundle *hotg-subset-syntax*

The axiom of extensionality and powerset.

axiomatization

powerset :: $\langle \text{set} \Rightarrow \text{set} \rangle$

where

extensionality: $\forall X\ Y. X \subseteq Y \longrightarrow Y \subseteq X \longrightarrow X = Y$ **and**
powerset: $\forall A\ x. x \in \text{powerset } A \longleftrightarrow x \subseteq A$

Lastly, we want to axiomatise the existence of Grothendieck universes. This can be done in different ways. We again follow the approach from [1].

definition *mem-trans-closed* :: $\langle \text{set} \Rightarrow \text{bool} \rangle$

where *mem-trans-closed* $X \equiv (\forall x. x \in X \longrightarrow x \subseteq X)$

definition *ZF-closed* :: $\langle \text{set} \Rightarrow \text{bool} \rangle$

where *ZF-closed* $U \equiv ($
 $(\forall X. X \in U \longrightarrow \bigcup X \in U) \wedge$
 $(\forall X. X \in U \longrightarrow \text{powerset } X \in U) \wedge$
 $(\forall X\ F. X \in U \longrightarrow (\forall x. x \in X \longrightarrow F\ x \in U) \longrightarrow \text{repl } X\ F \in U)$
 $)$

Note that *ZF-closed* is a second-order statement.

univ X is the smallest Grothendieck universe containing X .

axiomatization

univ :: $\langle \text{set} \Rightarrow \text{set} \rangle$

where

mem-univ [*iff*]: $X \in \text{univ } X$ **and**
mem-trans-closed-univ [*iff*]: *mem-trans-closed* (*univ* X) **and**
ZF-closed-univ [*iff*]: *ZF-closed* (*univ* X) **and**
univ-min: $\llbracket X \in U; \text{mem-trans-closed } U; \text{ZF-closed } U \rrbracket \implies \text{univ } X \subseteq U$

bundle *hotg-basic-syntax*
begin

```

unbundle
  hotg-mem-syntax
  hotg-not-mem-syntax
  hotg-emptyset-syntax
  hotg-union-syntax
  hotg-subset-syntax
end
bundle no-hotg-basic-syntax
begin
  unbundle
    no-hotg-mem-syntax
    no-hotg-not-mem-syntax
    no-hotg-emptyset-syntax
    no-hotg-union-syntax
    no-hotg-subset-syntax
  end
end

```

3 Basic Lemmas

```

theory Basic
  imports Axioms
begin

```

Summary Here we derive a few preliminary lemmas following from the axioms that are needed to formalise more complicated concepts.

The following are easier to work with variants of the axioms.

lemma *not-mem-emptyset* [*iff*]: $x \notin \{\}$ **using** *emptyset* **by** *blast*

lemma *eq-if-subset-if-subset* [*intro*]: $\llbracket X \subseteq Y; Y \subseteq X \rrbracket \implies X = Y$
by (*fact Axioms.extensionality*[*rule-format*])

lemma *mem-induction* [*case-names mem, induct type: set*]:
 $(\bigwedge X. (\bigwedge x. x \in X \implies P\ x) \implies P\ X) \implies P\ X$
by (*fact Axioms.mem-induction*[*rule-format*])

lemma *mem-union-iff-mem-mem* [*iff*]: $(x \in \bigcup X) \longleftrightarrow (\exists Y. Y \in X \wedge x \in Y)$
by (*fact Axioms.union*[*rule-format*])

corollary *mem-unionI*:
assumes $Y \in X$
and $x \in Y$
shows $x \in \bigcup X$
using *assms mem-union-iff-mem-mem* **by** *auto*

corollary *mem-unionE*:

```

assumes  $x \in \bigcup X$ 
obtains  $Y$  where  $Y \in X$   $x \in Y$ 
using assms mem-union-iff-mem-mem by auto

lemma mem-powerset-iff-subset [iff]:  $(x \in \text{powerset } A) \longleftrightarrow (x \subseteq A)$ 
by (fact Axioms.powerset[rule-format])

corollary mem-powerset-if-subset:
assumes  $x \subseteq A$ 
shows  $x \in \text{powerset } A$ 
using assms by blast

corollary subset-if-mem-powerset:
assumes  $x \in \text{powerset } A$ 
shows  $x \subseteq A$ 
using assms by blast

lemma mem-repl-iff-mem-eq [iff]:  $(y \in \text{repl } X f) \longleftrightarrow (\exists x. x \in X \wedge y = f x)$ 
by (fact Axioms.replacement[rule-format])

corollary mem-replI:
assumes  $y = f x$ 
and  $x \in X$ 
shows  $y \in \text{repl } X f$ 
using assms mem-repl-iff-mem-eq by blast

corollary mem-replE:
assumes  $y \in \text{repl } X f$ 
obtains  $x$  where  $y = f x$   $x \in X$ 
using assms mem-repl-iff-mem-eq by blast

end

```

4 Subset

```

theory Subset
imports Basic
begin

lemma subsetI [intro!]:  $(\bigwedge x. x \in A \implies x \in B) \implies A \subseteq B$ 
unfolding subset-def by simp

lemma subsetD [dest]:  $\llbracket A \subseteq B; a \in A \rrbracket \implies a \in B$ 
unfolding subset-def by blast

lemma mem-if-subset-if-mem [trans]:  $\llbracket a \in A; A \subseteq B \rrbracket \implies a \in B$  by blast

lemma subset-self [iff]:  $A \subseteq A$  by blast

```

lemma *empty-subset* [iff]: $\{\} \subseteq A$ **by** *blast*

lemma *subset-empty-iff* [iff]: $A \subseteq \{\} \longleftrightarrow A = \{\}$ **by** *blast*

lemma *not-mem-if-subset-if-not-mem* [trans]: $\llbracket a \notin B; A \subseteq B \rrbracket \Longrightarrow a \notin A$
by *blast*

lemma *subset-if-subset-if-subset* [trans]: $\llbracket A \subseteq B; B \subseteq C \rrbracket \Longrightarrow A \subseteq C$
by *blast*

lemma *subsetCE* [elim]:
 assumes $A \subseteq B$
 obtains $a \notin A \mid a \in B$
 using *assms* **by** *auto*

4.1 Strict Subsets

definition *ssubset* $A B \equiv A \subseteq B \wedge A \neq B$

bundle *hotg-ssubset-syntax* **begin notation** *ssubset* (infixl \subset 50) **end**
bundle *no-hotg-xsubset-syntax* **begin no-notation** *ssubset* (infixl \subset 50) **end**
unbundle *hotg-ssubset-syntax*

lemma *ssubsetI* [intro]:
 assumes $A \subseteq B$
 and $A \neq B$
 shows $A \subset B$
 unfolding *ssubset-def* **using** *assms* **by** *blast*

lemma *ssubsetE* [elim]:
 assumes $A \subset B$
 obtains $A \subseteq B \ A \neq B$
 using *assms* **unfolding** *ssubset-def* **by** *blast*

end

5 Transitive Sets

theory *Mem-Transitive-Closed-Base*
imports *Subset*
begin

lemma *mem-trans-closedI* [intro]: $(\bigwedge x. x \in X \Longrightarrow x \subseteq X) \Longrightarrow \text{mem-trans-closed } X$
unfolding *mem-trans-closed-def* **by** *auto*

lemma *mem-trans-closedI'*: $(\bigwedge x y. x \in X \implies y \in x \implies y \in X) \implies \text{mem-trans-closed } X$

by *auto*

lemma *mem-trans-closedD* [*dest*]:

assumes *mem-trans-closed* *x*

shows $\bigwedge y. y \in x \implies y \subseteq x$

using *assms* **unfolding** *mem-trans-closed-def* **by** *auto*

lemma *mem-trans-closed-empty* [*iff*]: *mem-trans-closed* $\{\}$ **by** *auto*

end

5.1 Order on Sets

theory *Order-Set*

imports

Transport.Functions-Monotone

HOL.Orderings

Subset

begin

unbundle *no-HOL-ascii-syntax*

instantiation *set* :: *order*

begin

definition *le-set-def*: *less-eq-set* $\equiv (\subseteq)$

definition *lt-set-def*: *less-set* $\equiv (\subset)$

lemma *le-set-eq-subset* [*simp*]: $(\leq) = (\subseteq)$ **unfolding** *le-set-def* **by** *simp*

lemma *lt-set-eq-ssubset* [*simp*]: $(<) = (\subset)$ **unfolding** *lt-set-def* **by** *simp*

instance **by** (*standard*) *auto*

end

lemma *mono-mem-of*: *mono mem-of*

by (*intro monoI*) *auto*

lemma *le-boolD'*: $P \leq Q \implies P \implies Q$ **by** (*rule le-boolE*)

lemma *le-boolD''*: $P \implies P \leq Q \implies Q$ **by** (*rule le-boolE*)

end

6 Powerset

```

theory Powerset
  imports Order-Set
begin

lemma mem-powerset-if-subset:  $A \subseteq B \implies A \in \text{powerset } B$ 
  by auto

lemma subset-if-mem-powerset:  $A \in \text{powerset } B \implies A \subseteq B$ 
  by auto

lemma empty-mem-powerset [iff]:  $\{\} \in \text{powerset } A$ 
  by auto

lemma mem-powerset-self [iff]:  $A \in \text{powerset } A$ 
  by auto

lemma mem-powerset-empty-iff-eq-empty [iff]:  $x \in \text{powerset } \{\} \longleftrightarrow x = \{\}$ 
  by auto

lemma mono-powerset: mono powerset
  by (intro monoI) auto

end

```

7 Bounded Quantifiers

```

theory Bounded-Quantifiers
  imports Order-Set
begin

definition ball ::  $\langle \text{set} \Rightarrow (\text{set} \Rightarrow \text{bool}) \Rightarrow \text{bool} \rangle$ 
  where ball  $A P \equiv (\forall x. x \in A \longrightarrow P x)$ 

definition bex ::  $\langle \text{set} \Rightarrow (\text{set} \Rightarrow \text{bool}) \Rightarrow \text{bool} \rangle$ 
  where bex  $A P \equiv \exists x. x \in A \wedge P x$ 

definition bex1 ::  $\langle \text{set} \Rightarrow (\text{set} \Rightarrow \text{bool}) \Rightarrow \text{bool} \rangle$ 
  where bex1  $A P \equiv \exists! x. x \in A \wedge P x$ 

bundle hotg-bounded-quantifiers-syntax
begin
syntax
  -ball ::  $\langle [\text{idts}, \text{set}, \text{bool}] \Rightarrow \text{bool} \rangle ((2\forall - \in - / -) 10)$ 
  -ball2 ::  $\langle [\text{idts}, \text{set}, \text{bool}] \Rightarrow \text{bool} \rangle$ 

```

```

-bex  :: ⟨[idts, set, bool] ⇒ bool⟩ ((2∃ - ∈ -./ -) 10)
-bex2 :: ⟨[idts, set, bool] ⇒ bool⟩
-bex1 :: ⟨[idt, set, bool] ⇒ bool⟩ ((2∃!- ∈ -./ -) 10)
end
bundle no-hotg-bounded-quantifiers-syntax
begin
no-syntax
-ball  :: ⟨[idts, set, bool] ⇒ bool⟩ ((2∀ - ∈ -./ -) 10)
-ball2 :: ⟨[idts, set, bool] ⇒ bool⟩
-bex  :: ⟨[idts, set, bool] ⇒ bool⟩ ((2∃ - ∈ -./ -) 10)
-bex2 :: ⟨[idts, set, bool] ⇒ bool⟩
-bex1 :: ⟨[idt, set, bool] ⇒ bool⟩ ((2∃!- ∈ -./ -) 10)
end
unbundle hotg-bounded-quantifiers-syntax
translations
  ∀ x xs ∈ A. P ⇝ CONST ball A (λx. -ball2 xs A P)
  -ball2 x A P ⇝ ∀ x ∈ A. P
  ∀ x ∈ A. P ⇐ CONST ball A (λx. P)

  ∃ x xs ∈ A. P ⇝ CONST bex A (λx. -bex2 xs A P)
  -bex2 x A P ⇝ ∃ x ∈ A. P
  ∃ x ∈ A. P ⇐ CONST bex A (λx. P)

  ∃! x ∈ A. P ⇐ CONST bex1 A (λx. P)

  Setup of one point rules.
simproc-setup defined-bex (∃ x ∈ A. P x ∧ Q x) =
  ⟨fn - => Quantifier1.rearrange-Bex
    (fn ctxt => unfold-tac ctxt @{thms bex-def})⟩
simproc-setup defined-ball (∀ x ∈ A. P x → Q x) =
  ⟨fn - => Quantifier1.rearrange-Ball
    (fn ctxt => unfold-tac ctxt @{thms ball-def})⟩
lemma ballI [intro!]: [∧ x. x ∈ A ⇒ P x] ⇒ ∀ x ∈ A. P x
  by (simp add: ball-def)
lemma ballD [dest?]: [∀ x ∈ A. P x; x ∈ A] ⇒ P x
  by (simp add: ball-def)
lemma ballE:
  assumes ∀ x ∈ A. P x
  obtains ∧ x. x ∈ A ⇒ P x
  using assms unfolding ball-def by auto
lemma ballE' [elim]:
  assumes ∀ x ∈ A. P x
  obtains x ∉ A | P x
  using assms by (auto elim: ballE)

```

lemma *ball-iff-ex-mem* [*iff*]: $(\forall x \in A. P) \longleftrightarrow ((\exists x. x \in A) \longrightarrow P)$
by (*simp add: ball-def*)

lemma *ball-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: ball-def*)

lemma *ball-or-iff-ball-or* [*iff*]: $(\forall x \in A. P\ x \vee Q) \longleftrightarrow ((\forall x \in A. P\ x) \vee Q)$
by *auto*

lemma *ball-or-iff-or-ball* [*iff*]: $(\forall x \in A. P \vee Q\ x) \longleftrightarrow (P \vee (\forall x \in A. Q\ x))$
by *auto*

lemma *ball-imp-iff-imp-ball* [*iff*]: $(\forall x \in A. P \longrightarrow Q\ x) \longleftrightarrow (P \longrightarrow (\forall x \in A. Q\ x))$
by *auto*

lemma *ball-empty* [*iff*]: $\forall x \in \{\}. P\ x$ **by** *auto*

lemma *atomize-ball*:
 $(\bigwedge x. x \in A \implies P\ x) \equiv \text{Trueprop } (\forall x \in A. P\ x)$
by (*simp only: ball-def atomize-all atomize-imp*)

declare *atomize-ball*[*symmetric, rulify*]
declare *atomize-ball*[*symmetric, defn*]

lemma *bexI* [*intro*]: $\llbracket P\ x; x \in A \rrbracket \implies \exists x \in A. P\ x$
by (*simp add: bex-def, blast*)

corollary *bexI'*: $\llbracket x \in A; P\ x \rrbracket \implies \exists x \in A. P\ x \dots$

lemma *bexE* [*elim!*]: $\llbracket \exists x \in A. P\ x; \bigwedge x. \llbracket x \in A; P\ x \rrbracket \implies Q \rrbracket \implies Q$
unfolding *bex-def* **by** *blast*

lemma *bex-iff-ex-and* [*simp*]: $(\exists x \in A. P) \longleftrightarrow ((\exists x. x \in A) \wedge P)$
unfolding *bex-def* **by** *simp*

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

lemma *bex-and-iff-bex-and* [*simp*]: $(\exists x \in A. P\ x \wedge Q) \longleftrightarrow ((\exists x \in A. P\ x) \wedge Q)$
by *auto*

lemma *bex-and-iff-or-bex* [*simp*]: $(\exists x \in A. P \wedge Q\ x) \longleftrightarrow (P \wedge (\exists x \in A. Q\ x))$
by *auto*

lemma *not-bex-empty* [*iff*]: $\neg(\exists x \in \{\}. P\ x)$ **by** *auto*

lemma *ball-imp-iff-bex-imp* [*simp*]: $(\forall x \in A. P\ x \longrightarrow Q) \longleftrightarrow ((\exists x \in A. P\ x) \longrightarrow Q)$
by *auto*

lemma *not-ball-iff-bex-not* [*simp*]: $(\neg(\forall x \in A. P\ x)) \longleftrightarrow (\exists x \in A. \neg P\ x)$
by *auto*

lemma *not-bex-iff-ball-not* [*simp*]: $(\neg(\exists x \in A. P\ x)) \longleftrightarrow (\forall x \in A. \neg P\ x)$
by *auto*

lemma *bex1I* [*intro*]: $\llbracket P\ x; x \in A; \bigwedge z. \llbracket P\ z; z \in A \rrbracket \Longrightarrow z = x \rrbracket \Longrightarrow \exists!x \in A. P\ x$
by (*simp add: bex1-def, blast*)

lemma *bex1I'*: $\llbracket x \in A; P\ x; \bigwedge z. \llbracket P\ z; z \in A \rrbracket \Longrightarrow z = x \rrbracket \Longrightarrow \exists!x \in A. P\ x$
by *blast*

lemma *bex1E* [*elim!*]: $\llbracket \exists!x \in A. P\ x; \bigwedge x. \llbracket x \in A; P\ x \rrbracket \Longrightarrow Q \rrbracket \Longrightarrow Q$
by (*simp add: bex1-def, blast*)

lemma *bex1-triv* [*simp*]: $(\exists!x \in A. P) \longleftrightarrow ((\exists!x. x \in A) \wedge P)$
by (*auto simp add: bex1-def*)

lemma *bex1-iff*: $(\exists!x \in A. P\ x) \longleftrightarrow (\exists!x. x \in A \wedge P\ x)$
by (*auto simp add: bex1-def*)

lemma *bex1-cong* [*cong*]:
 $\llbracket A = A'; \bigwedge x. x \in A' \Longrightarrow P\ x \longleftrightarrow P'\ x \rrbracket \Longrightarrow (\exists!x \in A. P\ x) \longleftrightarrow (\exists!x \in A'. P'\ x)$
by (*simp add: bex1-def cong: conj-cong*)

lemma *bex-if-bex1*: $\exists!x \in A. P\ x \Longrightarrow \exists x \in A. P\ x$
by *auto*

lemma *ball-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 *auto*

lemma *antimono-ball-set*: *antimono* $(\lambda A. \forall x \in A. P\ x)$
by (*intro antimonoI*) *auto*

lemma *mono-ball-pred*: *mono* $(\lambda P. \forall x \in A. P\ x)$
by (*intro monoI*) *auto*

lemma *mono-bex-set*: *mono* ($\lambda A. \exists x \in A. P\ x$)
by (*intro monoI*) *auto*

lemma *mono-bex-pred*: *mono* ($\lambda P. \exists x \in A. P\ x$)
by (*intro monoI*) *auto*

8 Bounded definite description

definition *bthe* :: *set* \Rightarrow (*set* \Rightarrow *bool*) \Rightarrow *set*
where *bthe* *A* *P* \equiv *The* ($\lambda x. x \in A \wedge P\ x$)

bundle *hotg-bounded-the-syntax*
begin
syntax *-bthe* :: [*pttrn*, *set*, *bool*] \Rightarrow *set* ((*THE* - \in -/ -) [*0*, *0*, *10*] *10*)
end
bundle *no-hotg-bounded-the-syntax*
begin
no-syntax *-bthe* :: [*pttrn*, *set*, *bool*] \Rightarrow *set* ((*THE* - \in -/ -) [*0*, *0*, *10*] *10*)
end
unbundle *hotg-bounded-the-syntax*

translations *THE* $x \in A. P \rightleftharpoons$ *CONST* *bthe* *A* ($\lambda x. P$)

lemma *bthe-eqI* [*intro*]:
assumes *P* *a*
and $a \in A$
and $\bigwedge x. \llbracket x \in A; P\ x \rrbracket \Longrightarrow x = a$
shows (*THE* $x \in A. P\ x$) = *a*
unfolding *bthe-def* **by** (*auto intro: assms*)

lemma
bthe-memI: $\exists! x \in A. P\ x \Longrightarrow (THE\ x \in A. P\ x) \in A$ **and**
btheI: $\exists! x \in A. P\ x \Longrightarrow P\ (THE\ x \in A. P\ x)$
unfolding *bex1-def bthe-def* **by** (*auto simp: theI'[of $\lambda x. x \in A \wedge P\ x$]*)

end

9 Set Equality

theory *Equality*
imports *Subset*
begin

lemma *eqI* [*intro*]: ($\bigwedge x. x \in A \Longrightarrow x \in B$) \Longrightarrow ($\bigwedge x. x \in B \Longrightarrow x \in A$) $\Longrightarrow A = B$
by *auto*

```

lemma eqI':  $(\bigwedge x. x \in A \longleftrightarrow x \in B) \implies A = B$  by auto

lemma eqE:  $\llbracket A = B; \llbracket A \subseteq B ; B \subseteq A \rrbracket \implies P \rrbracket \implies P$  by blast

lemma eqD [dest]:  $A = B \implies (\bigwedge x. x \in A \longleftrightarrow x \in B)$  by auto

lemma ne-if-ex-mem-not-mem:  $\exists x. x \in A \wedge x \notin B \implies A \neq B$  by auto

lemma neD:  $A \neq B \implies \exists x. (x \in A \wedge x \notin B) \vee (x \notin A \wedge x \in B)$  by auto

end

theory Functions-Restrict
  imports Basic
begin

Summary If the input  $x$  is within the restricted domain, then  $f\ x$  is returned; otherwise, returns undefined.

consts fun-restrict ::  $('a \Rightarrow 'b) \Rightarrow 'c \Rightarrow 'a \Rightarrow 'b$ 

overloading
  fun-restrict-pred  $\equiv$  fun-restrict ::  $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'b$ 
begin
  definition fun-restrict-pred  $f\ P\ x \equiv$  if  $P\ x$  then  $f\ x$  else undefined
end

bundle fun-restrict-syntax
begin
notation fun-restrict  $((-)|(-) [1000])$ 
end
bundle no-fun-restrict-syntax
begin
no-notation fun-restrict  $((-)|(-) [1000])$ 
end

context
  includes fun-restrict-syntax
begin

lemma fun-restrict-eq [simp]:
  assumes  $P\ x$ 
  shows  $f|_P\ x = f\ x$ 
  using assms unfolding fun-restrict-pred-def by auto

lemma fun-restrict-eq-if-not [simp]:
  assumes  $\neg(P\ x)$ 
  shows  $f|_P\ x = \text{undefined}$ 
  using assms unfolding fun-restrict-pred-def by auto

```

```

end

overloading
  fun-restrict-set  $\equiv$  fun-restrict :: (set  $\Rightarrow$  'a)  $\Rightarrow$  set  $\Rightarrow$  set  $\Rightarrow$  'a
begin
  definition fun-restrict-set f X  $\equiv$  fun-restrict f (mem-of X) :: set  $\Rightarrow$  'a
end

lemma fun-restrict-set-eq-fun-restrict [simp]:
  fun-restrict (f :: set  $\Rightarrow$  'a) X = fun-restrict f (mem-of X)
  unfolding fun-restrict-set-def by auto

end

```

10 Replacement

```

theory Replacement
  imports
    Bounded-Quantifiers
    Equality
    Functions-Restrict
    Transport.Functions-Injective
begin

bundle hotg-repl-syntax
begin
syntax -repl ::  $\langle$ [set, pttrn, set]  $\Rightarrow$  set $\rangle$  ({- | / -  $\in$  -})
end
bundle no-hotg-repl-syntax
begin
no-syntax -repl ::  $\langle$ [set, pttrn, set]  $\Rightarrow$  set $\rangle$  ({- | / -  $\in$  -})
end
unbundle hotg-repl-syntax

translations
  {y | x  $\in$  A}  $\rightleftharpoons$  CONST repl A ( $\lambda x.$  y)

lemma app-mem-repl-if-mem [intro]: a  $\in$  A  $\implies$  f a  $\in$  {f x | x  $\in$  A}
  by auto

lemma bex-eq-app-if-mem-repl: b  $\in$  {f x | x  $\in$  A}  $\implies$   $\exists a \in A. b = f a$ 
  by auto

lemma replE [elim!]:

```


assumes $b \in \{f\ x \mid x \in A\}$
obtains x **where** $x \in A$ **and** $b = f\ x$
using *assms* **by** (*auto dest: bex-eq-app-if-mem-repl*)

lemma *repl-cong* [*cong*]:
 $\llbracket A = B; \bigwedge x. x \in B \implies f\ x = g\ x \rrbracket \implies \{f\ x \mid x \in A\} = \{g\ x \mid x \in B\}$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *repl-repl-eq-repl* [*simp*]: $\{g\ b \mid b \in \{f\ a \mid a \in A\}\} = \{g\ (f\ a) \mid a \in A\}$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *repl-eq-dom* [*simp*]: $\{x \mid x \in A\} = A$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *repl-eq-empty* [*simp*]: $\{f\ x \mid x \in \{\}\} = \{\}$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *repl-eq-empty-iff* [*iff*]: $\{f\ x \mid x \in A\} = \{\} \longleftrightarrow A = \{\}$
by *auto*

lemma *repl-subset-repl-if-subset-dom* [*intro!*]:
 $A \subseteq B \implies \{g\ y \mid y \in A\} \subseteq \{g\ y \mid y \in B\}$
by *auto*

lemma *ball-repl-iff-ball* [*iff*]: $(\forall x \in \{f\ x \mid x \in A\}. P\ x) \longleftrightarrow (\forall x \in A. P\ (f\ x))$
by *auto*

lemma *bex-repl-iff-bex* [*iff*]: $(\exists x \in \{f\ x \mid x \in A\}. P\ x) \longleftrightarrow (\exists x \in A. P\ (f\ x))$
by *auto*

lemma *mono-repl-set*: *mono* ($\lambda A. \{f\ x \mid x \in A\}$)
by (*intro monoI*) *auto*

10.1 Image

definition *image* $f\ A \equiv \{f\ x \mid x \in A\}$

lemma *image-eq-repl* [*simp*]: $\text{image}\ f\ A = \text{repl}\ A\ f$
unfolding *image-def* **by** *simp*

lemma *repl-fun-restrict-eq-repl* [*simp*]: $\{\text{fun-restrict}\ f\ A\ x \mid x \in A\} = \{f\ x \mid x \in A\}$
by *simp*

lemma *injective-image-if-injective*:
assumes *injective* f
shows *injective* ($\text{image}\ f$)
by (*intro injectiveI eqI*) (*use assms in <auto dest: injectiveD>*)

```

lemma injective-if-injective-image:
  assumes injective (image f)
  shows injective f
proof (rule injectiveI)
  fix X Y assume f X = f Y
  then have image f {X | - ∈ powerset {}} = image f {Y | - ∈ powerset {}} by
simp
  with assms show X = Y by (blast dest: injectiveD)
qed

corollary injective-image-iff-injective [iff]: injective (image f) ⟷ injective f
  using injective-image-if-injective injective-if-injective-image by blast

end

```

11 Unordered Pairs

theory *Unordered-Pairs*

imports

Powerset

Replacement

begin

We define an unordered pair *upair* using replacement. We then use it to define finite sets in `Finite_Sets.thy`.

definition *upair* *a b* \equiv {if *i* = {} then *a* else *b* | *i* ∈ *powerset* (*powerset* {})}

lemma *mem-upair-leftI* [*intro*]: *a* ∈ *upair a b* **unfolding** *upair-def* **by** *auto*

lemma *mem-upair-rightI* [*intro*]: *b* ∈ *upair a b* **unfolding** *upair-def* **by** *auto*

lemma *mem-upairE* [*elim!*]:

assumes *x* ∈ *upair a b*

obtains *x* = *a* | *x* = *b*

using *assms* **unfolding** *upair-def* **by** (*auto split: if-splits*)

lemma *mem-upair-iff*: *x* ∈ *upair a b* \longleftrightarrow *x* = *a* \vee *x* = *b* **by** *auto*

definition *insert* *x A* \equiv \bigcup (*upair A (upair x x)*)

lemma *mem-insert-leftI* [*intro*]: *x* ∈ *insert x A*

unfolding *insert-def* **by** *auto*

lemma *mem-insert-rightI* [*intro*]: *y* ∈ *A* \implies *y* ∈ *insert x A*

unfolding *insert-def* **by** *auto*

lemma *mem-insertE* [*elim*]:

assumes $y \in \text{insert } x \ A$
obtains $y = x \mid y \neq x \ y \in A$
using *assms* **unfolding** *insert-def* **by** *auto*

lemma *mem-insert-iff*: $y \in \text{insert } x \ A \longleftrightarrow y = x \vee y \in A$ **by** *auto*

lemma *not-mem-insert-if-not-mem-if-ne*: $\llbracket x \neq a; x \notin A \rrbracket \Longrightarrow x \notin \text{insert } a \ A$ **by** *auto*

lemma *insert-eq-if-mem* [*simp*]: $a \in A \Longrightarrow \text{insert } a \ A = A$ **by** *auto*

lemma *mem-insert-if-not-mem-imp-eq* [*intro!*]:
 $(a \notin B \Longrightarrow a = b) \Longrightarrow a \in \text{insert } b \ B$
by *auto*

lemma *insert-ne-empty* [*iff*]: $\text{insert } a \ B \neq \{\}$
by *auto*

lemma *insert-comm*: $\text{insert } x \ (\text{insert } y \ A) = \text{insert } y \ (\text{insert } x \ A)$
by *auto*

lemma *insert-insert-eq-insert* [*simp*]: $\text{insert } x \ (\text{insert } x \ A) = \text{insert } x \ A$
by *auto*

lemma *bex-insert-iff-or-bex* [*iff*]:
 $(\exists x \in \text{insert } a \ A. P \ x) \longleftrightarrow (P \ a \vee (\exists x \in A. P \ x))$
by *auto*

lemma *ball-insert-iff-and-ball* [*iff*]:
 $(\forall x \in \text{insert } a \ A. P \ x) \longleftrightarrow (P \ a \wedge (\forall x \in A. P \ x))$
by *auto*

lemma *mono-insert-set*: $\text{mono } (\text{insert } x)$
by (*intro monoI*) *auto*

lemma *insert-subset-iff-mem-subset* [*iff*]: $\text{insert } x \ A \subseteq B \longleftrightarrow x \in B \wedge A \subseteq B$
by *blast*

lemma *repl-insert-eq*: $\{f \ x \mid x \in \text{insert } x \ A\} = \text{insert } (f \ x) \ \{f \ x \mid x \in A\}$
by *auto*

end

12 Finite Sets

```

theory Finite-Sets
  imports Unordered-Pairs
begin

bundle hotg-finite-sets-syntax
begin
syntax -finset ::  $\langle \text{args} \Rightarrow \text{set} \rangle (\{(-)\})$ 
end
bundle no-hotg-finite-sets-syntax
begin
no-syntax -finset ::  $\langle \text{args} \Rightarrow \text{set} \rangle (\{(-)\})$ 
end
unbundle hotg-finite-sets-syntax
unbundle no-HOL-ascii-syntax

translations
   $\{x, xs\} \rightleftharpoons \text{CONST insert } x \{xs\}$ 
   $\{x\} \rightleftharpoons \text{CONST insert } x \{\}$ 

lemma singleton-eq-iff-eq [iff]:  $\{a\} = \{b\} \longleftrightarrow a = b$ 
  by auto

lemma subset-singleton-iff-eq-or-eq [iff]:  $A \subseteq \{a\} \longleftrightarrow A = \{\} \vee A = \{a\}$ 
  by auto

lemma singleton-mem-iff-eq [iff]:  $x \in \{a\} \longleftrightarrow x = a$  by auto

lemma powerset-empty-eq [simp]:  $\text{powerset } \{\} = \{\{\}\}$ 
  by auto

lemma powerset-singleton-eq [simp]:  $\text{powerset } \{a\} = \{\{\}, \{a\}\}$ 
  by auto

lemma powerset-powerset-empty-eq [simp]:  $\text{powerset } (\text{powerset } \{\}) = \{\{\}, \{\{\}\}\}$ 
  by simp

corollary powerset-singleton-elems [iff]:  $x \in \text{powerset } \{a\} \longleftrightarrow x = \{\} \vee x = \{a\}$ 
  by auto

corollary subset-singleton-iff [iff]:  $x \subseteq \{a\} \longleftrightarrow x = \{\} \vee x = \{a\}$  by auto

lemma singleton-subset-iff-mem [iff]:  $\{a\} \subseteq B \longleftrightarrow a \in B$ 
  by blast

lemma mem-upair-iff [iff]:  $x \in \{a, b\} \longleftrightarrow x = a \vee x = b$  by auto

lemma upair-eq-iff:  $\{a, b\} = \{c, d\} \longleftrightarrow (a = c \wedge b = d) \vee (a = d \wedge b = c)$ 

```

```

by auto

lemma upair-eq-singleton-iff [iff]:  $\{a, b\} = \{c\} \longleftrightarrow a = c \wedge b = c$ 
by (subst insert-insert-eq-insert[of c, symmetric]) (auto simp only: upair-eq-iff)

lemma singleton-eq-upair-iff [iff]:  $\{a\} = \{b, c\} \longleftrightarrow b = a \wedge c = a$ 
using upair-eq-singleton-iff by (auto dest: sym[of {a}])

upair x y and  $\{x, y\}$  are equal, and thus interchangeable in developments.

lemma upair-eq-insert-singleton [simp]:  $upair\ x\ y = \{x, y\}$ 
unfolding upair-def by (rule eqI) auto

```

12.1 Replacement

```

lemma repl-singleton-eq [simp]:  $\{f\ x \mid x \in \{a\}\} = \{f\ a\}$  by auto

```

end

13 Restricted Comprehension

```

theory Comprehension
imports
  Finite-Sets
  Order-Set
begin

unbundle no-HOL-ascii-syntax

definition collect ::  $\langle set \Rightarrow (set \Rightarrow bool) \Rightarrow set \rangle$ 
where  $collect\ A\ P \equiv \bigcup \{if\ P\ x\ then\ \{x\}\ else\ \{\}\ \mid x \in A\}$ 

bundle hotg-collect-syntax
begin
syntax -collect ::  $\langle idt \Rightarrow set \Rightarrow (set \Rightarrow bool) \Rightarrow set \rangle$   $((1\{- \in - \mid -\}))$ 
end
bundle no-hotg-collect-syntax
begin
no-syntax -collect ::  $\langle idt \Rightarrow set \Rightarrow (set \Rightarrow bool) \Rightarrow set \rangle$   $((1\{- \in - \mid -\}))$ 
end
unbundle hotg-collect-syntax

translations
   $\{x \in A \mid P\} \rightleftharpoons CONST\ collect\ A\ (\lambda x. P)$ 

```

```

lemma mem-collect-iff [iff]:  $x \in \{y \in A \mid P\ y\} \longleftrightarrow x \in A \wedge P\ x$ 
  by (auto simp: collect-def)

lemma mem-collectI [intro]:  $\llbracket x \in A; P\ x \rrbracket \Longrightarrow x \in \{y \in A \mid P\ y\}$  by auto

lemma mem-collectD:  $x \in \{y \in A \mid P\ y\} \Longrightarrow x \in A$  by auto

lemma mem-collectD':  $x \in \{y \in A \mid P\ y\} \Longrightarrow P\ x$  by auto

lemma collect-subset:  $\{x \in A \mid P\ x\} \subseteq A$  by blast

lemma collect-cong [cong]:
   $A = B \Longrightarrow (\bigwedge x. x \in B \Longrightarrow P\ x = Q\ x) \Longrightarrow \{x \in A \mid P\ x\} = \{x \in B \mid Q\ x\}$ 
  unfolding collect-def by simp

lemma collect-collect-eq [simp]:  $\text{collect } (\text{collect } A\ P)\ Q = \{x \in A \mid P\ x \wedge Q\ x\}$ 
  by auto

lemma collect-insert-eq:
   $\{x \in \text{insert } a\ B \mid P\ x\} = (\text{if } P\ a \text{ then insert } a\ \{x \in B \mid P\ x\} \text{ else } \{x \in B \mid P\ x\})$ 
  by auto

lemma mono-collect-set:  $\text{mono } (\lambda A. \{x \in A \mid P\ x\})$ 
  by (intro monoI) auto

lemma mono-collect-pred:  $\text{mono } (\lambda P. \{x \in A \mid P\ x\})$ 
  by (intro monoI) auto

end

```

14 Union and Intersection

```

theory Union-Intersection
  imports Comprehension
begin

definition inter  $A \equiv \{x \in \bigcup A \mid \forall y \in A. x \in y\}$ 

bundle hotg-inter-syntax begin notation inter ( $\bigcap$  - [90] 90) end
bundle no-hotg-inter-syntax begin no-notation inter ( $\bigcap$  - [90] 90) end
unbundle hotg-inter-syntax

```

Intersection is well-behaved only if the family is non-empty!

```

lemma mem-inter-iff [iff]:  $A \in \bigcap C \longleftrightarrow C \neq \{\} \wedge (\forall x \in C. A \in x)$ 

```

unfolding *inter-def* **by** *auto*

lemma *interD* [*dest*]: $\llbracket A \in \bigcap C; B \in C \rrbracket \implies A \in B$ **by** *auto*

lemma *union-empty-eq* [*iff*]: $\bigcup \{\} = \{\}$ **by** *auto*

lemma *inter-empty-eq* [*iff*]: $\bigcap \{\} = \{\}$ **by** *auto*

lemma *union-eq-empty-iff*: $\bigcup A = \{\} \longleftrightarrow A = \{\} \vee A = \{\{\}\}$
proof
 assume $\bigcup A = \{\}$
 show $A = \{\} \vee A = \{\{\}\}$
proof (*rule or-if-not-imp*)
 assume $A \neq \{\}$
 then obtain x where $x \in A$ **by** *auto*
 from $\langle \bigcup A = \{\} \rangle$ have [*simp*]: $\bigwedge x. x \in A \implies x = \{\}$ **by** *auto*
 with $\langle x \in A \rangle$ have $x = \{\}$ **by** *simp*
 with $\langle x \in A \rangle$ have [*simp*]: $\{\} \in A$ **by** *simp*
 show $A = \{\{\}\}$ **by** *auto*
qed
qed *auto*

lemma *union-eq-empty-iff'*: $\bigcup A = \{\} \longleftrightarrow (\forall B \in A. B = \{\})$ **by** *auto*

lemma *union-singleton-eq* [*simp*]: $\bigcup \{b\} = b$ **by** *auto*

lemma *inter-singleton-eq* [*simp*]: $\bigcap \{b\} = b$ **by** *auto*

lemma *subset-union-if-mem*: $B \in A \implies B \subseteq \bigcup A$ **by** *blast*

lemma *inter-subset-if-mem*: $B \in A \implies \bigcap A \subseteq B$ **by** *blast*

lemma *union-subset-iff*: $\bigcup A \subseteq C \longleftrightarrow (\forall x \in A. x \subseteq C)$ **by** *blast*

lemma *subset-inter-iff-all-mem-subset-if-ne-empty*:
 $A \neq \{\} \implies C \subseteq \bigcap A \longleftrightarrow (\forall x \in A. C \subseteq x)$
by *blast*

lemma *union-subset-if-all-mem-subset*: $(\bigwedge x. x \in A \implies x \subseteq C) \implies \bigcup A \subseteq C$ **by** *blast*

lemma *subset-inter-if-all-mem-subset-if-ne-empty*:
 $\llbracket A \neq \{\}; \bigwedge x. x \in A \implies C \subseteq x \rrbracket \implies C \subseteq \bigcap A$
using *subset-inter-iff-all-mem-subset-if-ne-empty* **by** *auto*

lemma *mono-union*: *mono union*
by (*intro monoI*) *auto*

lemma *antimono-inter*: $A \neq \{\} \implies A \subseteq A' \implies \bigcap A' \subseteq \bigcap A$
by *auto*

14.1 Indexed Union and Intersection:

bundle *hotg-idx-union-inter-syntax*

begin

syntax

-*idx-union* :: $\langle [pttrn, set, set \Rightarrow set] \Rightarrow set \rangle ((\exists \bigcup - \in -./ -) [0, 0, 10] 10)$

-*idx-inter* :: $\langle [pttrn, set, set \Rightarrow set] \Rightarrow set \rangle ((\exists \bigcap - \in -./ -) [0, 0, 10] 10)$

end

bundle *no-hotg-idx-union-inter-syntax*

begin

no-syntax

-*idx-union* :: $\langle [pttrn, set, set \Rightarrow set] \Rightarrow set \rangle ((\exists \bigcup - \in -./ -) [0, 0, 10] 10)$

-*idx-inter* :: $\langle [pttrn, set, set \Rightarrow set] \Rightarrow set \rangle ((\exists \bigcap - \in -./ -) [0, 0, 10] 10)$

end

unbundle *hotg-idx-union-inter-syntax*

translations

$\bigcup x \in A. B \Rightarrow \bigcup \{B \mid x \in A\}$

$\bigcap x \in A. B \Rightarrow \bigcap \{B \mid x \in A\}$

lemma *mem-idx-unionE* [*elim!*]:

assumes $b \in (\bigcup x \in A. B x)$

obtains x **where** $x \in A$ **and** $b \in B x$

using *assms* **by** *blast*

lemma *mem-idx-interD*:

assumes $b \in (\bigcap x \in A. B x)$ **and** $x \in A$

shows $b \in B x$

using *assms* **by** *blast*

lemma *idx-union-cong* [*cong*]:

$\llbracket A = B; \bigwedge x. x \in B \implies C x = D x \rrbracket \implies (\bigcup x \in A. C x) = (\bigcup x \in B. D x)$

by *simp*

lemma *idx-inter-cong* [*cong*]:

$\llbracket A = B; \bigwedge x. x \in B \implies C x = D x \rrbracket \implies (\bigcap x \in A. C x) = (\bigcap x \in B. D x)$

by *simp*

lemma *idx-union-const-eq-if-ne-empty*: $A \neq \{\} \implies (\bigcup x \in A. B) = B$

by (*rule eq-if-subset-if-subset*) *auto*

lemma *idx-inter-const-eq-if-ne-empty*: $A \neq \{\} \implies (\bigcap x \in A. B) = B$

by (*rule eq-if-subset-if-subset*) *auto*

lemma *idx-union-empty-dom-eq* [*simp*]: $(\bigcup x \in \{\}. B x) = \{\}$ **by** *auto*

lemma *idx-inter-empty-dom-eq* [simp]: $(\bigcap x \in \{\}. B\ x) = \{\}$ **by** *auto*

lemma *idx-union-empty-eq* [simp]: $(\bigcup x \in A. \{\}) = \{\}$ **by** *auto*

lemma *idx-inter-empty-eq* [simp]: $(\bigcap x \in A. \{\}) = \{\}$ **by** *blast*

lemma *idx-union-eq-union* [simp]: $(\bigcup x \in A. x) = \bigcup A$ **by** *auto*

lemma *idx-inter-eq-inter* [simp]: $(\bigcap x \in A. x) = \bigcap A$ **by** *auto*

lemma *idx-union-subset-iff*: $(\bigcup x \in A. B\ x) \subseteq C \longleftrightarrow (\forall x \in A. B\ x \subseteq C)$ **by** *blast*

lemma *subset-idx-inter-iff-if-ne-empty*:
 $C \neq \{\} \implies C \subseteq (\bigcap x \in A. B\ x) \longleftrightarrow (A \neq \{\} \wedge (\forall x \in A. C \subseteq B\ x))$
by *auto*

lemma *subset-idx-union-if-mem*: $x \in A \implies B\ x \subseteq (\bigcup x \in A. B\ x)$ **by** *blast*

lemma *idx-inter-subset-if-mem*: $x \in A \implies (\bigcap x \in A. B\ x) \subseteq B\ x$ **by** *blast*

lemma *idx-union-subset-if-all-mem-app-subset*:
 $(\bigwedge x. x \in A \implies B\ x \subseteq C) \implies (\bigcup x \in A. B\ x) \subseteq C$
by *blast*

lemma *subset-idx-inter-if-all-mem-subset-app-if-ne-empty*:
 $\llbracket A \neq \{\}; \bigwedge x. x \in A \implies C \subseteq B\ x \rrbracket \implies C \subseteq (\bigcap x \in A. B\ x)$
by *blast*

lemma *idx-union-singleton-eq* [simp]: $(\bigcup x \in A. \{x\}) = A$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *idx-union-flatten* [simp]:
 $(\bigcup x \in (\bigcup y \in A. B\ y). C\ x) = (\bigcup y \in A. \bigcup x \in B\ y. C\ x)$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *idx-union-const* [simp]: $(\bigcup y \in A. c) = (\text{if } A = \{\} \text{ then } \{\} \text{ else } c)$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *idx-inter-const* [simp]: $(\bigcap y \in A. c) = (\text{if } A = \{\} \text{ then } \{\} \text{ else } c)$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *idx-union-repl-eq-idx-union* [simp]: $(\bigcup y \in \{f\ x \mid x \in A\}. B\ y) = (\bigcup x \in A. B\ (f\ x))$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *idx-inter-repl-eq-idx-inter* [simp]: $(\bigcap x \in \{f\ x \mid x \in A\}. B\ x) = (\bigcap a \in A. B\ (f\ a))$

by *auto*

lemma *idx-union-repl-eq-repl-union*: $(\bigcup Y \in X. \{f x \mid x \in Y\}) = \{f x \mid x \in \bigcup X\}$
 by *auto*

lemma *repl-inter-subset-idx-inter-repl*: $\{f x \mid x \in \bigcap X\} \subseteq (\bigcap Y \in X. \{f x \mid x \in Y\})$
 by *auto*

lemma *idx-inter-union-eq-idx-inter-idx-inter*:
 $\{\} \notin A \implies (\bigcap x \in \bigcup A. B x) = (\bigcap y \in A. \bigcap x \in y. B x)$
 by (*auto iff: union-eq-empty-iff*)

lemma *idx-inter-idx-union-eq-idx-inter-idx-inter*:
 assumes $\bigwedge x. (x \in A \implies B x \neq \{\})$
 shows $(\bigcap z \in (\bigcup x \in A. B x). C z) = (\bigcap x \in A. \bigcap z \in B x. C z)$
proof (*rule eqI*)
 fix *x* assume $x \in (\bigcap z \in (\bigcup x \in A. B x). C z)$
 with *assms* show $x \in (\bigcap x \in A. \bigcap z \in B x. C z)$ by (*auto 5 0*)
next
 fix *x* assume *x-mem*: $x \in (\bigcap x \in A. \bigcap z \in B x. C z)$
 then have $A \neq \{\}$ by *auto*
 then obtain *y* where $y \in A$ by *auto*
 with *assms* have $B y \neq \{\}$ by *auto*
 with $\langle y \in A \rangle$ have $\{B x \mid x \in A\} \neq \{\{\}\}$ by *auto*
 with *x-mem* show $x \in (\bigcap z \in (\bigcup x \in A. B x). C z)$
 by (*auto simp: union-eq-empty-iff*)
qed

lemma *mono-idx-union*:
 assumes $A \subseteq A'$
 and $\bigwedge x. x \in A \implies B x \subseteq B' x$
 shows $(\bigcup x \in A. B x) \subseteq (\bigcup x \in A'. B' x)$
 using *assms* by *auto*

lemma *mono-antimono-idx-inter*:
 assumes $A \neq \{\}$
 and $A \subseteq A'$
 and $\bigwedge x. x \in A \implies B' x \subseteq B x$
 shows $(\bigcap x \in A'. B' x) \subseteq (\bigcap x \in A. B x)$
 using *assms* by (*intro subsetI*) *auto*

14.2 Binary Union and Intersection

definition *bin-union* $A B \equiv \bigcup \{A, B\}$

bundle *hotg-bin-union-syntax* **begin** notation *bin-union* (*infixl* \cup 70) **end**
bundle *no-hotg-bin-union-syntax* **begin** no-notation *bin-union* (*infixl* \cup 70) **end**

unbundle *hotg-bin-union-syntax*

definition *bin-inter* $A \ B \equiv \bigcap \{A, B\}$

bundle *hotg-bin-inter-syntax* **begin notation** *bin-inter* (**infixl** \cap 70) **end**

bundle *no-hotg-bin-inter-syntax* **begin no-notation** *bin-inter* (**infixl** \cap 70) **end**

unbundle *hotg-bin-inter-syntax*

lemma *mem-bin-union-iff* [*iff*]: $x \in A \cup B \longleftrightarrow x \in A \vee x \in B$

unfolding *bin-union-def* **by** *auto*

lemma *mem-bin-inter-iff* [*iff*]: $x \in A \cap B \longleftrightarrow x \in A \wedge x \in B$

unfolding *bin-inter-def* **by** *auto*

Binary Union **lemma** *mem-bin-union-if-mem-left* [*elim?*]: $c \in A \implies c \in A \cup B$

by *simp*

lemma *mem-bin-union-if-mem-right* [*elim?*]: $c \in B \implies c \in A \cup B$

by *simp*

lemma *bin-unionE* [*elim!*]:

assumes $c \in A \cup B$

obtains (*mem-left*) $c \in A$ | (*mem-right*) $c \in B$

using *assms* **by** *auto*

lemma *bin-unionE'* [*elim!*]:

assumes $c \in A \cup B$

obtains (*mem-left*) $c \in A$ | (*mem-right*) $c \in B$ **and** $c \notin A$

using *assms* **by** *auto*

lemma *mem-bin-union-if-mem-if-not-mem*: $(c \notin B \implies c \in A) \implies c \in A \cup B$

by *auto*

lemma *bin-union-comm*: $A \cup B = B \cup A$

by (*rule eq-if-subset-if-subset*) *auto*

lemma *bin-union-assoc*: $(A \cup B) \cup C = A \cup (B \cup C)$

by (*rule eq-if-subset-if-subset*) *auto*

lemma *bin-union-comm-left*: $A \cup (B \cup C) = B \cup (A \cup C)$ **by** *auto*

lemmas *bin-union-AC-rules* = *bin-union-comm bin-union-assoc bin-union-comm-left*

lemma *empty-bin-union-eq* [*iff*]: $\{\} \cup A = A$

by (*rule eq-if-subset-if-subset*) *auto*

lemma *bin-union-empty-eq* [iff]: $A \cup \{\} = A$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *singleton-bin-union-absorb* [simp]: $a \in A \implies \{a\} \cup A = A$
by *auto*

lemma *singleton-bin-union-eq-insert*[simp]: $\{x\} \cup A = \text{insert } x \ A$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *bin-union-singleton-eq-insert*[simp]: $A \cup \{x\} = \text{insert } x \ A$
using *singleton-bin-union-eq-insert* **by** (subst *bin-union-comm*)

lemma *mem-singleton-bin-union* [iff]: $a \in \{a\} \cup B$ **by** *auto*

lemma *mem-bin-union-singleton* [iff]: $b \in A \cup \{b\}$ **by** *auto*

lemma *bin-union-subset-iff* [iff]: $A \cup B \subseteq C \longleftrightarrow A \subseteq C \wedge B \subseteq C$
by *blast*

lemma *bin-union-eq-left-iff* [iff]: $A \cup B = A \longleftrightarrow B \subseteq A$
using *mem-bin-union-if-mem-right*[of - $B \ A$] **by** (auto simp only: *sym*[of $A \cup B$])

lemma *bin-union-eq-right-iff* [iff]: $A \cup B = B \longleftrightarrow A \subseteq B$
by (subst *bin-union-comm*) (fact *bin-union-eq-left-iff*)

lemma *subset-bin-union-left*: $A \subseteq A \cup B$ **by** *blast*

lemma *subset-bin-union-right*: $B \subseteq A \cup B$
by (subst *bin-union-comm*) (fact *subset-bin-union-left*)

lemma *bin-union-subset-if-subset-if-subset*: $\llbracket A \subseteq C; B \subseteq C \rrbracket \implies A \cup B \subseteq C$
by *blast*

lemma *bin-union-self-eq-self* [simp]: $A \cup A = A$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *bin-union-absorb*: $A \cup (A \cup B) = A \cup B$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *bin-union-eq-right-if-subset*: $A \subseteq B \implies A \cup B = B$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *bin-union-eq-left-if-subset*: $B \subseteq A \implies A \cup B = A$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *bin-union-subset-bin-union-if-subset*: $B \subseteq C \implies A \cup B \subseteq A \cup C$
by *auto*

lemma *bin-union-subset-bin-union-if-subset'*: $A \subseteq B \implies A \cup C \subseteq B \cup C$

by *auto*

lemma *bin-union-eq-empty-iff* [iff]: $(A \cup B = \{\}) \longleftrightarrow (A = \{\} \wedge B = \{\})$
 by *auto*

lemma *mono-bin-union-left*: *mono* $(\lambda A. A \cup B)$
 by (*intro monoI*) *auto*

lemma *mono-bin-union-right*: *mono* $(\lambda B. A \cup B)$
 by (*intro monoI*) *auto*

lemma *union-insert-eq-bin-union-union*: $\bigcup (\text{insert } X \ Y) = X \cup \bigcup Y$ by *auto*

Binary Intersection **lemma** *mem-bin-inter-if-mem-if-mem* [intro!]: $\llbracket c \in A; c \in B \rrbracket \implies c \in A \cap B$
 by *simp*

lemma *mem-bin-inter-if-mem-left*: $c \in A \cap B \implies c \in A$
 by *simp*

lemma *mem-bin-inter-if-mem-right*: $c \in A \cap B \implies c \in B$
 by *simp*

lemma *mem-bin-interE* [elim!]:
 assumes $c \in A \cap B$
 obtains $c \in A$ and $c \in B$
 using *assms* by *simp*

lemma *bin-inter-empty-iff* [iff]: $A \cap B = \{\} \longleftrightarrow (\forall a \in A. a \notin B)$
 by *auto*

lemma *bin-inter-comm*: $A \cap B = B \cap A$
 by *auto*

lemma *bin-inter-assoc*: $(A \cap B) \cap C = A \cap (B \cap C)$
 by *auto*

lemma *bin-inter-comm-left*: $A \cap (B \cap C) = B \cap (A \cap C)$
 by *auto*

lemmas *bin-inter-AC-rules* = *bin-inter-comm bin-inter-assoc bin-inter-comm-left*

lemma *empty-bin-inter-eq-empty* [iff]: $\{\} \cap B = \{\}$
 by *auto*

lemma *bin-inter-empty-eq-empty* [iff]: $A \cap \{\} = \{\}$
 by *auto*

lemma *bin-inter-subset-iff* [iff]: $C \subseteq A \cap B \longleftrightarrow C \subseteq A \wedge C \subseteq B$

by *blast*

lemma *bin-inter-subset-left* [iff]: $A \cap B \subseteq A$
by *blast*

lemma *bin-inter-subset-right* [iff]: $A \cap B \subseteq B$
by *blast*

lemma *subset-bin-inter-if-subset-if-subset*: $\llbracket C \subseteq A; C \subseteq B \rrbracket \implies C \subseteq A \cap B$
by *blast*

lemma *bin-inter-self-eq-self* [iff]: $A \cap A = A$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *bin-inter-absorb* [iff]: $A \cap (A \cap B) = A \cap B$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *bin-inter-eq-right-if-subset*: $B \subseteq A \implies A \cap B = B$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *bin-inter-eq-left-if-subset*: $A \subseteq B \implies A \cap B = A$
by (subst *bin-inter-comm*) (fact *bin-inter-eq-right-if-subset*)

lemma *bin-inter-bin-union-distrib*: $(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *bin-inter-bin-union-distrib'*: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *bin-union-bin-inter-distrib*: $(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *bin-union-bin-inter-distrib'*: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
by (rule *eq-if-subset-if-subset*) *auto*

lemma *bin-inter-eq-left-iff-subset*: $A \subseteq B \longleftrightarrow A \cap B = A$
by *auto*

lemma *bin-inter-eq-right-iff-subset*: $A \subseteq B \longleftrightarrow B \cap A = A$
by *auto*

lemma *bin-inter-bin-union-assoc-iff*:
 $(A \cap B) \cup C = A \cap (B \cup C) \longleftrightarrow C \subseteq A$
by *auto*

lemma *bin-inter-bin-union-swap3*:
 $(A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A)$
by *auto*

lemma *mono-bin-inter-left*: $\text{mono } (\lambda A. A \cap B)$
by (*intro monoI*) *auto*

lemma *mono-bin-inter-right*: $\text{mono } (\lambda B. A \cap B)$
by (*intro monoI*) *auto*

lemma *inter-insert-eq-bin-inter-inter*: $Y \neq \{\} \implies \bigcap (\text{insert } X \ Y) = X \cap \bigcap Y$ **by** *auto*

Comprehension lemma *collect-eq-bin-inter* [*simp*]: $\{a \in A \mid a \in A'\} = A \cap A'$ **by** *auto*

lemma *collect-bin-union-eq*:
 $\{x \in A \cup B \mid P \ x\} = \{x \in A \mid P \ x\} \cup \{x \in B \mid P \ x\}$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *collect-bin-inter-eq*:
 $\{x \in A \cap B \mid P \ x\} = \{x \in A \mid P \ x\} \cap \{x \in B \mid P \ x\}$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *bin-inter-collect-absorb* [*iff*]:
 $A \cap \{x \in A \mid P \ x\} = \{x \in A \mid P \ x\}$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *collect-idx-union-eq-union-collect* [*simp*]:
 $\{y \in (\bigcup x \in A. B \ x) \mid P \ y\} = (\bigcup x \in A. \{y \in B \ x \mid P \ y\})$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *bin-inter-collect-left-eq-collect*:
 $\{x \in A \mid P \ x\} \cap B = \{x \in A \cap B \mid P \ x\}$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *bin-inter-collect-right-eq-collect*:
 $A \cap \{x \in B \mid P \ x\} = \{x \in A \cap B \mid P \ x\}$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *collect-and-eq-inter-collect*:
 $\{x \in A \mid P \ x \wedge Q \ x\} = \{x \in A \mid P \ x\} \cap \{x \in A \mid Q \ x\}$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *collect-or-eq-union-collect*:
 $\{x \in A \mid P \ x \vee Q \ x\} = \{x \in A \mid P \ x\} \cup \{x \in A \mid Q \ x\}$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *union-bin-union-eq-bin-union-union*: $\bigcup (A \cup B) = \bigcup A \cup \bigcup B$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *union-bin-inter-subset-bin-inter-union*: $\bigcup (A \cap B) \subseteq \bigcup A \cap \bigcup B$

by *blast*

lemma *union--disjoint-iff*: $\bigcup C \cap A = \{\} \longleftrightarrow (\forall B \in C. B \cap A = \{\})$
by *blast*

lemma *subset-idx-union-iff-eq*:
 $A \subseteq (\bigcup i \in I. B\ i) \longleftrightarrow A = (\bigcup i \in I. A \cap B\ i)$ (**is** $A \subseteq ?lhs\text{-}union \longleftrightarrow A = ?rhs\text{-}union$)
proof
assume $A\text{-eq}$: $A = ?rhs\text{-}union$
show $A \subseteq ?lhs\text{-}union$
proof (*rule subsetI*)
fix a **assume** $a \in A$
with $A\text{-eq}$ **have** $a \in ?rhs\text{-}union$ **by** *simp*
then obtain x **where** $x \in I$ **and** $a \in A \cap B\ x$ **by** *auto*
then show $a \in ?lhs\text{-}union$ **by** *auto*
qed
qed (*auto 5 0 intro! eqI*)

lemma *bin-inter-union-eq-idx-union-inter*: $\bigcup B \cap A = (\bigcup C \in B. C \cap A)$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *bin-union-inter-subset-inter-bin-inter*:
 $\llbracket z \in A; z \in B \rrbracket \implies \bigcap A \cup \bigcap B \subseteq \bigcap (A \cap B)$
by *blast*

lemma *inter-bin-union-eq-bin-inter-inter*:
 $\llbracket A \neq \{\}; B \neq \{\} \rrbracket \implies \bigcap (A \cup B) = \bigcap A \cap \bigcap B$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *idx-union-insert-dom-eq-bin-union-idx-union*: $(\bigcup i \in insert\ A\ B. C\ i) = C$
 $A \cup (\bigcup i \in B. C\ i)$
by *auto*

lemma *idx-inter-insert-dom-eq-bin-inter-idx-inter*:
assumes $B \neq \{\}$
shows $(\bigcap i \in insert\ A\ B. C\ i) = C \cap A \cap (\bigcap i \in B. C\ i)$
using *assms* **by** *auto*

lemma *idx-union-bin-union-dom-eq-bin-union-idx-union*:
 $(\bigcup i \in A \cup B. C\ i) = (\bigcup i \in A. C\ i) \cup (\bigcup i \in B. C\ i)$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *idx-inter-bin-inter-dom-eq-bin-inter-idx-inter*:
 $(\bigcap i \in I \cup J. A\ i) = ($
 $\quad if\ I = \{\} \text{ then } \bigcap j \in J. A\ j$
 $\quad else\ if\ J = \{\} \text{ then } \bigcap i \in I. A\ i$
 $\quad else\ (\bigcap i \in I. A\ i) \cap (\bigcap j \in J. A\ j)$
 $)$


```

by (rule eq-if-subset-if-subset) auto

lemma idx-union-bin-inter-eq-bin-inter-idx-union [simp]:
   $(\bigcup i \in I. A \cap B i) = A \cap (\bigcup i \in I. B i)$ 
by (rule eq-if-subset-if-subset) auto

lemma idx-inter-bin-union-eq-bin-union-idx-inter [simp]:
   $I \neq \{\} \implies (\bigcap i \in I. A \cup B i) = A \cup (\bigcap i \in I. B i)$ 
by (rule eq-if-subset-if-subset) auto

lemma idx-union-idx-union-bin-inter-eq-bin-inter-idx-union [simp]:
   $(\bigcup i \in I. \bigcup j \in J. A i \cap B j) = (\bigcup i \in I. A i) \cap (\bigcup j \in J. B j)$ 
by (rule eq-if-subset-if-subset) auto

lemma idx-inter-idx-inter-bin-union-eq-bin-union-idx-inter [simp]:
   $\llbracket I \neq \{\}; J \neq \{\} \rrbracket \implies$ 
   $(\bigcap i \in I. \bigcap j \in J. A i \cup B j) = (\bigcap i \in I. A i) \cup (\bigcap j \in J. B j)$ 
by (rule eq-if-subset-if-subset) auto

lemma idx-union-bin-union-eq-bin-union-idx-union [simp]:
   $(\bigcup i \in I. A i \cup B i) = (\bigcup i \in I. A i) \cup (\bigcup i \in I. B i)$ 
by (rule eq-if-subset-if-subset) auto

lemma idx-inter-bin-inter-eq-bin-inter-idx-inter [simp]:
   $I \neq \{\} \implies (\bigcap i \in I. A i \cap B i) = (\bigcap i \in I. A i) \cap (\bigcap i \in I. B i)$ 
by (rule eq-if-subset-if-subset) auto

lemma idx-union-bin-inter-subset-bin-inter-idx-union:
   $(\bigcup z \in I \cap J. A z) \subseteq (\bigcup z \in I. A z) \cap (\bigcup z \in J. A z)$ 
by blast

lemma idx-union-union-eq-idx-union-idx-union [simp]:  $(\bigcup x \in \bigcup X. f x) = (\bigcup x \in$ 
 $X. \bigcup y \in x. f y)$ 
by auto

end

```

15 Well-Foundedness of Sets

```

theory Foundation
  imports
    Mem-Transitive-Closed-Base
    Union-Intersection
begin

```

lemma *foundation-if-ne-empty*: $X \neq \{\}$ $\implies \exists Y \in X. Y \cap X = \{\}$
using *Axioms.mem-induction*[**where** $?P = \lambda x. x \notin X$] **by** *blast*

lemma *foundation-if-ne-empty'*: $X \neq \{\} \implies \exists Y \in X. \neg(\exists y \in Y. y \in X)$
proof –
assume $X \neq \{\}$
with *foundation-if-ne-empty* **obtain** Y **where** $Y \in X$ **and** $Y \cap X = \{\}$ **by** *auto*
thus $\exists Y \in X. \neg(\exists y \in Y. y \in X)$ **by** *auto*
qed

lemma *empty-or-foundation*: $X = \{\} \vee (\exists Y \in X. \forall y \in Y. y \notin X)$
using *foundation-if-ne-empty* **by** *auto*

lemma *empty-mem-if-mem-trans-closed*:
assumes *mem-trans-closed* X
and $X \neq \{\}$
shows $\{\} \in X$
proof (*rule ccontr*)
from *foundation-if-ne-empty* $\langle X \neq \{\} \rangle$
obtain A **where** $A \in X$ **and** *X-foundation*: $\forall a \in A. a \notin X$ **by** *auto*
assume $\{\} \notin X$
with $\langle A \in X \rangle$ **have** $A \neq \{\}$ **by** *auto*
then obtain a **where** $a \in A$ **by** *auto*
with *mem-trans-closedD*[*OF* $\langle \text{mem-trans-closed } X \rangle \langle A \in X \rangle$] **have** $a \in X$ **by** *auto*
with *X-foundation* $\langle a \in A \rangle$ **show** *False* **by** *auto*
qed

lemma *not-mem-if-mem*:
assumes $a \in b$
shows $b \notin a$
proof (*rule ccontr*)
presume $b \in a$
consider (*empty*) $\{a, b\} = \{\}$ | (*ne-empty*) $\exists c \in \{a, b\}. \forall d \in c. d \notin \{a, b\}$
using *empty-or-foundation*[*of* $\{a, b\}$] **by** *simp*
with $\langle b \in a \rangle$ **assms** **show** *False* **by** *cases auto*
qed *auto*

lemma *not-mem-self* [*iff*]: $a \notin a$ **using** *not-mem-if-mem* **by** *blast*

lemma *bin-union-singleton-self-ne-self* [*iff*]: $A \cup \{A\} \neq A$ **by** *auto*

lemma *bin-inter-singleton-self-eq-empty* [*simp*]: $A \cap \{A\} = \{\}$ **by** *auto*

lemma *ne-if-mem*: $a \in A \implies a \neq A$
using *not-mem-self* **by** *blast*

```

lemma not-mem-if-eq:  $a = A \implies a \notin A$ 
  by simp

lemma not-mem-if-mem-if-mem:
  assumes  $a \in b$   $b \in c$ 
  shows  $c \notin a$ 
proof
  assume  $c \in a$ 
  let  $?X = \{a, b, c\}$ 
  have  $?X \neq \{\}$  by simp
  from foundation-if-ne-empty[OF this] obtain  $Y$  where  $Y \in ?X$   $Y \cap ?X = \{\}$ 
  by blast
  from  $\langle Y \in ?X \rangle$  have  $Y = a \vee Y = b \vee Y = c$  by auto
  with assms  $\langle c \in a \rangle$  have  $a \in Y \vee b \in Y \vee c \in Y$  by blast
  with  $\langle Y \cap ?X = \{\} \rangle$  show False by blast
qed

lemma mem-double-induct:
  assumes  $\bigwedge X Y. \llbracket \bigwedge x. x \in X \implies P\ x\ Y; \bigwedge y. y \in Y \implies P\ X\ y \rrbracket \implies P\ X\ Y$ 
  shows  $P\ X\ Y$ 
proof (induction X arbitrary: Y rule: mem-induction)
  case (mem X)
  then show  $?case$  by (induction Y rule: mem-induction) (auto intro: assms)
qed

lemma insert-ne-self [iff]:  $\text{insert } x\ A \neq x$ 
  by (rule ne-if-mem[symmetric]) auto

end

```

16 Transfinite Recursion

```

theory Transfinite-Recursion
  imports
    Functions-Restrict
begin

```

Summary We give the axiomatization of transfinite recursion from [3], https://foss.heptapod.net/isa-afp/afp-devel/-/blob/06458dfa40c7b4aaeb855a37ae77993cb4c8c18/thys/ZFC_in_HOL/ZFC_in_HOL.thy#L1151.

```

axiomatization transrec ::  $((\text{set} \Rightarrow 'a) \Rightarrow \text{set} \Rightarrow 'a) \Rightarrow \text{set} \Rightarrow 'a$ 
  where transrec-eq:  $\text{transrec } f\ X = f\ (\text{fun-restrict } (\text{transrec } f)\ X)\ X$ 

end

```

17 Transitive Closure With Respect To Membership

```

theory Mem-Transitive-Closure
  imports
    Foundation
    Transfinite-Reursion
begin

```

Summary The transitive closure of a set X is the set that contains as its members all sets that are transitively contained in X . In particular, each such set is transitively closed.

We follow the approach from [3], https://foss.heptapod.net/isa-afp/afp-devel/-/blob/06458dfa40c7b4aaab855a37ae77993cb4c8c18/thys/ZFC_in_HOL/ZFC_Cardinals.thy#L410.

definition *mem-trans-closure* \equiv *transrec* ($\lambda f X. X \cup (\bigcup x \in X. f x)$)

lemma *mem-trans-closure-eq-bin-union-idx-union*:
mem-trans-closure $X = X \cup (\bigcup x \in X. \text{mem-trans-closure } x)$
by (*simp add: mem-trans-closure-def transrec-eq* [**where** $?X=X$])

corollary *subset-mem-trans-closure-self*: $X \subseteq \text{mem-trans-closure } X$
by (*auto simp: mem-trans-closure-eq-bin-union-idx-union* [**where** $?X=X$])

corollary *mem-mem-trans-closure-if-mem*: $X \in Y \implies X \in \text{mem-trans-closure } Y$
using *subset-mem-trans-closure-self* **by** *blast*

corollary *mem-mem-trans-closure-if-mem-idx-union*:
assumes $X \in (\bigcup x \in Y. \text{mem-trans-closure } x)$
shows $X \in \text{mem-trans-closure } Y$
using *assms* **by** (*subst mem-trans-closure-eq-bin-union-idx-union*) *auto*

lemma *mem-mem-trans-closureE* [*elim*]:
assumes $X \in \text{mem-trans-closure } Y$
obtains (*mem*) $X \in Y \mid (\text{mem-trans-closure}) y$ **where** $y \in Y \ X \in \text{mem-trans-closure } y$
using *assms* **by** (*subst (asm) mem-trans-closure-eq-bin-union-idx-union*) *auto*

lemma *mem-mem-trans-closure-iff-mem-or-mem*:
 $X \in \text{mem-trans-closure } Y \iff X \in Y \vee (X \in (\bigcup y \in Y. \text{mem-trans-closure } y))$
by (*subst mem-trans-closure-eq-bin-union-idx-union*) *auto*

lemma *mem-trans-closure-empty-eq-empty* [*simp*]: $\text{mem-trans-closure } \{\} = \{\}$
by (*simp add: mem-trans-closure-eq-bin-union-idx-union* [**where** $?X=\{\}$])

```

lemma mem-trans-closure-eq-empty-iff-eq-empty [iff]: mem-trans-closure  $X = \{\}$ 
 $\longleftrightarrow X = \{\}$ 
  using subset-mem-trans-closure-self by auto

lemma mem-trans-closed-mem-trans-closure: mem-trans-closed (mem-trans-closure
 $X$ )
proof (induction  $X$ )
  case (mem  $X$ )
  show ?case
  proof (rule mem-trans-closedI')
    fix  $x\ y$  assume  $x \in \text{mem-trans-closure } X\ y \in x$ 
    then show  $y \in \text{mem-trans-closure } X$ 
    proof (cases rule: mem-mem-trans-closureE)
      case mem
      have  $y \in \text{mem-trans-closure } x$  using  $\langle y \in x \rangle$  subset-mem-trans-closure-self
by blast
      with mem show ?thesis by (subst mem-trans-closure-eq-bin-union-idx-union)
blast
      next
      case mem-trans-closure
      with  $\langle y \in x \rangle$  mem.IH show ?thesis by (subst mem-trans-closure-eq-bin-union-idx-union)
blast
      qed
    qed
  qed

lemma not-mem-mem-trans-closure-self [iff]:  $X \notin \text{mem-trans-closure } X$ 
proof
  assume  $X \in \text{mem-trans-closure } X$ 
  then show False
  proof (cases rule: mem-mem-trans-closureE)
    case (mem-trans-closure  $x$ )
    with mem-trans-closed-mem-trans-closure show ?thesis by (induction  $X$  arbitrary:  $x$ ) blast
    qed auto
  qed

lemma mem-trans-closure-le-if-le-if-mem-trans-closed:
 $\llbracket \text{mem-trans-closed } X; Y \leq X \rrbracket \implies \text{mem-trans-closure } Y \leq X$ 
proof (induction  $Y$ )
  case (mem  $Y$ )
  show ?case
  proof (cases  $Y = \{\}$ )
    case False
    with mem have  $(\bigcup y \in Y. \text{mem-trans-closure } y) \leq X$  by auto
    with mem.prems show ?thesis by (simp add: mem-trans-closure-eq-bin-union-idx-union[of
 $Y$ ])
    qed auto
  
```

qed

lemma *mem-mem-trans-closure-if-mem-if-mem-mem-trans-closure*:
assumes $X \in \text{mem-trans-closure } Y$
and $Y \in Z$
shows $X \in \text{mem-trans-closure } Z$
using *assms* **by** (*auto iff: mem-mem-trans-closure-iff-mem-or-mem[of X Z]*)

The next lemma demonstrates a transitivity property.

lemma *mem-mem-trans-closure-trans*:
assumes $X \in \text{mem-trans-closure } Y$
and $Y \in \text{mem-trans-closure } Z$
shows $X \in \text{mem-trans-closure } Z$
using *assms*
proof (*induction Z*)
case (*mem Z*)
show ?*case*
proof (*cases Z = {}*)
case *False*
with *mem* **obtain** *z* **where** $z \in Z$ $X \in \text{mem-trans-closure } z$ **by** *auto*
with *mem* **show** ?*thesis* **using** *mem-mem-trans-closure-if-mem-if-mem-mem-trans-closure*
by *auto*
qed (*use mem in simp*)
qed

end

18 Less-Than and Less-Than or Equal Orders

theory *Less-Than*
imports
Transport.Partial-Orders
Transport.HOL-Syntax-Bundles-Groups
Transport.HOL-Syntax-Bundles-Orders
Mem-Transitive-Closure
begin

Summary We define less and less-than or equal on sets and then show that less is a preoder and the latter is a partial order.

A set X is smaller than a set Y if X is contained in the transitive closure of Y ; cf. *mem-trans-closure*.

abbreviation *zero-set* $\equiv \{\}$
abbreviation *one-set* $\equiv \{\text{zero-set}\}$
abbreviation *two-set* $\equiv \{\text{zero-set}, \text{one-set}\}$

```

bundle hotg-set-zero-syntax begin notation zero-set (0) end
bundle no-hotg-set-zero-syntax begin no-notation zero-set (0) end

```

```

bundle hotg-set-one-syntax begin notation one-set (1) end
bundle no-hotg-set-one-syntax begin no-notation one-set (1) end

```

```

bundle hotg-set-two-syntax begin notation two-set (2) end
bundle no-hotg-set-two-syntax begin no-notation two-set (2) end

```

```

unbundle
  hotg-set-zero-syntax
  hotg-set-one-syntax
  hotg-set-two-syntax
unbundle
  no-HOL-ascii-syntax
  no-HOL-groups-syntax

```

Less-Than Order We follow the definition by Kirby [2].

definition $lt\ X\ Y \equiv X \in mem\text{-}trans\text{-}closure\ Y$

```

bundle hotg-lt-syntax begin notation lt (infix < 50) end
bundle no-hotg-lt-syntax begin no-notation lt (infix < 50) end
unbundle hotg-lt-syntax
unbundle no-HOL-order-syntax

```

lemma *lt-iff-mem-trans-closure*: $X < Y \longleftrightarrow X \in mem\text{-}trans\text{-}closure\ Y$
unfolding *lt-def* **by** *simp*

lemma *lt-if-mem-trans-closure*:
assumes $X \in mem\text{-}trans\text{-}closure\ Y$
shows $X < Y$
using *assms* **unfolding** *lt-iff-mem-trans-closure* **by** *simp*

corollary *lt-if-mem*:
assumes $X \in Y$
shows $X < Y$
using *assms* *subset-mem-trans-closure-self* *lt-if-mem-trans-closure* **by** *auto*

lemma *mem-trans-closure-if-lt*:
assumes $X < Y$
shows $X \in mem\text{-}trans\text{-}closure\ Y$
using *assms* **unfolding** *lt-iff-mem-trans-closure* **by** *simp*

lemma *lt-if-lt-if-mem* [*trans*]:
assumes $x \in X$
and $X < Y$
shows $x < Y$
using *assms* *mem-trans-closed-mem-trans-closure* **unfolding** *lt-iff-mem-trans-closure*
by *auto*

lemma *lt-trans* [*trans*]:
 assumes $X < Y$
 and $Y < Z$
 shows $X < Z$
 using *assms* **unfolding** *lt-iff-mem-trans-closure* **by** (*rule mem-mem-trans-closure-trans*)

corollary *transitive-lt*: *transitive* ($<$)
 using *lt-trans* **by** *blast*

The lemma demonstrates the anti-reflexivity of less.

lemma *not-lt-self* [*iff*]: $\neg(X < X)$
unfolding *lt-iff-mem-trans-closure* **by** *auto*

lemma *not-lt-zero* [*iff*]: $\neg(X < 0)$
unfolding *lt-iff-mem-trans-closure* **by** *auto*

lemma *zero-lt-if-ne-zero* [*iff*]:
 assumes $X \neq 0$
 shows $0 < X$
 using *assms* *mem-trans-closed-mem-trans-closure*
by (*intro lt-if-mem-trans-closure empty-mem-if-mem-trans-closed*) *auto*

Less-Than or Equal Order **definition** *le* $X Y \equiv X < Y \vee X = Y$

bundle *hotg-le-syntax* **begin notation** *le* (*infix* \leq 60) **end**
bundle *no-hotg-le-syntax* **begin no-notation** *le* (*infix* \leq 60) **end**
unbundle *hotg-le-syntax*

lemma *le-if-lt*:
 assumes $X < Y$
 shows $X \leq Y$
 using *assms* **unfolding** *le-def* **by** *auto*

lemma *le-self* [*iff*]: $X \leq X$ **unfolding** *le-def* **by** *simp*

lemma *leE*:
 assumes $X \leq Y$
 obtains $(lt) X < Y \mid (eq) X = Y$
 using *assms* **unfolding** *le-def* **by** *auto*

corollary *le-iff-lt-or-eq*: $X \leq Y \longleftrightarrow X < Y \vee X = Y$
 using *le-if-lt* *leE* **by** *blast*

lemma *le-trans* [*trans*]:
 assumes $X \leq Y$
 and $Y \leq Z$
 shows $X \leq Z$
 using *assms* *lt-trans* **unfolding** *le-iff-lt-or-eq* **by** *auto*

corollary *reflexive-le*: *reflexive* (\leq) **by** *auto*

corollary *transitive-le*: *transitive* (\leq)
using *le-trans* **by** *blast*

corollary *preorder-le*: *preorder* (\leq)
using *reflexive-le transitive-le* **by** *blast*

lemma *zero-le* [*iff*]: $0 \leq X$ **by** (*subst le-iff-lt-or-eq*) *auto*

lemma *lt-mem-leE*:
assumes $X < Y$
obtains y **where** $y \in Y$ $X \leq y$
using *assms* **unfolding** *le-iff-lt-or-eq lt-iff-mem-trans-closure* **by** *auto*

lemma *lt-if-mem-if-le* [*trans*]:
assumes $X \leq Y$
and $Y \in Z$
shows $X < Z$
using *assms mem-trans-closure-eq-bin-union-idx-union*[*of Z*]
unfolding *le-iff-lt-or-eq lt-iff-mem-trans-closure*
by *auto*

corollary *lt-iff-bex-le*: $X < Y \longleftrightarrow (\exists y \in Y. X \leq y)$
by (*auto elim: lt-mem-leE intro: lt-if-mem-if-le*)

lemma *lt-if-lt-if-le* [*trans*]:
assumes $X \leq Y$
and $Y < Z$
shows $X < Z$
using *assms mem-trans-closure-eq-bin-union-idx-union*[*of Z*] *mem-mem-trans-closure-trans*
unfolding *le-iff-lt-or-eq lt-iff-mem-trans-closure*
by *blast*

lemma *lt-if-le-if-lt* [*trans*]:
assumes $X < Y$
and $Y \leq Z$
shows $X < Z$
using *assms mem-trans-closure-eq-bin-union-idx-union*[*of Z*] *mem-mem-trans-closure-trans*
unfolding *le-iff-lt-or-eq lt-iff-mem-trans-closure*
by *blast*

lemma *not-le-if-lt*: $X < Y \implies \neg(Y \leq X)$
using *lt-trans le-iff-lt-or-eq* **by** *auto*

lemma *not-lt-if-le*: $X \leq Y \implies \neg(Y < X)$
using *not-le-if-lt* **by** *auto*

lemma *antisymmetric-le*: *antisymmetric* (\leq)
unfolding *le-iff-lt-or-eq* **using** *lt-trans* **by** *auto*

corollary *partial-order-le*: *partial-order* (\leq)
using *preorder-le antisymmetric-le* **by** *blast*

These next lemmas show some relationships between ($<$), (\leq) and ($=$).

lemma *ne-if-lt*:
assumes $X < Y$
shows $X \neq Y$
using *assms* **by** *auto*

lemma *lt-if-ne-if-le*:
assumes $X \leq Y$
and $X \neq Y$
shows $X < Y$
using *assms* **unfolding** *le-iff-lt-or-eq* **by** *auto*

corollary *lt-iff-le-and-ne*: $X < Y \longleftrightarrow X \leq Y \wedge X \neq Y$
using *le-if-lt ne-if-lt lt-if-ne-if-le* **by** *blast*

lemma *le-if-eq*: $X = Y \implies X \leq Y$
by *simp*

lemma *not-lt-if-not-le-or-eq*: $\neg(X < Y) \longleftrightarrow \neg(X \leq Y) \vee X = Y$
unfolding *le-iff-lt-or-eq* **by** *auto*

The following sets up automation for goals involving the (\leq) and ($<$) relations.

local-setup \langle
HOL-Order-Tac.declare-order {
 $ops = \{eq = @\{term \langle(=) :: set \Rightarrow set \Rightarrow bool\rangle\}, le = @\{term \langle(\leq)\rangle\}, lt = @\{term \langle(<)\rangle\},$
 $thms = \{trans = @\{thm le-trans\}, refl = @\{thm le-self\}, eqD1 = @\{thm le-if-eq\},$
 $eqD2 = @\{thm le-if-eq[OF sym]\}, antisym = @\{thm antisymmetricD[OF antisymmetric-le]\},$
 $contr = @\{thm notE\},$
 $conv-thms = \{less-le = @\{thm eq-reflection[OF lt-iff-le-and-ne]\},$
 $nless-le = @\{thm eq-reflection[OF not-lt-if-not-le-or-eq]\}$
 $\}$
 \rangle

end

19 Generalised Addition

```

theory SAddition
  imports
    Less-Than
begin

```

Summary Translation of generalised set addition from [2] and [3]. Note that general set addition is associative and monotone and injective in the second argument, but it is not commutative (not proven here).

We take the definition from [2].

```

definition add X  $\equiv$  transrec ( $\lambda$ addX Y.  $X \cup \text{image addX } Y$ )

```

```

bundle hotg-add-syntax begin notation add (infixl + 65) end
bundle no-hotg-add-syntax begin no-notation add (infixl + 65) end
unbundle hotg-add-syntax

```

```

lemma add-eq-bin-union-repl-add:  $X + Y = X \cup \{X + y \mid y \in Y\}$ 
  unfolding add-def by (simp add: transrec-eq)

```

The lift operation is from [2].

```

definition lift X  $\equiv$  image ((+) X)

```

```

lemma lift-eq-image-add: lift X = image ((+) X)
  unfolding lift-def by simp

```

```

lemma lift-eq-repl-add: lift X Y =  $\{X + y \mid y \in Y\}$ 
  using lift-eq-image-add by simp

```

```

lemma add-eq-bin-union-lift:  $X + Y = X \cup \text{lift } X \ Y$ 
  unfolding lift-eq-image-add by (subst add-eq-bin-union-repl-add) simp

```

```

corollary lift-subset-add: lift X Y  $\subseteq$  X + Y
  using add-eq-bin-union-lift by auto

```

```

Lemma 3.2 from [2] lemma lift-bin-union-eq-lift-bin-union-lift: lift X (A  $\cup$  B) = lift X A  $\cup$  lift X B
  by (auto simp: lift-eq-image-add)

```

```

lemma lift-union-eq-idx-union-lift: lift X ( $\bigcup Y$ ) = ( $\bigcup y \in Y. \text{lift } X \ y$ )
  by (auto simp: lift-eq-image-add)

```

```

lemma idx-union-add-eq-add-idx-union:
   $Y \neq \{\}$   $\implies$  ( $\bigcup y \in Y. X + f \ y$ ) = X + ( $\bigcup y \in Y. f \ y$ )
  by (simp add: lift-union-eq-idx-union-lift add-eq-bin-union-lift)

```

```

lemma lift-zero-eq-zero [simp]: lift X 0 = 0
  by (auto simp: lift-eq-image-add)

```

0 is the right identity of set addition.

lemma *add-zero-eq-self* [simp]: $X + 0 = X$
unfolding *add-eq-bin-union-lift* **by** *simp*

lemma *lift-one-eq-singleton-self* [simp]: $\text{lift } X \ 1 = \{X\}$
unfolding *lift-def* **by** *simp*

definition *succ* $X \equiv X + 1$

lemma *succ-eq-add-one*: $\text{succ } X = X + 1$
unfolding *succ-def* **by** *simp*

lemma *insert-self-eq-add-one*: $\text{insert } X \ X = X + 1$
by (*auto simp: add-eq-bin-union-lift succ-eq-add-one*)

lemma *succ-eq-insert*: $\text{succ } X = \text{insert } X \ X$
by (*simp add: succ-def insert-self-eq-add-one[of X]*)

lemma *lift-insert-eq-insert-add-lift*: $\text{lift } X \ (\text{insert } Y \ Z) = \text{insert } (X + Y) \ (\text{lift } X \ Z)$
unfolding *lift-def* **by** (*simp add: repl-insert-eq*)

lemma *add-insert-eq-insert-add*: $X + \text{insert } Y \ Z = \text{insert } (X + Y) \ (X + Z)$
by (*auto simp: lift-insert-eq-insert-add-lift add-eq-bin-union-lift*)

Proposition 3.3 from [2] 0 is the left identity of set addition.

lemma *zero-add-eq-self* [simp]: $0 + X = X$
proof (*induction X*)
case (*mem X*)
have $0 + X = \text{lift } 0 \ X$ **by** (*simp add: add-eq-bin-union-lift*)
also from mem have $\dots = X$ **by** (*simp add: lift-eq-image-add*)
finally show $?case$.
qed

corollary *lift-zero-eq-self* [simp]: $\text{lift } 0 \ X = X$
by (*simp add: lift-eq-image-add*)

corollary *add-eq-zeroE*:
assumes $X + Y = 0$
obtains $X = 0 \ Y = 0$
using *assms* **by** (*auto simp: add-eq-bin-union-lift*)

corollary *add-eq-zero-iff-and-eq-zero* [iff]: $X + Y = 0 \longleftrightarrow X = 0 \wedge Y = 0$
using *add-eq-zeroE* **by** *auto*

The next lemma demonstrates the associativity of set addition.

lemma *add-assoc*: $(X + Y) + Z = X + (Y + Z)$
proof (*induction Z*)

case (*mem* Z)
from *add-eq-bin-union-lift* **have** $(X + Y) + Z = (X + Y) \cup (\text{lift } (X + Y) Z)$
by *simp*
also from *lift-eq-repl-add* **have** $\dots = (X + Y) \cup \{(X + Y) + z \mid z \in Z\}$ **by**
simp
also from *add-eq-bin-union-lift* **have** $\dots = X \cup (\text{lift } X Y) \cup \{(X + Y) + z \mid z \in Z\}$ **by** *simp*
also from *mem* **have** $\dots = X \cup (\text{lift } X Y) \cup \{X + (Y + z) \mid z \in Z\}$ **by** *simp*
also have $\dots = X \cup \text{lift } X (Y + Z)$
proof–
from *add-eq-bin-union-lift* **have** $\text{lift } X (Y + Z) = \text{lift } X (Y \cup \text{lift } Y Z)$ **by**
simp
also from *lift-bin-union-eq-lift-bin-union-lift* **have** $\dots = (\text{lift } X Y) \cup \text{lift } X (\text{lift } Y Z)$ **by** *simp*
also from *lift-eq-repl-add* **have** $\dots = (\text{lift } X Y) \cup \{X + (Y + z) \mid z \in Z\}$ **by**
simp
finally have $\text{lift } X (Y + Z) = (\text{lift } X Y) \cup \{X + (Y + z) \mid z \in Z\}$.
then show *?thesis* **by** *auto*
qed
also from *add-eq-bin-union-lift* **have** $\dots = X + (Y + Z)$ **by** *simp*
finally show *?case* .
qed

lemma *lift-lift-eq-lift-add*: $\text{lift } X (\text{lift } Y Z) = \text{lift } (X + Y) Z$
by (*simp add: lift-eq-image-add add-assoc*)

lemma *add-succ-eq-succ-add*: $X + \text{succ } Y = \text{succ } (X + Y)$
by (*auto simp: succ-eq-add-one add-assoc*)

lemma *add-mem-lift-if-mem-right*:
assumes $X \in Y$
shows $Z + X \in \text{lift } Z Y$
using *assms* **by** (*auto simp: lift-eq-repl-add*)

corollary *add-mem-add-if-mem-right*:
assumes $X \in Y$
shows $Z + X \in Z + Y$
using *assms add-mem-lift-if-mem-right lift-subset-add* **by** *blast*

lemma *not-add-lt-left [iff]*: $\neg(X + Y < X)$
proof
assume $X + Y < X$
then show *False*
proof (*induction Y rule: mem-induction*)
case (*mem* Y)
then show *?case*
proof (*cases Y = {}*)
case *False*
then obtain y **where** $y \in Y$ **by** *blast*

```

    with add-mem-add-if-mem-right have  $X + y \in X + Y$  by auto
    with mem.premis have  $X + y < X$  by (auto intro: lt-if-lt-if-mem)
    with  $\langle y \in Y \rangle$  mem.IH show ?thesis by auto
  qed simp
qed
qed

```

```

lemma not-add-mem-left [iff]:  $X + Y \notin X$ 
  using subset-mem-trans-closure-self lt-iff-mem-trans-closure by auto

```

```

corollary add-subset-left-iff-right-eq-zero [iff]:  $X + Y \subseteq X \longleftrightarrow Y = 0$ 
  by (subst add-eq-bin-union-repl-add) auto

```

```

corollary lift-subset-left-iff-right-eq-zero [iff]:  $\text{lift } X \ Y \subseteq X \longleftrightarrow Y = 0$ 
  by (auto simp: lift-eq-repl-add)

```

```

lemma mem-trans-closure-bin-inter-lift-eq-empty [simp]:  $\text{mem-trans-closure } X \cap \text{lift } X \ Y = \{\}$ 
  by (auto simp: lift-eq-image-add simp flip: lt-iff-mem-trans-closure)

```

The next lemma shows that X and $\text{lift } X \ Y$ are disjoint, implying that $X + Y$ can be split into two disjoint parts.

```

lemma bin-inter-lift-self-eq-empty [simp]:  $X \cap \text{lift } X \ Y = \{\}$ 
  using mem-trans-closure-bin-inter-lift-eq-empty subset-mem-trans-closure-self by blast

```

```

corollary lift-bin-inter-self-eq-empty [simp]:  $\text{lift } X \ Y \cap X = \{\}$ 
  using bin-inter-lift-self-eq-empty by blast

```

```

lemma lift-eq-lift-if-bin-union-lift-eq-bin-union-lift:
  assumes  $X \cup \text{lift } X \ Y = X \cup \text{lift } X \ Z$ 
  shows  $\text{lift } X \ Y = \text{lift } X \ Z$ 
  using assms bin-inter-lift-self-eq-empty by blast

```

Proposition 3.4 from [2] lemma lift-injective-right: injective ($\text{lift } X$)

```

proof (rule injectiveI)
  fix  $Y \ Z$  assume  $\text{lift } X \ Y = \text{lift } X \ Z$ 
  then show  $Y = Z$ 
  proof (induction  $Y$  arbitrary:  $Z$  rule: mem-induction)
    case (mem  $Y$ )
    {
      fix  $U \ V \ u$  assume uvassms:  $U \in \{Y, Z\} \ V \in \{Y, Z\} \ U \neq V \ u \in U$ 
      with mem have  $X + u \in \text{lift } X \ V$  by (auto simp: lift-eq-repl-add)
      then obtain  $v$  where  $v \in V \ X + u = X + v$  using lift-eq-repl-add by auto
      then have  $X \cup \text{lift } X \ u = X \cup \text{lift } X \ v$  by (simp add: add-eq-bin-union-lift)
      with bin-inter-lift-self-eq-empty have  $\text{lift } X \ u = \text{lift } X \ v$  by blast
      with uvassms  $\langle v \in V \rangle$  mem.IH have  $u \in V$  by auto
    }
  then show ?case by blast

```

qed
qed

corollary *lift-eq-lift-if-eq-right*: $\text{lift } X \ Y = \text{lift } X \ Z \implies Y = Z$
using *lift-injective-right* **by** (*blast dest: injectiveD*)

corollary *lift-eq-lift-iff-eq-right* [*iff*]: $\text{lift } X \ Y = \text{lift } X \ Z \longleftrightarrow Y = Z$
using *lift-eq-lift-if-eq-right* **by** *auto*

lemma *add-injective-right*: *injective* $((+) \ X)$
using *lift-injective-right lift-eq-image-add* **by** *auto*

corollary *add-eq-add-if-eq-right*: $X + Y = X + Z \implies Y = Z$
using *add-injective-right* **by** (*blast dest: injectiveD*)

corollary *add-eq-add-iff-eq-right* [*iff*]: $X + Y = X + Z \longleftrightarrow Y = Z$
using *add-eq-add-if-eq-right* **by** *auto*

lemma *mem-if-add-mem-add-right*:
assumes $X + Y \in X + Z$
shows $Y \in Z$

proof –

have $X + Z = X \cup \text{lift } X \ Z$ **by** (*simp only: add-eq-bin-union-lift*)
with *assms* have $X + Y \in \text{lift } X \ Z$ **by** *auto*
also have $\dots = \{X + z \mid z \in Z\}$ **by** (*simp add: lift-eq-image-add*)
finally have $X + Y \in \{X + z \mid z \in Z\}$.
then show $Y \in Z$ **by** *blast*

qed

corollary *add-mem-add-iff-mem-right* [*iff*]: $X + Y \in X + Z \longleftrightarrow Y \in Z$
using *mem-if-add-mem-add-right add-mem-add-if-mem-right* **by** *blast*

Next we show some monotonicity lemmas for addition and lift.

lemma *mono-lift*: *mono* $(\text{lift } X)$
by (*auto simp: lift-eq-repl-add*)

lemma *subset-if-lift-subset-lift*: $\text{lift } X \ Y \subseteq \text{lift } X \ Z \implies Y \subseteq Z$
by (*auto simp: lift-eq-repl-add*)

corollary *lift-subset-lift-iff-subset*: $\text{lift } X \ Y \subseteq \text{lift } X \ Z \longleftrightarrow Y \subseteq Z$
using *subset-if-lift-subset-lift mono-lift[of X]* **by** (*auto del: subsetI*)

lemma *mono-add*: *mono* $((+) \ X)$

proof (*rule monoI[of (+) X, simplified]*)

fix $Y \ Z$ assume $Y \subseteq Z$

then have $\text{lift } X \ Y \subseteq \text{lift } X \ Z$ **by** (*simp only: lift-subset-lift-iff-subset*)

then show $X + Y \subseteq X + Z$ **by** (*auto simp: add-eq-bin-union-lift*)

qed

lemma *subset-if-add-subset-add*:

assumes $X + Y \subseteq X + Z$

shows $Y \subseteq Z$

proof –

have $X + Z = X \cup \text{lift } X \ Z$ **by** (*simp only: add-eq-bin-union-lift*)

with *assms* **have** $\text{lift } X \ Y \subseteq X \cup \text{lift } X \ Z$ **by** (*auto simp: add-eq-bin-union-lift*)

moreover **have** $\text{lift } X \ Y \cap X = \{\}$ **by** (*fact lift-bin-inter-self-eq-empty*)

ultimately **have** $\text{lift } X \ Y \subseteq \text{lift } X \ Z$ **by** *blast*

with *lift-subset-lift-iff-subset* **show** *?thesis* **by** *simp*

qed

corollary *add-subset-add-iff-subset [iff]*: $X + Y \subseteq X + Z \longleftrightarrow Y \subseteq Z$

using *subset-if-add-subset-add mono-add[of X]* **by** (*auto del: subsetI*)

The transitive closure of addition can be split into two smaller closures depending on its arguments.

lemma *mem-trans-closure-add-eq-mem-trans-closure-bin-union*:

$\text{mem-trans-closure } (X + Y) = \text{mem-trans-closure } X \cup \text{lift } X \ (\text{mem-trans-closure } Y)$

proof (*induction Y*)

case (*mem Y*)

have $\text{mem-trans-closure } (X + Y) = (X + Y) \cup (\bigcup z \in X + Y. \text{mem-trans-closure } z)$

by (*subst mem-trans-closure-eq-bin-union-idx-union*) *simp*

also **have** $\dots = \text{mem-trans-closure } X \cup \text{lift } X \ Y \cup (\bigcup y \in Y. \text{mem-trans-closure } (X + y))$

(*is - = ?unions \cup -*)

by (*auto simp: lift-eq-repl-add idx-union-bin-union-dom-eq-bin-union-idx-union add-eq-bin-union-lift[of X Y] mem-trans-closure-eq-bin-union-idx-union[of X]*)

also **have** $\dots = ?unions \cup (\bigcup y \in Y. \text{mem-trans-closure } X \cup \text{lift } X \ (\text{mem-trans-closure } y))$

using *mem.IH* **by** *simp*

also **have** $\dots = ?unions \cup (\bigcup y \in Y. \text{lift } X \ (\text{mem-trans-closure } y))$ **by** *auto*

also **have** $\dots = \text{mem-trans-closure } X \cup \text{lift } X \ (Y \cup (\bigcup y \in Y. \text{mem-trans-closure } y))$

by (*simp add: lift-bin-union-eq-lift-bin-union-lift*

lift-union-eq-idx-union-lift bin-union-assoc mem-trans-closure-eq-bin-union-idx-union[of X])

also **have** $\dots = \text{mem-trans-closure } X \cup \text{lift } X \ (\text{mem-trans-closure } Y)$

by (*simp flip: mem-trans-closure-eq-bin-union-idx-union*)

finally **show** *?case* .

qed

corollary *lt-add-if-lt-left*:

assumes $X < Y$

shows $X < Y + Z$

using *assms mem-trans-closure-add-eq-mem-trans-closure-bin-union*

by (*auto simp: lt-iff-mem-trans-closure*)

corollary *add-lt-add-if-lt-right*:

assumes $X < Y$

shows $Z + X < Z + Y$

using *assms mem-trans-closure-add-eq-mem-trans-closure-bin-union*

by (*auto simp: lt-iff-mem-trans-closure lift-eq-image-add*)

corollary *lt-add-if-eq-add-if-lt*:

assumes $x < X$

and $Y = Z + x$

shows $Y < Z + X$

using *assms add-lt-add-if-lt-right* **by** *simp*

corollary *lt-addE*:

assumes $X < Y + Z$

obtains (*lt-left*) $X < Y$ | (*lt-eq*) z **where** $z < Z$ $X = Y + z$

using *assms mem-trans-closure-add-eq-mem-trans-closure-bin-union*

by (*auto simp: lt-iff-mem-trans-closure lift-eq-image-add*)

corollary *lt-add-iff-lt-or-lt-eq*: $X < Y + Z \longleftrightarrow X < Y \vee (\exists z. z < Z \wedge X = Y + z)$

by (*blast intro: lt-add-if-lt-left add-lt-add-if-lt-right elim: lt-addE*)

lemma *lt-add-self-if-ne-zero* [*simp*]:

assumes $Y \neq 0$

shows $X < X + Y$

using *assms* **by** (*intro lt-add-if-eq-add-if-lt*) *auto*

corollary *le-self-add* [*iff*]: $X \leq X + Y$

using *lt-add-self-if-ne-zero le-iff-lt-or-eq* **by** (*cases* $Y = 0$) *auto*

end

theory *Mem-Transitive-Closed*

imports

Mem-Transitive-Closed-Base

SAddition

begin

lemma *mem-trans-closed-succI* [*intro*]:

assumes *mem-trans-closed* X

shows *mem-trans-closed* (*succ* X)

unfolding *succ-def* **using** *assms*

by (*auto simp flip: insert-self-eq-add-one*)

lemma *mem-trans-closed-unionI*:

assumes $\bigwedge x. x \in X \implies \text{mem-trans-closed } x$

shows *mem-trans-closed* ($\bigcup X$)

using *assms* **by** (*intro mem-trans-closedI*) *auto*

```

lemma mem-trans-closed-interI:
  assumes  $\bigwedge x. x \in X \implies \text{mem-trans-closed } x$ 
  shows mem-trans-closed  $(\bigcap X)$ 
  using assms by (intro mem-trans-closedI) auto

lemma mem-trans-closed-bin-unionI:
  assumes mem-trans-closed  $X$ 
  and mem-trans-closed  $Y$ 
  shows mem-trans-closed  $(X \cup Y)$ 
  using assms by blast

lemma mem-trans-closed-bin-interI:
  assumes mem-trans-closed  $X$ 
  and mem-trans-closed  $Y$ 
  shows mem-trans-closed  $(X \cap Y)$ 
  using assms by blast

lemma mem-trans-closed-powersetI: mem-trans-closed  $X \implies \text{mem-trans-closed}$ 
  (powerset  $X$ )
  by auto

lemma union-succ-eq-self-if-mem-trans-closed [simp]: mem-trans-closed  $X \implies \bigcup (\text{succ}$ 
   $X) = X$ 
  by (auto simp flip: insert-self-eq-add-one simp: succ-eq-add-one)

end

```

20 Ordinals

```

theory Ordinals
  imports
    Mem-Transitive-Closed
begin

unbundle no-HOL-groups-syntax

```

Summary Translation of ordinals from https://www.isa-afp.org/entries/ZFC_in_HOL.html. We give the definition of ordinals and limit ordinals. In addition, two ordinal inductions are proven.

And we use the Von Neumann encoding of natural numbers. The Von Neumann numbers are defined inductively. The Von Neumann number 0 is defined to be the empty set, and there are no smaller Von Neumann numbers. The Von Neumann number N is then the set of all Von Neumann numbers less than N . Further details can be found in <https://planetmath>.

[org/vonneumanninteger](https://foss.heptapod.net/isa-afp/afp-devel/-/blob/06458dfa40c7b4aaeb855a37ae77993cb4c8c18/thys/ZFC_in_HOL/ZFC_in_HOL.thy#L601).

Ordinals We follow the definition from [3], https://foss.heptapod.net/isa-afp/afp-devel/-/blob/06458dfa40c7b4aaeb855a37ae77993cb4c8c18/thys/ZFC_in_HOL/ZFC_in_HOL.thy#L601.

definition *ordinal* $X \equiv \text{mem-trans-closed } X \wedge (\forall x \in X. \text{mem-trans-closed } x)$

lemma *ordinal-mem-trans-closedE*:

assumes *ordinal* X

obtains $\text{mem-trans-closed } X \wedge x. x \in X \implies \text{mem-trans-closed } x$

using *assms* **unfolding** *ordinal-def* **by** *auto*

lemma *ordinal-if-mem-trans-closedI*:

assumes $\text{mem-trans-closed } X$

and $\wedge x. x \in X \implies \text{mem-trans-closed } x$

shows *ordinal* X

using *assms* **unfolding** *ordinal-def* **by** *auto*

context

notes *ordinal-mem-trans-closedE*[*elim!*] *ordinal-if-mem-trans-closedI*[*intro!*]

begin

lemma *ordinal-zero* [*iff*]: *ordinal* 0 **by** *auto*

lemma *ordinal-one* [*iff*]: *ordinal* 1 **by** *auto*

lemma *ordinal-succI* [*intro*]:

assumes *ordinal* x

shows *ordinal* (*succ* x)

using *assms* **by** (*auto simp flip: insert-self-eq-add-one simp: succ-eq-add-one*)

lemma *ordinal-unionI*:

assumes $\wedge x. x \in X \implies \text{ordinal } x$

shows *ordinal* ($\bigcup X$)

using *assms* **by** *blast*

lemma *ordinal-interI*:

assumes $\wedge x. x \in X \implies \text{ordinal } x$

shows *ordinal* ($\bigcap X$)

using *assms* **by** *blast*

lemma *ordinal-bin-unionI*:

assumes *ordinal* X

and *ordinal* Y

shows *ordinal* ($X \cup Y$)

using *assms* **by** *blast*

lemma *ordinal-bin-interI*:

assumes *ordinal* X
and *ordinal* Y
shows *ordinal* $(X \cap Y)$
using *assms* **by** *blast*

lemma *subset-if-mem-if-ordinal*: $\text{ordinal } X \implies Y \in X \implies Y \subseteq X$ **by** *auto*

lemma *mem-trans-if-ordinal*: $\llbracket \text{ordinal } X; Y \in Z; Z \in X \rrbracket \implies Y \in X$ **by** *auto*

lemma *ordinal-if-mem-if-ordinal*: $\llbracket \text{ordinal } X; Y \in X \rrbracket \implies \text{ordinal } Y$
by *blast*

lemma *union-succ-eq-self-if-ordinal* [*simp*]: $\text{ordinal } \beta \implies \bigcup (\text{succ } \beta) = \beta$ **by** *auto*

Membership ordinal induction:

lemma *ordinal-mem-induct* [*consumes 1, case-names step*]:
assumes *ordinal* X
and $\bigwedge x. x \in X \implies P\ x \implies P\ X$
shows $P\ X$
using *assms* *ordinal-if-mem-if-ordinal*
by (*induction* X *rule: mem-induction*) *auto*

Limit Ordinals We follow the definition from [3], https://foss.heptapod.net/isa-afp/afp-devel/-/blob/06458dfa40c7b4aaaeb855a37ae77993cb4c8c18/thys/ZFC_in_HOL/ZFC_in_HOL.thy#L939. A limit ordinal X is an ordinal number greater than 0 that is not a successor ordinal. Further details can be found in https://en.wikipedia.org/wiki/Limit_ordinal.

definition *limit* $X \equiv \text{ordinal } X \wedge 0 \in X \wedge (\forall x \in X. \text{succ } x \in X)$

lemma *limitI*:
assumes *ordinal* X
and $0 \in X$
and $\bigwedge x. x \in X \implies \text{succ } x \in X$
shows *limit* X
using *assms* **unfolding** *limit-def* **by** *auto*

lemma *limitE*:
assumes *limit* X
obtains *ordinal* X $0 \in X$ $\bigwedge x. x \in X \implies \text{succ } x \in X$
using *assms* **unfolding** *limit-def* **by** *auto*

For the second induction theorem, some lemmas are left unproven.

lemma *Limit-eq-Sup-self*: $\text{limit } X \implies \bigcup X = X$
sorry

lemma *ordinal-cases* [*cases type: set, case-names 0 succ limit*]:
assumes *ordinal* k
obtains $k = 0 \mid l$ **where** *ordinal* l $\text{succ } l = k \mid \text{limit } k$

```

sorry

lemma elts-succ [simp]:  $\{xx \mid xx \in (succ\ x)\} = insert\ x\ \{xx \mid xx \in x\}$ 
by (simp add: succ-eq-insert)

lemma image-ident: image id Y = Y
by auto

Standard ordinal induction:

lemma ordinal-induct [consumes 1, case-names zero succ limit, induct type: set]:
  assumes a: ordinal X
  and P: P 0  $\wedge$  X.  $\llbracket ordinal\ X; P\ X \rrbracket \implies P (succ\ X)$ 
   $\wedge$  X.  $\llbracket limit\ X; \wedge x. x \in X \implies P\ x \rrbracket \implies P (\bigcup X)$ 
  shows P X
using a
proof (induction X rule: ordinal-mem-induct)
  case (step X)
  then show ?case
  proof (cases rule: ordinal-cases)
    case 0
    with P(1) show ?thesis by simp
  next
    case (succ l)
    from succ step succ-eq-insert have P (succ l) by (intro P(2)) auto
    with succ show ?thesis by simp
  next
    case limit

    Missing proof see https://foss.heptapod.net/isa-afp/afp-devel/-/blob/06458dfa40c7b4aaae8b855a37ae77993cb4c8c18/thys/ZFC\_in\_HOL/ZFC\_in\_HOL.thy#L991.

    then show ?thesis sorry
  qed
qed

end

end

```

21 Generalised Multiplication

```

theory SMultiplication
imports
  SAddition
  Ordinals
begin

```

Summary Translation of generalised set multiplication for sets from [2] and [3]. Note that general set multiplication is associative, but it is not commutative (not proven here).

We take the definition from [2].

definition $mul\ X \equiv transrec\ (\lambda mulX\ Y. \bigcup (image\ (\lambda y. lift\ (mulX\ y)\ X)\ Y))$

bundle *hotg-mul-syntax* **begin notation** mul (**infixl** * 70) **end**

bundle *no-hotg-mul-syntax* **begin no-notation** mul (**infixl** * 70) **end**

unbundle *hotg-mul-syntax*

lemma *mul-eq-idx-union-lift-mul*: $X * Y = (\bigcup y \in Y. lift\ (X * y)\ X)$
by (*simp add: mul-def transrec-eq*)

corollary *mul-eq-idx-union-repl-mul-add*: $X * Y = (\bigcup y \in Y. \{X * y + x \mid x \in X\})$
using *mul-eq-idx-union-lift-mul*[*of X Y*] *lift-eq-repl-add* **by** *simp*

Lemma 4.2 from [2] **lemma** *mul-zero-eq-zero* [*simp*]: $X * 0 = 0$
by (*subst mul-eq-idx-union-lift-mul*) *simp*

1 is the right identity of set multiplication.

lemma *mul-one-eq-self* [*simp*]: $X * 1 = X$
by (*auto simp: mul-eq-idx-union-lift-mul*[**where** ?Y=1])

lemma *mul-singleton-one-eq-lift-self*: $X * \{1\} = lift\ X\ X$
by (*auto simp: mul-eq-idx-union-lift-mul*[**where** ?Y={1}])

lemma *mul-two-eq-add-self*: $X * 2 = X + X$

proof –

have $X * 2 = (\bigcup y \in 2. lift\ (X * y)\ X)$ **by** (*simp only: mul-eq-idx-union-lift-mul*[**where** ?Y=2])

also have $\dots = lift\ (X * 1)\ X \cup lift\ (X * 0)\ X$

using *idx-union-insert-dom-eq-bin-union-idx-union* **by** *auto*

also have $\dots = X + X$ **by** (*auto simp: add-eq-bin-union-lift*)

finally show *?thesis* .

qed

lemma *mul-bin-union-eq-bin-union-mul*: $X * (Y \cup Z) = (X * Y) \cup (X * Z)$

proof –

have $X * (Y \cup Z) = (\bigcup y \in (Y \cup Z). lift\ (X * y)\ X)$ **by** (*simp flip: mul-eq-idx-union-lift-mul*)

also have $\dots = (\bigcup y \in Y. lift\ (X * y)\ X) \cup (\bigcup z \in Z. lift\ (X * z)\ X)$

using *idx-union-bin-union-dom-eq-bin-union-idx-union* **by** *simp*

also have $\dots = (X * Y) \cup (X * Z)$ **by** (*auto simp flip: mul-eq-idx-union-lift-mul*)

finally show *?thesis* .

qed

lemma *mul-insert-eq-mul-bin-union-lift-mul*: $X * (insert\ Z\ Y) = (X * Y) \cup lift\ (X * Z)\ X$

proof –
 have $X * (\text{insert } Z \ Y) = X * (Y \cup \{Z\})$ **by** *auto*
 also have $\dots = (X * Y) \cup (X * \{Z\})$ **by** (*simp only: mul-bin-union-eq-bin-union-mul*)
 also have $\dots = (X * Y) \cup \text{lift } (X * Z) \ X$ **by** (*auto simp: mul-eq-idx-union-lift-mul* [**where**
 $?Y = \{Z\}$])
 finally **show** *?thesis* .
qed

lemma *mul-succ-eq-mul-add* [*simp*]: $X * \text{succ } Y = X * Y + X$
proof –
 have $X * \text{succ } Y = X * (\text{insert } Y \ Y)$
by (*simp only: insert-self-eq-add-one* [**where** $?X = Y$] *succ-eq-add-one*)
 also have $\dots = (X * Y) \cup \text{lift } (X * Y) \ X$ **by** (*simp only: mul-insert-eq-mul-bin-union-lift-mul*)
 also have $\dots = (X * Y) + X$ **by** (*simp add: add-eq-bin-union-lift*)
 finally **show** *?thesis* .
qed

lemma *subset-self-mul-if-zero-mem*:
 assumes $0 \in X$
 shows $Y \subseteq Y * X$
 using *assms* **by** (*subst mul-eq-idx-union-lift-mul*) *fastforce*

Proposition 4.3 from [2] **lemma** *zero-mul-eq-zero* [*simp*]: $0 * X = 0$
by (*induction X, subst mul-eq-idx-union-lift-mul*) *auto*

1 is also the left identity of set multiplication.

lemma *one-mul-eq* [*simp*]: $1 * X = X$
by (*induction X, subst mul-eq-idx-union-lift-mul*) *auto*

lemma *mul-union-eq-idx-union-mul*: $X * \bigcup Y = (\bigcup y \in Y. X * y)$
proof –
 have $X * \bigcup Y = (\bigcup y \in Y. \bigcup z \in y. \text{lift } (X * z) \ X)$ **by** (*subst mul-eq-idx-union-lift-mul*)
simp
 also have $\dots = (\bigcup y \in Y. X * y)$ **by** (*simp flip: mul-eq-idx-union-lift-mul*)
 finally **show** *?thesis* .
qed

lemma *mul-lift-eq-lift-mul-mul*: $X * (\text{lift } Y \ Z) = \text{lift } (X * Y) \ (X * Z)$
proof (*induction Z rule: mem-induction*)
 case (*mem Z*)
 have $X * (\text{lift } Y \ Z) = (\bigcup z \in \text{lift } Y \ Z. \text{lift } (X * z) \ X)$ **by** (*simp flip: mul-eq-idx-union-lift-mul*)
 also have $\dots = (\bigcup z \in Z. \text{lift } (X * (Y + z)) \ X)$ **by** (*simp add: lift-eq-image-add*)
 also **from** *mem* **have** $\dots = \text{lift } (X * Y) \ (\bigcup z \in Z. \text{lift } (X * z) \ X)$
by (*simp add: add-eq-bin-union-lift lift-union-eq-idx-union-lift lift-lift-eq-lift-add*
mul-bin-union-eq-bin-union-mul)
 also have $\dots = \text{lift } (X * Y) \ (X * Z)$ **by** (*simp flip: mul-eq-idx-union-lift-mul*)
 finally **show** *?case* .
qed

lemma *mul-add-eq-mul-add-mul*: $X * (Y + Z) = X * Y + X * Z$
by (*simp only: add-eq-bin-union-lift mul-bin-union-eq-bin-union-mul mul-lift-eq-lift-mul-mul*)

The lemma demonstrates the associativity of set multiplication.

lemma *mul-assoc*: $(X * Y) * Z = X * (Y * Z)$
proof (*induction Z rule: mem-induction*)
case (*mem Z*)
have $(X * Y) * Z = (\bigcup z \in Z. \text{lift } ((X * Y) * z) (X * Y))$
by (*subst mul-eq-idx-union-lift-mul*) *simp*
also from *mem* **have** $\dots = (\bigcup z \in Z. X * \text{lift}(Y * z) Y)$ **by** (*simp add: mul-lift-eq-lift-mul-mul*)
also have $\dots = X * (\bigcup z \in Z. \text{lift}(Y * z) Y)$ **by** (*simp add: mul-union-eq-idx-union-mul*)
also have $\dots = X * (Y * Z)$ **by** (*simp flip: mul-eq-idx-union-lift-mul*)
finally show *?case* .
qed

Lemma 4.5 from [2] **lemma** *le-mul-if-ne-zero*:
assumes $Y \neq 0$
shows $X \leq X * Y$
proof (*cases X = 0*)
case *False*
from *assms* **show** *?thesis*
proof (*induction Y rule: mem-induction*)
case (*mem Y*)
then show *?case*
proof (*cases Y = 1*)
case *False*
with *mem* **obtain** *P* **where** $P: P \in Y \ P \neq 0$ **by** *blast*
from $\langle X \neq 0 \rangle$ **obtain** *R* **where** $R: R \in X$ **by** *auto*
from *mem.IH* **have** $X \leq X * P$ **using** *P* **by** *auto*
also have $\dots \leq X * P + R$ **by** *simp*
also have $\dots \leq X * Y$
proof –
from *R* **have** $X * P + R \in \text{lift } (X * P) X$ **by** (*auto simp: lift-eq-image-add*)
also have $\dots \subseteq X * Y$ **using** *P* **by** (*auto simp: mul-eq-idx-union-lift-mul*) **where**
 $?Y=Y$)
finally have $X * P + R \in X * Y$.
then show *?thesis* **by** (*intro le-if-lt lt-if-mem*)
qed
finally show *?thesis* .
qed *simp*
qed
qed *simp*

Lemma 4.6 from [2] **lemma** *lt-mul-if-ne-zero*: **assumes** $X \neq 0 \ Y \neq 0 \ Y \neq 1$
shows $X < X * Y$
sorry


```

lemma zero-if-multi-eq-multi-add: assumes  $A * X = A * Y + B$   $B < A$ 
shows  $B = 0$ 
proof (cases  $A = 0 \vee X = 0$ )
  case True
    with assms show ?thesis
  proof (cases  $A = 0$ )
    case False
      then have  $A * Y + B = 0$  using  $\langle A = 0 \vee X = 0 \rangle$  assms by auto
      then show ?thesis
        by (auto simp: add-eq-zero-iff-and-eq-zero[of  $A * Y B$ ])
    qed auto
  next
    case False
      then have  $A \neq 0$   $X \neq 0$  by auto
      then show ?thesis
        proof (cases  $Y = 0$ )
          case True
            then show ?thesis sorry
          next
            case False
              then show ?thesis sorry
            qed
          qed
        qed

```

Lemma 4.7 from [2] **lemma** subset-if-mul-add-subset-mul-add: **assumes** $R < A$ $S < A$ $A * X + R \subseteq A * Y + S$
shows $X \subseteq Y$
sorry

lemma eq-if-mul-add-eq-mul-add: **assumes** $R < A$ $S < A$ $A * X + R = A * Y + S$
shows $X = Y$ $R = S$
sorry

lemma bin-inter-lift-mul-mem-trans-closure-lift-mul-mem-trans-closure-eq-zero:
assumes $X \neq Y$
shows $\text{lift } (A * X) (\text{mem-trans-closure } A) \cap \text{lift } (A * Y) (\text{mem-trans-closure } A) = 0$
 (is ?s1 \cap ?s2 = 0)
proof (rule eqI)
fix x **assume** *asm*: $x \in ?s1 \cap ?s2$
then obtain r **where** R : $x = A * X + r$ $r \in \text{mem-trans-closure } A$
using lift-eq-repl-add **by** auto
from *asm* **obtain** rr **where** RR : $x = A * Y + rr$ $rr \in \text{mem-trans-closure } A$
using lift-eq-repl-add **by** auto
with R **have** $A * X + r = A * Y + rr$ $r < A$ $rr < A$ **by** (auto simp: lt-iff-mem-trans-closure)
then have $X = Y$ $r = rr$ **using** eq-if-mul-add-eq-mul-add[*of* $r - rr X$ -] **by** auto
then show $x \in 0$ **by** (simp add: *assms*)

qed *simp*

end

22 Pairs (Σ -types)

theory *Pairs*

imports

Foundation

begin

definition *pair* :: $\langle \text{set} \Rightarrow \text{set} \Rightarrow \text{set} \rangle$

where *pair* *a b* $\equiv \{\{a\}, \{a, b\}\}$

definition *fst* :: $\langle \text{set} \Rightarrow \text{set} \rangle$

where *fst* *p* $\equiv \text{THE } a. \exists b. p = \text{pair } a \ b$

definition *snd* :: $\langle \text{set} \Rightarrow \text{set} \rangle$

where *snd* *p* $\equiv \text{THE } b. \exists a. p = \text{pair } a \ b$

bundle *hotg-tuple-syntax*

begin

syntax *-tuple* :: $\langle \text{args} \Rightarrow \text{set} \rangle (\langle - \rangle)$

end

bundle *no-hotg-tuple-syntax*

begin

no-syntax *-tuple* :: $\langle \text{args} \Rightarrow \text{set} \rangle (\langle - \rangle)$

end

unbundle *hotg-tuple-syntax*

translations

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

$\langle x, y \rangle \equiv \text{CONST pair } x \ y$

lemma *pair-eq-iff* [*iff*]: $\langle a, b \rangle = \langle c, d \rangle \longleftrightarrow a = c \wedge b = d$

unfolding *pair-def* **by** (*auto dest: iffD1[OF upair-eq-iff]*)

lemma *eq-if-pair-eq-left*: $\langle a, b \rangle = \langle c, d \rangle \implies a = c$ **by** *simp*

lemma *eq-if-pair-eq-right*: $\langle a, b \rangle = \langle c, d \rangle \implies b = d$ **by** *simp*

lemma *fst-pair-eq* [*simp*]: $\text{fst } \langle a, b \rangle = a$

by (*simp add: fst-def*)

lemma *snd-pair-eq* [*simp*]: $\text{snd } \langle a, b \rangle = b$

```

by (simp add: snd-def)

lemma pair-ne-empty [iff]:  $\langle a, b \rangle \neq \{\}$ 
  unfolding pair-def by blast

lemma fst-snd-eq-if-eq-pair [simp]:  $p = \langle a, b \rangle \implies \langle \text{fst } p, \text{snd } p \rangle = p$ 
  by simp

lemma pair-ne-fst [iff]:  $\langle a, b \rangle \neq a$ 
  unfolding pair-def using not-mem-if-mem
  by (intro ne-if-ex-mem-not-mem, intro exI[where x={a}]) auto

lemma pair-ne-snd [iff]:  $\langle a, b \rangle \neq b$ 
  unfolding pair-def using not-mem-if-mem
  by (intro ne-if-ex-mem-not-mem, intro exI[where x={a, b}]) auto

lemma pair-not-mem-fst [iff]:  $\langle a, b \rangle \notin a$ 
  unfolding pair-def using not-mem-if-mem-if-mem by auto

lemma pair-not-mem-snd [iff]:  $\langle a, b \rangle \notin b$ 
  unfolding pair-def by (auto dest: not-mem-if-mem-if-mem)

```

22.1 Set-Theoretic Dependent Pair Type

```

definition dep-pairs ::  $\langle \text{set} \Rightarrow (\text{set} \Rightarrow \text{set}) \Rightarrow \text{set} \rangle$ 
  where dep-pairs A B  $\equiv \bigcup x \in A. \bigcup y \in B x. \{\langle x, y \rangle\}$ 

bundle hotg-dependent-pairs-syntax
begin
syntax
  -dep-pairs ::  $\langle [\text{pttrn}, \text{set}, \text{set} \Rightarrow \text{set}] \Rightarrow \text{set} \rangle$  ( $\sum - \in \text{.} / - [0, 0, 100] 51$ )
end
bundle no-hotg-dependent-pairs-syntax
begin
no-syntax
  -dep-pairs ::  $\langle [\text{pttrn}, \text{set}, \text{set} \Rightarrow \text{set}] \Rightarrow \text{set} \rangle$  ( $\sum - \in \text{.} / - [0, 0, 100] 51$ )
end
unbundle hotg-dependent-pairs-syntax

translations
   $\sum x \in A. B \Rightarrow \text{CONST } \text{dep-pairs } A (\lambda x. B)$ 

abbreviation pairs ::  $\langle \text{set} \Rightarrow \text{set} \Rightarrow \text{set} \rangle$ 
  where pairs A B  $\equiv \sum - \in A. B$ 

bundle hotg-pairs-syntax begin notation pairs (infixl  $\times 80$ ) end
bundle no-hotg-pairs-syntax begin no-notation pairs (infixl  $\times 80$ ) end

unbundle hotg-pairs-syntax

```

lemma *mem-dep-pairs-iff* [iff]: $\langle a, b \rangle \in (\sum x \in A. B x) \longleftrightarrow a \in A \wedge b \in B a$
unfolding *dep-pairs-def* **by** *blast*

lemma *mem-if-mem-dep-pairs-fst*: $\langle a, b \rangle \in (\sum x \in A. B x) \implies a \in A$ **by** *simp*
lemma *mem-if-mem-dep-pairs-snd*: $\langle a, b \rangle \in (\sum x \in A. B x) \implies b \in B a$ **by** *simp*

lemma *mem-dep-pairsE* [elim!]:
assumes $p \in \sum x \in A. B x$
obtains $x y$ **where** $x \in A \ y \in B x \ p = \langle x, y \rangle$
using *assms* **unfolding** *dep-pairs-def* **by** *blast*

lemma *dep-pairs-cong* [cong]:
 $\llbracket A = A'; \bigwedge x. x \in A' \implies B x = B' x \rrbracket \implies (\sum x \in A. B x) = (\sum x \in A'. B' x)$
unfolding *dep-pairs-def* **by** *auto*

lemma *fst-mem-if-mem-dep-pairs*: $p \in \sum x \in A. B x \implies \text{fst } p \in A$
by *auto*

lemma *snd-mem-if-mem-dep-pairs*: $p \in \sum x \in A. B x \implies \text{snd } p \in B (\text{fst } p)$
by *auto*

lemma *fst-snd-eq-pair-if-mem-dep-pairs* [simp]:
 $p \in \sum x \in P. B x \implies \langle \text{fst } p, \text{snd } p \rangle = p$
by *auto*

lemma *dep-pairs-empty-dom-eq-empty* [simp]: $\sum x \in \{\}. B x = \{\}$
by *auto*

lemma *dep-pairs-empty-eq-empty* [simp]: $\sum x \in A. \{\} = \{\}$
by *auto*

lemma *pairs-empty-iff* [iff]: $A \times B = \{\} \longleftrightarrow A = \{\} \vee B = \{\}$
by (*auto intro!*: *eqI*)

lemma *pairs-singleton-eq* [simp]: $\{a\} \times \{b\} = \{\langle a, b \rangle\}$
by (*rule eqI*) *auto*

lemma *dep-pairs-subset-pairs*: $\sum x \in A. B x \subseteq A \times (\bigcup x \in A. B x)$
by *auto*

Splitting quantifiers:

lemma *bex-dep-pairs-iff-bex-bex* [iff]:
 $(\exists z \in \sum x \in A. B x. P z) \longleftrightarrow (\exists x \in A. \exists y \in B x. P \langle x, y \rangle)$
by *blast*

lemma *ball-dep-pairs-iff-ball-ball* [iff]:
 $(\forall z \in \sum x \in A. B x. P z) \longleftrightarrow (\forall x \in A. \forall y \in B x. P \langle x, y \rangle)$
by *blast*

22.2 Monotonicity

lemma *mono-dep-pairs*:

assumes $A \subseteq A'$

and $\bigwedge x. x \in A \implies B\ x \subseteq B'\ x$

shows $(\sum x \in A. B\ x) \subseteq (\sum x \in A'. B'\ x)$

using *assms* **by** *auto*

lemma *mono-dep-pairs-dom*:

assumes $A \subseteq A'$

shows $(\sum x \in A. B\ x) \subseteq (\sum x \in A'. B\ x)$

using *assms* **by** (*intro mono-dep-pairs*) *auto*

lemma *mono-dep-pairs-rng*:

assumes $\bigwedge x. x \in A \implies B\ x \subseteq B'\ x$

shows $(\sum x \in A. B\ x) \subseteq (\sum x \in A. (B'\ x))$

using *assms* **by** (*intro mono-dep-pairs*) *auto*

lemma *mono-pairs-dom*: *mono* $(\lambda A. A \times B)$

by (*intro monoI*) *auto*

lemma *mono-pairs-rng*: *mono* $(\lambda B. A \times B)$

by (*intro monoI*) *auto*

22.3 Functions on Dependent Pairs

definition *uncurry* $f\ p \equiv f\ (fst\ p)\ (snd\ p)$

bundle *hotg-uncurry-syntax*

begin

syntax *-uncurry-args* $:: args \Rightarrow pttrn\ (\langle \cdot \rangle)$

end

bundle *no-hotg-uncurry-syntax*

begin

no-syntax *-uncurry-args* $:: args \Rightarrow pttrn\ (\langle \cdot \rangle)$

end

unbundle *hotg-uncurry-syntax*

translations

$\lambda \langle x, y, zs \rangle. b \equiv CONST\ uncurry\ (\lambda x\ \langle y, zs \rangle. b)$

$\lambda \langle x, y \rangle. b \equiv CONST\ uncurry\ (\lambda x\ y. b)$

lemma *uncurry [simp]*: *uncurry* $f\ \langle a, b \rangle = f\ a\ b$

unfolding *uncurry-def* **by** *simp*

definition *swap* $p = \langle snd\ p, fst\ p \rangle$

lemma *swap-pair-eq [simp]*: *swap* $\langle x, y \rangle = \langle y, x \rangle$ **unfolding** *swap-def* **by** *simp*

end

23 Coproduct (\coprod -types)

Aka binary disjoint union.

theory *Coproduct*
imports *Pairs*
begin

definition *inl* $a = \langle \{\}, a \rangle$

definition *inr* $b = \langle \{\{\}\}, b \rangle$

definition *coprod* $A \ B \equiv \{inl\ a \mid a \in A\} \cup \{inr\ b \mid b \in B\}$

bundle *hotg-coprod-syntax* **begin notation** *coprod* (**infixl** \coprod 70) **end**

bundle *no-hotg-coprod-syntax* **begin no-notation** *coprod* (**infixl** \coprod 70) **end**

unbundle *hotg-coprod-syntax*

lemma *mem-coprod-iff* [*iff*]:

$x \in A \coprod B \longleftrightarrow (\exists a \in A. x = inl\ a) \vee (\exists b \in B. x = inr\ b)$

unfolding *coprod-def inl-def inr-def* **by** *auto*

lemma *mem-coprodE*:

assumes $x \in A \coprod B$

obtains (*inl*) a **where** $a \in A$ $x = inl\ a \mid$ (*inr*) b **where** $b \in B$ $x = inr\ b$

using *assms* **by** *blast*

lemma

inl-inj-iff [*iff*]: $inl\ x = inl\ y \longleftrightarrow x = y$ **and**

inr-inj-iff [*iff*]: $inr\ x = inr\ y \longleftrightarrow x = y$ **and**

inl-ne-inr [*iff*]: $inl\ x \neq inr\ y$ **and**

inr-ne-inl [*iff*]: $inr\ x \neq inl\ y$

unfolding *inl-def inr-def* **by** *auto*

lemma *inl-mem-coprod-iff* [*iff*]: $inl\ a \in A \coprod B \longleftrightarrow a \in A$

unfolding *coprod-def* **by** *auto*

lemma *inr-mem-coprod-iff* [*iff*]: $inr\ b \in A \coprod B \longleftrightarrow b \in B$

unfolding *coprod-def* **by** *auto*

definition *coprod-rec* $l\ r\ x = (if\ fst\ x = \{\} \ then\ l\ (snd\ x) \ else\ r\ (snd\ x))$

lemma *coprod-rec-eq*:

shows *coprod-rec-inl-eq* [*simp*]: $coprod-rec\ l\ r\ (inl\ a) = l\ a$

and *coprod-rec-inr-eq* [*simp*]: $coprod-rec\ l\ r\ (inr\ b) = r\ b$

unfolding *coprod-rec-def inl-def inr-def* **by** *auto*

```

lemma mono-coprod-left: mono ( $\lambda A. A \coprod B$ )
  by (intro monoI) auto

lemma mono-coprod-right: mono ( $\lambda B. A \coprod B$ )
  by (intro monoI) auto

end

```

24 Cardinals

```

theory Cardinals
  imports
    Coproduct
    Ordinals
    Transport.Functions-Bijection
    Transport.Equivalence-Relations
    Transport.Functions-Surjective
begin

```

Summary Translation of equipollence, cardinality and cardinal addition from HOL-Library and [3].

It illustrates that equipollence is an equivalence relationship and cardinal addition is commutative and associative. Finally, we derive the connection between set addition and cardinal addition.

```

lemma inverse-on-if-THE-eq-if-injectice:
  assumes injective f
  shows inverse f ( $\lambda z. \text{THE } y. z = f y$ )
  using assms injectiveD by fastforce

lemma inverse-on-if-injectice:
  assumes injective f
  obtains g where inverse f g
  using assms inverse-on-if-THE-eq-if-injectice by blast

```

```

unbundle no-HOL-groups-syntax no-HOL-ascii-syntax

```

Equipollence Two sets X and Y are said to be equipollent if there exists a bijection between f and g .

definition *equipollent* $X Y \equiv \exists f g. \text{bijection-on } (\text{mem-of } X) (\text{mem-of } Y) (f :: \text{set} \Rightarrow \text{set}) g$

```

bundle hotg-equipollent-syntax begin notation equipollent (infixl  $\approx$  50) end
bundle no-hotg-equipollent-syntax begin no-notation equipollent (infixl  $\approx$  50)
end

```

unbundle *hotg-equipollent-syntax*

lemma *equipollentI* [*intro*]:

assumes *bijection-on* (*mem-of* *X*) (*mem-of* *Y*) (*f* :: *set* \Rightarrow *set*) *g*
shows $X \approx Y$
using *assms* **by** (*auto simp: equipollent-def*)

lemma *equipollentE* [*elim*]:

assumes $X \approx Y$
obtains *f g* **where** *bijection-on* (*mem-of* *X*) (*mem-of* *Y*) (*f* :: *set* \Rightarrow *set*) *g*
using *assms* **by** (*auto simp: equipollent-def*)

lemma *reflexive-equipollent*: *reflexive* (\approx)

using *bijection-on-self-id* **by** *auto*

lemma *symmetric-equipollent*: *symmetric* (\approx)

by (*intro symmetricI*) (*auto dest: bijection-on-right-left-if-bijection-on-left-right*)

lemma *inverse-on-compI*:

fixes *P* :: '*a* \Rightarrow *bool* **and** *P'* :: '*b* \Rightarrow *bool*
and *f* :: '*a* \Rightarrow '*b* **and** *g* :: '*b* \Rightarrow '*a* **and** *f'* :: '*b* \Rightarrow '*c* **and** *g'* :: '*c* \Rightarrow '*b*
assumes *inverse-on* *P f g*
and *inverse-on* *P' f' g'*
and (*P* \Rightarrow_m *P'*) *f*
shows *inverse-on* *P (f' \circ f) (g \circ g')*
using *assms* **by** (*intro inverse-onI*) (*auto dest!: inverse-onD*)

The lemma shows that the composition of two bijections results in another bijection.

lemma *bijection-on-compI*:

fixes *P* :: '*a* \Rightarrow *bool* **and** *P'* :: '*b* \Rightarrow *bool* **and** *P''* :: '*c* \Rightarrow *bool*
and *f* :: '*a* \Rightarrow '*b* **and** *g* :: '*b* \Rightarrow '*a* **and** *f'* :: '*b* \Rightarrow '*c* **and** *g'* :: '*c* \Rightarrow '*b*
assumes *bijection-on* *P P' f g*
and *bijection-on* *P' P'' f' g'*
shows *bijection-on* *P P'' (f' \circ f) (g \circ g')*
using *assms* **by** (*intro bijection-onI*)
(*auto intro: dep-mono-wrt-pred-comp-dep-mono-wrt-pred-compI' inverse-on-compI elim!: bijection-onE*)

lemma *transitive-equipollent*: *transitive* (\approx)

by (*intro transitiveI*) (*blast intro: bijection-on-compI*)

lemma *preorder-equipollent*: *preorder* (\approx)

by (*intro preorderI transitive-equipollent reflexive-equipollent*)

lemma *partial-equivalence-rel-equipollent*: *partial-equivalence-rel* (\approx)

by (*intro partial-equivalence-relI transitive-equipollent symmetric-equipollent*)

lemma *equivalence-rel-equipollent*: *equivalence-rel* (\approx)

by (intro equivalence-relI partial-equivalence-rel-equipollent reflexive-equipollent)

Cardinality Cardinality is defined as in [3], https://foss.heptapod.net/isa-afp/afp-devel/-/blob/06458dfa40c7b4aaae855a37ae77993cb4c8c18/thys/ZFC_in_HOL/ZFC_Cardinals.thy#L1785. The cardinality of a set X is defined as the smallest ordinal number α such that there exists a bijection between X and α . Further details can be found in https://en.wikipedia.org/wiki/Cardinal_number.

definition *cardinality* ($X :: \text{set}$) \equiv (LEAST Y . ordinal $Y \wedge X \approx Y$)

bundle *hotg-cardinality-syntax* **begin notation** *cardinality* ($|-|$) **end**
bundle *no-hotg-cardinality-syntax* **begin no-notation** *cardinality* ($|-|$) **end**
unbundle *hotg-cardinality-syntax*

lemma *Least-eq-Least-if-iff*:
assumes $\bigwedge Z. P\ Z \longleftrightarrow Q\ Z$
shows (LEAST Z . $P\ Z$) = (LEAST Z . $Q\ Z$)
using *assms* **by** *simp*

lemma *cardinality-eq-if-equipollent*:
assumes $X \approx Y$
shows $|X| = |Y|$
unfolding *cardinality-def* **using** *assms* *transitive-equipollent* *symmetric-equipollent*
by (intro *Least-eq-Least-if-iff*) (blast dest: *symmetricD*)

The next lemma shows that X is equipollent with *cardinality* of X . Order types are necessary to prove it. See https://foss.heptapod.net/isa-afp/afp-devel/-/blob/06458dfa40c7b4aaae855a37ae77993cb4c8c18/thys/ZFC_in_HOL/ZFC_Cardinals.thy#L1829.

lemma *cardinal-equipollent-self* [*iff*]: $|X| \approx X$
sorry

lemma *cardinality-cardinality-eq-cardinality* [*simp*]: $||X|| = |X|$
by (intro *cardinality-eq-if-equipollent* *cardinal-equipollent-self*)

Cardinal Addition Cardinal addition is the cardinality of the disjoint union of two sets.

definition *cardinal-add* $\kappa\ \mu \equiv |\kappa \amalg \mu|$

bundle *hotg-cardinal-add-syntax* **begin notation** *cardinal-add* (**infixl** \oplus 65) **end**
bundle *no-hotg-cardinal-add-syntax* **begin no-notation** *cardinal-add* (**infixl** \oplus 65) **end**
unbundle *hotg-cardinal-add-syntax*

lemma *cardinal-add-eq-cardinality-coprod*: $\kappa \oplus \mu = |\kappa \amalg \mu|$
unfolding *cardinal-add-def* **..**

lemma *equipollent-coprod-self-commute*: $X \coprod Y \approx Y \coprod X$
by (*intro equipollentI*[**where** $?f = \text{coprod-rec inr inl}$ **and** $?g = \text{coprod-rec inr inl}$])
(fastforce dest: inverse-onD)

lemma *cardinal-add-comm*: $X \oplus Y = Y \oplus X$
unfolding *cardinal-add-eq-cardinality-coprod*
by (*intro cardinality-eq-if-equipollent cardinality-eq-if-equipollent equipollent-coprod-self-commute*)

lemma *coprod-zero-epoll*: $\{\} \coprod X \approx X$
by (*intro equipollentI*[**where** $?f = \text{coprod-rec inr id}$ **and** $?g = \text{inr}$] *bijection-onI*
inverse-onI)
auto

0 is the left neutral element for cardinal addition.

corollary *zero-cardinal-add-eq-cardinality-self*: $0 \oplus X = |X|$
unfolding *cardinal-add-eq-cardinality-coprod*
by (*intro cardinality-eq-if-equipollent coprod-zero-epoll*)

lemma *coprod-assoc-epoll*: $(X \coprod Y) \coprod Z \approx X \coprod (Y \coprod Z)$
proof (*intro equipollentI*)
show *bijection-on* (*mem-of* $((X \coprod Y) \coprod Z)$) (*mem-of* $(X \coprod (Y \coprod Z))$)
(coprod-rec (coprod-rec inl (inr \circ inl)) (inr \circ inr))
(coprod-rec (inl \circ inl) (coprod-rec (inl \circ inr) inr))
by (*intro bijection-onI inverse-onI dep-mono-wrt-predI*) *auto*
qed

lemma *cardinality-lift-eq-cardinality-right*: $|\text{lift } X \ Y| = |Y|$
proof (*intro cardinality-eq-if-equipollent equipollentI*)
let $?f = \lambda z. \text{THE } y. y \in Y \wedge z = X + y$
let $?g = ((+) X)$
from *inverse-on-if-injective* **show** *bijection-on* (*mem-of* $(\text{lift } X \ Y)$) (*mem-of* Y)
 $?f \ ?g$
by (*intro bijection-onI dep-mono-wrt-predI*)
(auto intro: the1I2 simp: lift-eq-repl-add)
qed

lemma *equipollent-bin-union-coprod-if-bin-inter-eq-empty*:
assumes $X \cap Y = \{\}$
shows $X \cup Y \approx X \coprod Y$
proof –
let $?l = \lambda z. \text{if } z \in X \text{ then inl } z \text{ else inr } z$
let $?r = \text{coprod-rec id id}$
from *assms* **have** *bijection-on* (*mem-of* $(X \cup Y)$) (*mem-of* $(X \coprod Y)$) $?l \ ?r$
by (*intro bijection-onI dep-mono-wrt-predI inverse-onI*) *auto*
then show *?thesis* **by** *blast*
qed

lemma *equipollent-coprod-if-equipollent*:
assumes $X \approx X'$

```

and  $Y \approx Y'$ 
shows  $X \coprod Y \approx X' \coprod Y'$ 
proof –
  obtain  $fX$   $gX$   $fY$   $gY$  where bijections:
     $\text{bijection-on } (\text{mem-of } X) (\text{mem-of } X') (fX :: \text{set} \Rightarrow \text{set}) gX$ 
     $\text{bijection-on } (\text{mem-of } Y) (\text{mem-of } Y') (fY :: \text{set} \Rightarrow \text{set}) gY$ 
  using assms by (elim equipollentE)
  let  $?f = \text{coprod-rec } (\text{inl} \circ fX) (\text{inr} \circ fY)$ 
  let  $?g = \text{coprod-rec } (\text{inl} \circ gX) (\text{inr} \circ gY)$ 
  have  $\text{bijection-on } (\text{mem-of } (X \coprod Y)) (\text{mem-of } (X' \coprod Y')) ?f ?g$ 
  apply (intro bijection-onI dep-mono-wrt-predI inverse-onI)
  apply (auto elim: mem-coprodE)
  using bijections by (auto intro: elim: mem-coprodE bijection-onE simp: bijec-
    tion-on-left-right-eq-self
    dest: bijection-on-right-left-if-bijection-on-left-right)
  then show ?thesis by auto
qed

lemma cardinal-add-assoc:  $(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)$ 
proof –
  have  $|(X \coprod Y)| \coprod Z \approx (X \coprod Y) \coprod Z$ 
  using reflexive-equipollent by (blast intro: equipollent-coprod-if-equipollent dest:
    reflexiveD)
  moreover have  $\dots \approx X \coprod (Y \coprod Z)$  by (simp add: coprod-assoc-eqpoll)
  moreover have  $\dots \approx X \coprod |Y \coprod Z|$ 
  using partial-equivalence-rel-equipollent
  by (blast intro: equipollent-coprod-if-equipollent dest: reflexiveD symmetricD)
  ultimately have  $|(X \coprod Y)| \coprod Z \approx X \coprod |Y \coprod Z|$  using transitive-equipollent
by blast
  then show ?thesis
  by (auto intro: cardinality-eq-if-equipollent simp: cardinal-add-eq-cardinality-coprod)
qed

lemma cardinality-bin-union-eq-cardinal-add-if-bin-inter-eq-empty:
  assumes  $X \cap Y = \{\}$ 
  shows  $|X \cup Y| = |X| \oplus |Y|$ 
proof –
  have replacement:  $\bigwedge X. X \approx |X|$ 
  using symmetric-equipollent symmetricD[of equipollent] cardinal-equipollent-self
  by auto
  have cardinalization:  $X \coprod Y \approx |X| \coprod |Y|$ 
  using symmetric-equipollent equipollent-coprod-if-equipollent by (force dest:
    symmetricD)
  from assms have  $X \cup Y \approx X \coprod Y$  by (intro equipollent-bin-union-coprod-if-bin-inter-eq-empty)
  auto
  moreover have  $\dots \approx |X| \coprod |Y|$ 
  using replacement equipollent-coprod-if-equipollent by auto
  ultimately have  $X \cup Y \approx |X| \coprod |Y|$  using transitiveD[OF transitive-equipollent]
by blast

```

```

from cardinal-add-eq-cardinality-coprod have  $|X| \oplus |Y| = ||X| \coprod |Y||$  by simp
show  $|X \cup Y| = |X| \oplus |Y|$ 
proof –
  have  $X \cup Y \approx |X| \coprod |Y|$ 
  using assms cardinalization equipollent-bin-union-coprod-if-bin-inter-eq-empty

  transitiveD[OF transitive-equipollent] by blast
then have  $|X \cup Y| = ||X| \coprod |Y||$  using cardinality-eq-if-equipollent by auto
then show ?thesis by (subst cardinal-add-eq-cardinality-coprod)
  qed
qed

```

The next theorem shows that set addition is compatible with cardinality addition.

```

theorem cardinality-add-eq-cardinal-add:  $|X + Y| = |X| \oplus |Y|$ 
using cardinality-lift-eq-cardinality-right
by (simp add: add-eq-bin-union-lift cardinality-bin-union-eq-cardinal-add-if-bin-inter-eq-empty)

```

A similar theorem shows that set multiplication is compatible with cardinality multiplication, but it is not proven here. See

end

```

theory Arithmetics
imports
  SAddition
  SMultiplication
  Cardinals
  Ordinals
begin

```

Summary Translation of generalised arithmetics from https://www.isa-afp.org/entries/ZFC_in_HOL.html.

end

24.1 Antisymmetric

```

theory SBinary-Relations-Antisymmetric
imports
  Pairs
begin

```

definition antisymmetric $D R \equiv \forall x y \in D. \langle x, y \rangle \in R \wedge \langle y, x \rangle \in R \longrightarrow x = y$

```

lemma antisymmetricI [intro]:
  assumes  $\bigwedge x y. x \in D \implies y \in D \implies \langle x, y \rangle \in R \implies \langle y, x \rangle \in R \implies x = y$ 
  shows antisymmetric  $D R$ 
  using assms unfolding antisymmetric-def by blast

```

```

lemma antisymmetricD:
  assumes antisymmetric D R
  and  $x \in D \ y \in D$ 
  and  $\langle x, y \rangle \in R \ \langle y, x \rangle \in R$ 
  shows  $x = y$ 
  using assms unfolding antisymmetric-def by blast

```

end

24.2 Connected

```

theory SBinary-Relations-Connected
  imports
    Pairs
begin

```

```

definition connected D R  $\equiv \forall x \ y \in D. x \neq y \longrightarrow \langle x, y \rangle \in R \vee \langle y, x \rangle \in R$ 

```

```

lemma connectedI [intro]:
  assumes  $\bigwedge x \ y. x \in D \implies y \in D \implies x \neq y \implies \langle x, y \rangle \in R \vee \langle y, x \rangle \in R$ 
  shows connected D R
  using assms unfolding connected-def by blast

```

```

lemma connectedE:
  assumes connected D R
  and  $x \in D \ y \in D$ 
  and  $x \neq y$ 
  obtains  $\langle x, y \rangle \in R \mid \langle y, x \rangle \in R$ 
  using assms unfolding connected-def by auto

```

end

25 Replacement on Function-Like Predicates

```

theory Replacement-Predicates
  imports Comprehension
begin

```

Replacement based on function-like predicates, as formulated in first-order theories.

```

definition replace ::  $\langle \text{set} \Rightarrow (\text{set} \Rightarrow \text{set} \Rightarrow \text{bool}) \Rightarrow \text{set} \rangle$ 
  where replace A P =  $\{ \text{THE } y. P \ x \ y \mid x \in \{x \in A \mid \exists ! y. P \ x \ y\} \}$ 

```

```

bundle hotg-replacement-syntax
begin
syntax
  -replace :: ⟨[pttrn, pttrn, set, set ⇒ set ⇒ bool] ⇒ set⟩ ({- | / - ∈ -, -})
end
bundle no-hotg-replacement-syntax
begin
no-syntax
  -replace :: ⟨[pttrn, pttrn, set, set ⇒ set ⇒ bool] ⇒ set⟩ ({- | / - ∈ -, -})
end
unbundle hotg-replacement-syntax
translations
  {y | x ∈ A, Q} ⇒ CONST replace A (λx y. Q)

lemma mem-replace-iff:
  b ∈ {y | x ∈ A, P x y} ⟷ (∃ x ∈ A. P x b ∧ (∀ y. P x y ⟶ y = b))
proof -
  have b ∈ {y | x ∈ A, P x y} ⟷ (∃ x ∈ A. (∃!y. P x y) ∧ b = (THE y. P x y))
    using replace-def by auto
  also have ... ⟷ (∃ x ∈ A. P x b ∧ (∀ y. P x y ⟶ y = b))
  proof (rule bex-cong[OF refl])
    fix x assume x ∈ A
    show
      (∃!y. P x y) ∧ b = (THE y. P x y) ⟷ P x b ∧ (∀ y. P x y ⟶ y = b)
      (is ?lhs ⟷ ?rhs)
    proof
      assume ?lhs
      then have ex1: ∃!y. P x y and b-eq: b = (THE y. P x y) by auto
      show ?rhs
      proof
        from ex1 show P x b unfolding b-eq by (rule theI')
        with ex1 show ∀ y. P x y ⟶ y = b unfolding Ex1-def by blast
      qed
    next
      assume ?rhs
      then have P: P x b and uniq: ∧y. P x y ⟶ y = b by auto
      show ?lhs
      proof
        from P show ∃!y. P x y by (rule ex1I)
        then show b = (THE y. P x y) using P by (rule the1-equality[symmetric])
      qed
    qed
  qed
  finally show ?thesis .
qed

```

```

lemma replaceI [intro!]:
  ⟦P x b; x ∈ A; ∧y. P x y ⟶ y = b⟧ ⟹ b ∈ {y | x ∈ A, P x y}

```

by (*rule mem-replace-iff*[*THEN iffD2*]) *blast*

lemma *replaceE*:

assumes $b \in \{y \mid x \in A, P\ x\ y\}$

obtains x **where** $x \in A$ **and** $P\ x\ b$ **and** $\bigwedge y. P\ x\ y \implies y = b$

using *assms* **by** (*rule mem-replace-iff*[*THEN iffD1*, *THEN bexE*]) *blast*

lemma *replaceE'* [*elim!*]:

assumes $b \in \{y \mid x \in A, P\ x\ y\}$

obtains x **where** $x \in A$ $P\ x\ b$

using *assms* **by** (*elim replaceE*) *blast*

lemma *replace-cong* [*cong*]:

$\llbracket A = B; \bigwedge x\ y. x \in B \implies P\ x\ y \longleftrightarrow Q\ x\ y \rrbracket \implies \{y \mid x \in A, P\ x\ y\} = \{y \mid x \in B, Q\ x\ y\}$

by (*rule eqI'*) (*simp add: mem-replace-iff*)

lemma *mono-replace-set*: *mono* ($\lambda A. \{y \mid x \in A, P\ x\ y\}$)

by (*intro monoI*) (*auto elim!: replaceE*)

end

25.1 Functions on Relations

theory *SBinary-Relation-Functions*

imports

Pairs

Replacement-Predicates

begin

25.1.1 Inverse

definition *set-rel-inv* $R \equiv \{\langle y, x \rangle \mid \langle x, y \rangle \in \{p \in R \mid \exists x\ y. p = \langle x, y \rangle\}\}$

bundle *hotg-rel-inv-syntax*

begin

notation *set-rel-inv* $((^{-1}) [1000])$

end

bundle *no-hotg-rel-inv-syntax*

begin

no-notation *set-rel-inv* $((^{-1}) [1000])$

end

unbundle *no-rel-inv-syntax*

unbundle *hotg-rel-inv-syntax*

lemma *mem-set-rel-invI* [*intro*]:

assumes $\langle x, y \rangle \in R$
shows $\langle y, x \rangle \in R^{-1}$
using *assms* **unfolding** *set-rel-inv-def* **by** *auto*

lemma *mem-set-rel-invE* [*elim!*]:
assumes $p \in R^{-1}$
obtains $x\ y$ **where** $p = \langle y, x \rangle \ \langle x, y \rangle \in R$
using *assms* **unfolding** *set-rel-inv-def* *uncurry-def* **by** (*auto*)

lemma *set-rel-inv-pairs-eq* [*simp*]: $(A \times B)^{-1} = B \times A$
by *auto*

lemma *set-rel-inv-empty-eq* [*simp*]: $\{\}^{-1} = \{\}$
by *auto*

lemma *set-rel-inv-inv-eq*: $R^{-1-1} = \{p \in R \mid \exists x\ y. p = \langle x, y \rangle\}$
by *auto*

lemma *mono-set-rel-inv*: *mono set-rel-inv*
by (*intro monoI*) *auto*

25.1.2 Extensions and Restricts

definition *extend* $x\ y\ R \equiv \text{insert } \langle x, y \rangle\ R$

lemma *mem-extendI* [*intro*]: $\langle x, y \rangle \in \text{extend } x\ y\ R$
unfolding *extend-def* **by** *blast*

lemma *mem-extendI'*:
assumes $p \in R$
shows $p \in \text{extend } x\ y\ R$
unfolding *extend-def* **using** *assms* **by** *blast*

lemma *mem-extendE* [*elim*]:
assumes $p \in \text{extend } x\ y\ R$
obtains $p = \langle x, y \rangle \mid p \neq \langle x, y \rangle \ p \in R$
using *assms* **unfolding** *extend-def* **by** *blast*

lemma *extend-eq-self-if-pair-mem* [*simp*]: $\langle x, y \rangle \in R \implies \text{extend } x\ y\ R = R$
by (*auto intro: mem-extendI'*)

lemma *insert-pair-eq-extend*: $\text{insert } \langle x, y \rangle\ R = \text{extend } x\ y\ R$
by (*auto intro: mem-extendI'*)

lemma *mono-extend-set*: *mono (extend x y)*
by (*intro monoI*) (*auto intro: mem-extendI'*)

definition *glue* $\mathcal{R} \equiv \bigcup \mathcal{R}$

lemma *mem-glueI* [*intro*]:
 assumes $p \in R$
 and $R \in \mathcal{R}$
 shows $p \in \text{glue } \mathcal{R}$
 using *assms* **unfolding** *glue-def* **by** *blast*

lemma *mem-glueE* [*elim*]:
 assumes $p \in \text{glue } \mathcal{R}$
 obtains R where $p \in R$ $R \in \mathcal{R}$
 using *assms* **unfolding** *glue-def* **by** *blast*

lemma *glue-empty-eq* [*simp*]: $\text{glue } \{\} = \{\}$ **by** *auto*

lemma *glue-singleton-eq* [*simp*]: $\text{glue } \{R\} = R$ **by** *auto*

lemma *mono-glue*: *mono glue*
by (*intro monoI*) *auto*

overloading
 $\text{set-restrict-left-pred} \equiv \text{restrict-left} :: \text{set} \Rightarrow (\text{set} \Rightarrow \text{bool}) \Rightarrow \text{set}$
 $\text{set-restrict-left-set} \equiv \text{restrict-left} :: \text{set} \Rightarrow \text{set} \Rightarrow \text{set}$
 $\text{set-restrict-right-pred} \equiv \text{restrict-right} :: \text{set} \Rightarrow (\text{set} \Rightarrow \text{bool}) \Rightarrow \text{set}$
 $\text{set-restrict-right-set} \equiv \text{restrict-right} :: \text{set} \Rightarrow \text{set} \Rightarrow \text{set}$
begin
definition $\text{set-restrict-left-pred } R \ P \equiv \{p \in R \mid \exists x \ y. \ P \ x \wedge p = \langle x, y \rangle\}$
definition $\text{set-restrict-left-set } (R :: \text{set}) \ A \equiv \text{restrict-left } R \ (\text{mem-of } A)$
definition $\text{set-restrict-right-pred } R \ P \equiv \{p \in R \mid \exists x \ y. \ P \ y \wedge p = \langle x, y \rangle\}$
definition $\text{set-restrict-right-set } (R :: \text{set}) \ A \equiv \text{restrict-right } R \ (\text{mem-of } A)$
end

lemma *set-restrict-left-set-eq-set-restrict-left* [*simp*]: $(R :: \text{set}) \upharpoonright_A :: \text{set} = R \upharpoonright_{\text{mem-of } A}$
unfolding *set-restrict-left-set-def* **by** *simp*

lemma *set-restrict-right-set-eq-set-restrict-right* [*simp*]: $(R :: \text{set}) \upharpoonright_A :: \text{set} = R \upharpoonright_{\text{mem-of } A}$
unfolding *set-restrict-right-set-def* **by** *simp*

lemma *mem-set-restrict-leftI* [*intro*]:
 assumes $\langle x, y \rangle \in R$
 and $P \ x$
 shows $\langle x, y \rangle \in R \upharpoonright_P$
 using *assms* **unfolding** *set-restrict-left-pred-def* **by** *blast*

lemma *mem-set-restrict-leftE* [*elim*]:
 assumes $p \in R \upharpoonright_P$
 obtains $x \ y$ where $p = \langle x, y \rangle$ $P \ x$ $\langle x, y \rangle \in R$
 using *assms* **unfolding** *set-restrict-left-pred-def* **by** *blast*

lemma *mem-set-restrict-rightI* [*intro*]:

```

assumes  $\langle x, y \rangle \in R$ 
and  $P\ y$ 
shows  $\langle x, y \rangle \in R \upharpoonright_P$ 
using assms unfolding set-restrict-right-pred-def by blast

lemma mem-set-restrict-rightE [elim]:
  assumes  $p \in R \upharpoonright_P$ 
  obtains  $x\ y$  where  $p = \langle x, y \rangle\ P\ y\ \langle x, y \rangle \in R$ 
  using assms unfolding set-restrict-right-pred-def by blast

lemma set-restrict-left-empty-eq [simp]:  $\{\} \upharpoonright_P :: \text{set} \Rightarrow \text{bool} = \{\}$  by auto

lemma set-restrict-left-empty-eq' [simp]:  $R \upharpoonright_{\{\}} = \{\}$  by auto

lemma set-restrict-left-subset-self [iff]:  $R \upharpoonright_P :: \text{set} \Rightarrow \text{bool} \subseteq R$  by auto

lemma set-restrict-left-dep-pairs-eq-dep-pairs-collect [simp]:
   $(\sum x \in A. B\ x) \upharpoonright_P = (\sum x \in \{a \in A \mid P\ a\}. B\ x)$ 
  by auto

lemma set-restrict-left-dep-pairs-eq-dep-pairs-bin-inter [simp]:
   $(\sum x \in A. B\ x) \upharpoonright_{A'} = (\sum x \in A \cap A'. B\ x)$ 
  by simp

lemma set-restrict-left-subset-dep-pairs-if-subset-dep-pairs [intro]:
  assumes  $R \subseteq \sum x \in A. B\ x$ 
  shows  $R \upharpoonright_P \subseteq \sum x \in \{x \in A \mid P\ x\}. B\ x$ 
  using assms by auto

lemma set-restrict-left-restrict-left-eq-restrict-left [simp]:
  fixes  $R :: \text{set}$  and  $P :: \text{set} \Rightarrow \text{bool}$ 
  shows  $(R \upharpoonright_P) \upharpoonright_P = R \upharpoonright_P$ 
  by auto

lemma mono-set-restrict-left-set: mono  $(\lambda R. R \upharpoonright_P :: \text{set} \Rightarrow \text{bool})$ 
  by (intro monoI) auto

lemma mono-set-restrict-left-pred: mono  $(\lambda P. (R :: \text{set}) \upharpoonright_P :: \text{set} \Rightarrow \text{bool})$ 
  by (intro monoI) auto

consts agree ::  $'a \Rightarrow 'b \Rightarrow \text{bool}$ 

overloading
  agree-pred-set  $\equiv \text{agree} :: (\text{set} \Rightarrow \text{bool}) \Rightarrow \text{set} \Rightarrow \text{bool}$ 
  agree-set-set  $\equiv \text{agree} :: \text{set} \Rightarrow \text{set} \Rightarrow \text{bool}$ 
begin
  definition agree-pred-set  $(P :: \text{set} \Rightarrow \text{bool})\ \mathcal{R} \equiv \forall R\ R' \in \mathcal{R}. R \upharpoonright_P = R' \upharpoonright_P$ 
  definition agree-set-set  $(A :: \text{set}) :: \text{set} \Rightarrow \text{set} \Rightarrow \text{bool} \equiv \text{agree}\ (\text{mem-of}\ A)$ 

```

end

lemma *agree-set-set-eq-agree-set* [*simp*]: $(\text{agree } (A :: \text{set}) :: \text{set} \Rightarrow -) = \text{agree } (\text{mem-of } A)$
unfolding *agree-set-set-def* **by** *simp*

lemma *agree-set-set-iff-agree-set* [*iff*]: $\text{agree } (A :: \text{set}) (\mathcal{R} :: \text{set}) \longleftrightarrow \text{agree } (\text{mem-of } A) \mathcal{R}$
by *simp*

lemma *agreeI* [*intro*]:
assumes $\bigwedge x y R R'. P x \implies R \in \mathcal{R} \implies R' \in \mathcal{R} \implies \langle x, y \rangle \in R \implies \langle x, y \rangle \in R'$
shows $\text{agree } P \mathcal{R}$
using *assms* **unfolding** *agree-pred-set-def* **by** *blast*

lemma *agreeD*:
assumes $\text{agree } P \mathcal{R}$
and $P x$
and $R \in \mathcal{R} \ R' \in \mathcal{R}$
and $\langle x, y \rangle \in R$
shows $\langle x, y \rangle \in R'$
proof –
from *assms*(2, 5) **have** $\langle x, y \rangle \in R \upharpoonright_P$ **by** (*intro mem-set-restrict-leftI*)
moreover from *assms*(1, 3–4) **have** $\dots = R' \upharpoonright_P$ **unfolding** *agree-pred-set-def*
by *blast*
ultimately show *?thesis* **by** *auto*
qed

lemma *antimono-agree-pred*: $\text{antimono } (\lambda P. \text{agree } (P :: \text{set} \Rightarrow \text{bool}) (\mathcal{R} :: \text{set}))$
by (*intro antimonoI*) (*auto dest: agreeD*)

lemma *antimono-agree-set*: $\text{antimono } (\lambda \mathcal{R}. \text{agree } (P :: \text{set} \Rightarrow \text{bool}) (\mathcal{R} :: \text{set}))$
by (*intro antimonoI*) (*auto dest: agreeD*)

lemma *set-restrict-left-eq-set-restrict-left-if-agree*:
fixes $P :: \text{set} \Rightarrow \text{bool}$
assumes $\text{agree } P \mathcal{R}$
and $R \in \mathcal{R} \ R' \in \mathcal{R}$
shows $R \upharpoonright_P = R' \upharpoonright_P$
using *assms* **by** (*auto dest: agreeD*)

lemma *eq-if-subset-dep-pairs-if-agree*:
assumes $\text{agree } A \mathcal{R}$
and *subset-dep-pairs*: $\bigwedge R. R \in \mathcal{R} \implies \exists B. R \subseteq \sum x \in A. B x$
and $R \in \mathcal{R}$
and $R' \in \mathcal{R}$
shows $R = R'$
proof –

from *subset-dep-pairs*[*OF* $\langle R \in \mathcal{R} \rangle$] **have** $R = R \setminus A$ **by** *auto*
also with *assms* **have** $\dots = R' \upharpoonright_A$
by ((*subst set-restrict-left-set-eq-set-restrict-left*)+,
intro set-restrict-left-eq-set-restrict-left-if-agree)
auto
also from *subset-dep-pairs*[*OF* $\langle R' \in \mathcal{R} \rangle$] **have** $\dots = R'$ **by** *auto*
finally show *?thesis* .
qed

lemma *subset-if-agree-if-subset-dep-pairs*:
assumes *subset-dep-pairs*: $R \subseteq \sum x \in A. B\ x$
and $R \in \mathcal{R}$
and *agree* $A\ \mathcal{R}$
and $R' \in \mathcal{R}$
shows $R \subseteq R'$
using *assms* **by** (*auto simp: agreeD*[**where** $?R=R$])

25.1.3 Domain and Range

definition $\text{dom } R \equiv \{x \mid p \in R, \exists y. p = \langle x, y \rangle\}$

lemma *mem-domI* [*intro*]:
assumes $\langle x, y \rangle \in R$
shows $x \in \text{dom } R$
using *assms* **unfolding** *dom-def* **by** *fast*

lemma *mem-domE* [*elim!*]:
assumes $x \in \text{dom } R$
obtains y **where** $\langle x, y \rangle \in R$
using *assms* **unfolding** *dom-def* **by** *blast*

lemma *mono-dom*: *mono dom*
by (*intro monoI*) *auto*

lemma *dom-empty-eq* [*simp*]: $\text{dom } \{\} = \{\}$
by *auto*

lemma *dom-union-eq* [*simp*]: $\text{dom } (\bigcup \mathcal{R}) = \bigcup \{\text{dom } R \mid R \in \mathcal{R}\}$
by *auto*

lemma *dom-bin-union-eq* [*simp*]: $\text{dom } (R \cup S) = \text{dom } R \cup \text{dom } S$
by *auto*

lemma *dom-collect-eq* [*simp*]: $\text{dom } \{\langle f\ x, g\ x \rangle \mid x \in A\} = \{f\ x \mid x \in A\}$
by *auto*

lemma *dom-extend-eq* [*simp*]: $\text{dom } (\text{extend } x\ y\ R) = \text{insert } x\ (\text{dom } R)$
by (*rule eqI*) (*auto intro: mem-extendI'*)

lemma *dom-dep-pairs-eqI* [intro]:
 assumes $\bigwedge x. B\ x \neq \{\}$
 shows $\text{dom } (\sum x \in A. B\ x) = A$
 using *assms* **by** (intro *eqI*) *auto*

lemma *dom-restrict-left-eq* [simp]: $\text{dom } (R \restriction P) = \{x \in \text{dom } R \mid P\ x\}$
by *auto*

lemma *dom-restrict-left-set-eq* [simp]: $\text{dom } (R \restriction A) = \text{dom } R \cap A$ **by** *simp*

lemma *glue-subset-dep-pairsI*:
 fixes \mathcal{R} defines $D \equiv \bigcup R \in \mathcal{R}. \text{dom } R$
 assumes *all-subset-dep-pairs*: $\bigwedge R. R \in \mathcal{R} \implies \exists A. R \subseteq \sum x \in A. B\ x$
 shows $\text{glue } \mathcal{R} \subseteq \sum x \in D. (B\ x)$
proof
 fix p assume $p \in \text{glue } \mathcal{R}$
 with *all-subset-dep-pairs* obtain $R\ A$ where $p \in R\ R \in \mathcal{R}\ R \subseteq \sum x \in A. B\ x$
 by *blast*
 then obtain $x\ y$ where $p = \langle x, y \rangle\ x \in \text{dom } R\ y \in B\ x$ **by** *blast*
 with $\langle R \in \mathcal{R} \rangle$ have $x \in D$ **unfolding** *D-def* **by** *auto*
 with $\langle p = \langle x, y \rangle \rangle\ \langle y \in B\ x \rangle$ show $p \in \sum x \in D. (B\ x)$ **by** *auto*
qed

definition *rng* $R \equiv \{y \mid p \in R, \exists x. p = \langle x, y \rangle\}$

lemma *mem-rngI* [intro]:
 assumes $\langle x, y \rangle \in R$
 shows $y \in \text{rng } R$
 using *assms* **unfolding** *rng-def* **by** *fast*

lemma *mem-rngE* [elim!]:
 assumes $y \in \text{rng } R$
 obtains x where $\langle x, y \rangle \in R$
 using *assms* **unfolding** *rng-def* **by** *blast*

lemma *mono-rng*: *mono* *rng*
by (intro *monoI*) *auto*

lemma *rng-empty-eq* [simp]: $\text{rng } \{\} = \{\}$
by *auto*

lemma *rng-union-eq* [simp]: $\text{rng } (\bigcup \mathcal{R}) = \bigcup \{\text{rng } R \mid R \in \mathcal{R}\}$
by *auto*

lemma *rng-bin-union-eq* [simp]: $\text{rng } (R \cup S) = \text{rng } R \cup \text{rng } S$
by *auto*

lemma *rng-collect-eq* [simp]: $\text{rng } \{\langle f\ x, g\ x \rangle \mid x \in A\} = \{g\ x \mid x \in A\}$
by *auto*

lemma *rng-extend-eq* [*simp*]: $\text{rng } (\text{extend } x \ y \ R) = \text{insert } y \ (\text{rng } R)$
by (*rule eqI*) (*auto intro: mem-extendI'*)

lemma *rng-dep-pairs-eq* [*simp*]: $\text{rng } (\sum x \in A. B \ x) = (\bigcup x \in A. B \ x)$
by *auto*

lemma *dom-rel-inv-eq-rng* [*simp*]: $\text{dom } R^{-1} = \text{rng } R$
by *auto*

lemma *rng-rel-inv-eq-dom* [*simp*]: $\text{rng } R^{-1} = \text{dom } R$
by *auto*

25.1.4 Composition

definition *set-comp* $S \ R \equiv$
 $\{p \in \text{dom } R \times \text{rng } S \mid \exists z. \langle \text{fst } p, z \rangle \in R \wedge \langle z, \text{snd } p \rangle \in S\}$

bundle *hotg-comp-syntax* **begin notation** *set-comp* (*infixr* \circ 60) **end**
bundle *no-hotg-comp-syntax* **begin no-notation** *set-comp* (*infixr* \circ 60) **end**
unbundle *no-comp-syntax*
unbundle *hotg-comp-syntax*

lemma *mem-compI* [*intro!*]:
assumes $\langle x, y \rangle \in R$
and $\langle y, z \rangle \in S$
shows $\langle x, z \rangle \in S \circ R$
using *assms* **unfolding** *set-comp-def* **by** *auto*

lemma *mem-compE* [*elim!*]:
assumes $p \in S \circ R$
obtains $x \ y \ z$ **where** $\langle x, y \rangle \in R \ \langle y, z \rangle \in S \ p = \langle x, z \rangle$
using *assms* **unfolding** *set-comp-def* **by** *auto*

lemma *dep-pairs-comp-pairs-eq*:
 $((\sum x \in B. (C \ x)) \circ (A \times B)) = A \times (\bigcup x \in B. (C \ x))$
by *auto*

lemma *set-comp-assoc*: $T \circ S \circ R = (T \circ S) \circ R$
by *auto*

lemma *mono-set-comp-left*: *mono* $(\lambda R. R \circ S)$
by (*intro monoI*) *auto*

lemma *mono-set-comp-right*: *mono* $(\lambda S. R \circ S)$
by (*intro monoI*) *auto*

25.1.5 Diagonal

definition *diag* $A \equiv \{\langle a, a \rangle \mid a \in A\}$

lemma *mem-diagI* [*intro!*]: $a \in A \implies \langle a, a \rangle \in \text{diag } A$
unfolding *diag-def* **by** *auto*

lemma *mem-diagE* [*elim!*]:
assumes $p \in \text{diag } A$
obtains a **where** $a \in A$ $p = \langle a, a \rangle$
using *assms* **unfolding** *diag-def* **by** *auto*

lemma *mono-diag*: *mono diag*
by (*intro monoI*) *auto*

end

25.2 Injective

theory *SBinary-Relations-Injective*
imports
Transport.Functions-Monotone
SBinary-Relation-Functions
begin

consts *set-injective-on* :: $'a \Rightarrow \text{set} \Rightarrow \text{bool}$

overloading

set-injective-on-pred \equiv *set-injective-on* :: $(\text{set} \Rightarrow \text{bool}) \Rightarrow \text{set} \Rightarrow \text{bool}$
set-injective-on-set \equiv *set-injective-on* :: $\text{set} \Rightarrow \text{set} \Rightarrow \text{bool}$

begin

definition *set-injective-on-pred* P $R \equiv$
 $\forall x x' y. P x \wedge P x' \wedge \langle x, y \rangle \in R \wedge \langle x', y \rangle \in R \longrightarrow x = x'$

definition *set-injective-on-set* B $R \equiv$ *set-injective-on* (*mem-of* B) R

end

lemma *set-injective-on-set-iff-set-injective-on* [*iff*]:
 $\text{set-injective-on } B \ R \longleftrightarrow \text{set-injective-on } (\text{mem-of } B) \ R$
unfolding *set-injective-on-set-def* **by** *simp*

lemma *set-injective-onI* [*intro*]:
assumes $\bigwedge x x' y. P x \implies P x' \implies \langle x, y \rangle \in R \implies \langle x', y \rangle \in R \implies x = x'$
shows *set-injective-on* $P \ R$
using *assms* **unfolding** *set-injective-on-pred-def* **by** *blast*

lemma *set-injective-onD*:
assumes *set-injective-on* $P \ R$
and $P x \ P x'$
and $\langle x, y \rangle \in R \ \langle x', y \rangle \in R$
shows $x = x'$
using *assms* **unfolding** *set-injective-on-pred-def* **by** *blast*

lemma *antimono-set-injective-on-pred*:
antimono ($\lambda P. \text{set-injective-on } (P :: \text{set} \Rightarrow \text{bool}) \ R$)
by (*intro antimonoI*) (*auto dest: set-injective-onD*)

lemma *antimono-set-injective-on-set*:
antimono ($\lambda R. \text{set-injective-on } (P :: \text{set} \Rightarrow \text{bool}) \ R$)
by (*intro antimonoI*) (*auto dest: set-injective-onD*)

lemma *set-injective-on-compI*:
fixes $P :: \text{set} \Rightarrow \text{bool}$
assumes *set-injective-on* (*dom* R) R
and *set-injective-on* (*rng* $R \cap \text{dom } S$) S
shows *set-injective-on* P ($S \circ R$)
using *assms* **by** (*auto dest: set-injective-onD*)

end

25.3 Irreflexive

theory *SBinary-Relations-Irreflexive*
imports
Pairs
begin

definition *irreflexive* $D \ R \equiv \forall x \in D. \langle x, x \rangle \notin R$

lemma *irreflexiveI* [*intro*]:
assumes $\bigwedge x. x \in D \implies \langle x, x \rangle \notin R$
shows *irreflexive* $D \ R$
using *assms* **unfolding** *irreflexive-def* **by** *blast*

lemma *irreflexiveD*:
assumes *irreflexive* $D \ R$
and $x \in D$
shows $\langle x, x \rangle \notin R$
using *assms* **unfolding** *irreflexive-def* **by** *blast*

end

25.4 Left Total

theory *SBinary-Relations-Left-Total*
imports
SBinary-Relation-Functions
begin

consts *set-left-total-on* $:: 'a \Rightarrow \text{set} \Rightarrow \text{bool}$

overloading

set-left-total-on-pred \equiv *set-left-total-on* :: (*set* \Rightarrow *bool*) \Rightarrow *set* \Rightarrow *bool*

set-left-total-on-set \equiv *set-left-total-on* :: *set* \Rightarrow *set* \Rightarrow *bool*

begin

definition *set-left-total-on-pred* *P R* $\equiv \forall x. P\ x \longrightarrow x \in \text{dom } R$

definition *set-left-total-on-set* *A R* $\equiv \text{set-left-total-on } (\text{mem-of } A)\ R$

end

lemma *set-left-total-on-set-iff-set-left-total-on* [iff]:

set-left-total-on *A R* $\longleftrightarrow \text{set-left-total-on } (\text{mem-of } A)\ R$

unfolding *set-left-total-on-set-def* **by** *simp*

lemma *set-left-total-onI* [intro]:

assumes $\bigwedge x. P\ x \Longrightarrow x \in \text{dom } R$

shows *set-left-total-on* *P R*

unfolding *set-left-total-on-pred-def* **using** *assms* **by** *blast*

lemma *set-left-total-onE* [elim]:

assumes *set-left-total-on* *P R*

and *P x*

obtains $x \in \text{dom } R$

using *assms* **unfolding** *set-left-total-on-pred-def* **by** *blast*

lemma *antimono-set-left-total-on-pred*:

antimono ($\lambda P. \text{set-left-total-on } (P :: \text{set} \Rightarrow \text{bool})\ R$)

by (*intro antimonoI*) *fastforce*

lemma *mono-set-left-total-on-set*:

mono ($\lambda R. \text{set-left-total-on } (P :: \text{set} \Rightarrow \text{bool})\ R$)

by (*intro monoI*) *fastforce*

lemma *set-left-total-on-set-iff-subset-dom* [iff]:

set-left-total-on *A R* $\longleftrightarrow A \subseteq \text{dom } R$

by *auto*

lemma *set-left-total-on-inf-restrict-leftI*:

fixes *P P' :: set* \Rightarrow *bool*

assumes *set-left-total-on* *P R*

shows *set-left-total-on* (*P* \sqcap *P'*) *R*_{|*P'*}

using *assms* **by** (*intro set-left-total-onI*) *auto*

lemma *set-left-total-on-compI*:

fixes *P :: set* \Rightarrow *bool*

assumes *set-left-total-on* *P R*

and *set-left-total-on* (*rng* (*R*_{|*P*})) *S*

shows *set-left-total-on* *P* (*S* \circ *R*)

using *assms* **by** (*intro set-left-total-onI*) *auto*

end

25.5 Reflexive

```
theory SBinary-Relations-Reflexive
  imports
    Pairs
begin
```

definition *reflexive* $D\ R \equiv \forall x \in D. \langle x, x \rangle \in R$

lemma *reflexiveI* [intro]:
 assumes $\bigwedge x. x \in D \implies \langle x, x \rangle \in R$
 shows *reflexive* $D\ R$
 using *assms* **unfolding** *reflexive-def* **by** *blast*

lemma *reflexiveD*:
 assumes *reflexive* $D\ R$
 and $x \in D$
 shows $\langle x, x \rangle \in R$
 using *assms* **unfolding** *reflexive-def* **by** *blast*

end

25.5.1 Right Unique

```
theory SBinary-Relations-Right-Unique
  imports
    SBinary-Relation-Functions
begin
```

consts *set-right-unique-on* :: $'a \Rightarrow \text{set} \Rightarrow \text{bool}$

overloading

set-right-unique-on-pred $\equiv \text{set-right-unique-on} :: (\text{set} \Rightarrow \text{bool}) \Rightarrow \text{set} \Rightarrow \text{bool}$

set-right-unique-on-set $\equiv \text{set-right-unique-on} :: \text{set} \Rightarrow \text{set} \Rightarrow \text{bool}$

begin

definition *set-right-unique-on-pred* $P\ R \equiv$
 $\forall x\ y\ y'. P\ x \wedge \langle x, y \rangle \in R \wedge \langle x, y' \rangle \in R \longrightarrow y = y'$

definition *set-right-unique-on-set* $A\ R \equiv \text{set-right-unique-on}\ (\text{mem-of}\ A)\ R$

end

lemma *set-right-unique-on-set-iff-set-right-unique-on* [iff]:
 $\text{set-right-unique-on}\ A\ R \longleftrightarrow \text{set-right-unique-on}\ (\text{mem-of}\ A)\ R$
unfolding *set-right-unique-on-set-def* **by** *simp*

lemma *set-right-unique-onI* [intro]:
 assumes $\bigwedge x\ y\ y'. P\ x \implies \langle x, y \rangle \in R \implies \langle x, y' \rangle \in R \implies y = y'$

```

shows set-right-unique-on  $P$   $R$ 
using assms unfolding set-right-unique-on-pred-def by blast

lemma set-right-unique-onD:
  assumes set-right-unique-on  $P$   $R$ 
  and  $P$   $x$ 
  and  $\langle x, y \rangle \in R$   $\langle x, y' \rangle \in R$ 
  shows  $y = y'$ 
  using assms unfolding set-right-unique-on-pred-def by blast

lemma antimono-set-right-unique-on-pred:
  antimono  $(\lambda P. \text{set-right-unique-on } (P :: \text{set} \Rightarrow \text{bool}) R)$ 
  by (intro antimonoI) (auto dest: set-right-unique-onD)

lemma antimono-set-right-unique-on-set:
  antimono  $(\lambda R. \text{set-right-unique-on } (P :: \text{set} \Rightarrow \text{bool}) R)$ 
  by (intro antimonoI) (auto dest: set-right-unique-onD)

lemma set-right-unique-on-glueI:
  fixes  $P :: \text{set} \Rightarrow \text{bool}$ 
  assumes  $\bigwedge R R'. R \in \mathcal{R} \implies R' \in \mathcal{R} \implies \text{set-right-unique-on } P (\text{glue } \{R, R'\})$ 
  shows set-right-unique-on  $P (\text{glue } \mathcal{R})$ 
proof
  fix  $x y y'$  assume  $P$   $x$   $\langle x, y \rangle \in \text{glue } \mathcal{R}$   $\langle x, y' \rangle \in \text{glue } \mathcal{R}$ 
  with assms obtain  $R R'$  where  $R \in \mathcal{R}$   $R' \in \mathcal{R}$   $\langle x, y \rangle \in R$   $\langle x, y' \rangle \in R'$ 
  and runique: set-right-unique-on  $P (\text{glue } \{R, R'\})$ 
  by auto
  then have  $\langle x, y \rangle \in (\text{glue } \{R, R'\})$   $\langle x, y' \rangle \in (\text{glue } \{R, R'\})$  by auto
  with  $\langle P \ x \rangle$  runique show  $y = y'$  by (intro set-right-unique-onD)
qed

lemma set-right-unique-on-compI:
  fixes  $P :: \text{set} \Rightarrow \text{bool}$ 
  assumes set-right-unique-on  $P$   $R$ 
  and set-right-unique-on  $(\text{rng } (R \upharpoonright_P) \cap \text{dom } S)$   $S$ 
  shows set-right-unique-on  $P (S \circ R)$ 
  using assms by (auto dest: set-right-unique-onD)

end



## 25.6 Surjective

theory SBinary-Relations-Surjective
imports
  SBinary-Relation-Functions
begin

consts set-surjective-at ::  $'a \Rightarrow \text{set} \Rightarrow \text{bool}$ 

```

overloading

set-surjective-at-pred \equiv *set-surjective-at* :: (*set* \Rightarrow *bool*) \Rightarrow *set* \Rightarrow *bool*
set-surjective-at-set \equiv *set-surjective-at* :: *set* \Rightarrow *set* \Rightarrow *bool*

begin

definition *set-surjective-at-pred* *P R* $\equiv \forall y. P\ y \longrightarrow y \in \text{rng}\ R$

definition *set-surjective-at-set* *B R* $\equiv \text{set-surjective-at}\ (\text{mem-of}\ B)\ R$

end

lemma *set-surjective-at-set-iff-set-surjective-at* [iff]:

set-surjective-at *B R* $\longleftrightarrow \text{set-surjective-at}\ (\text{mem-of}\ B)\ R$

unfolding *set-surjective-at-set-def* **by** *simp*

lemma *set-surjective-atI* [intro]:

assumes $\bigwedge y. P\ y \Longrightarrow y \in \text{rng}\ R$

shows *set-surjective-at* *P R*

unfolding *set-surjective-at-pred-def* **using** *assms* **by** *blast*

lemma *set-surjective-atE* [elim]:

assumes *set-surjective-at* *P R*

and *P y*

obtains *x* **where** $\langle x, y \rangle \in R$

using *assms* **unfolding** *set-surjective-at-pred-def* **by** *blast*

lemma *antimono-set-surjective-at-pred*:

antimono ($\lambda P. \text{set-surjective-at}\ (P :: \text{set} \Rightarrow \text{bool})\ R$)

by (*intro antimonoI*) *fastforce*

lemma *mono-set-surjective-at-set*:

mono ($\lambda R. \text{set-surjective-at}\ (P :: \text{set} \Rightarrow \text{bool})\ R$)

by (*intro monoI*) *fastforce*

lemma *subset-rng-if-set-surjective-at* [simp]:

set-surjective-at *B R* $\Longrightarrow B \subseteq \text{rng}\ R$

by *auto*

lemma *set-surjective-at-compI*:

fixes *P* :: *set* \Rightarrow *bool*

assumes *surj-R*: *set-surjective-at* (*dom S*) *R*

and *surj-S*: *set-surjective-at* *P S*

shows *set-surjective-at* *P* (*S* \circ *R*)

proof

fix *y* **assume** *P y*

then obtain *x* **where** $\langle x, y \rangle \in S$ **using** *surj-S* **by** *auto*

moreover then have $x \in \text{dom}\ S$ **by** *auto*

moreover then obtain *z* **where** $\langle z, x \rangle \in R$ **using** *surj-R* **by** *auto*

ultimately show $y \in \text{rng}\ (S \circ R)$ **by** *blast*

qed

end

25.7 Symmetric

```
theory SBinary-Relations-Symmetric
  imports
    Pairs
begin
```

definition *symmetric* $D R \equiv \forall x y \in D. \langle x, y \rangle \in R \longrightarrow \langle y, x \rangle \in R$

lemma *symmetricI* [intro]:
 assumes $\bigwedge x y. x \in D \implies y \in D \implies \langle x, y \rangle \in R \implies \langle y, x \rangle \in R$
 shows *symmetric* $D R$
 using *assms* **unfolding** *symmetric-def* **by** *blast*

lemma *symmetricD*:
 assumes *symmetric* $D R$
 and $x \in D \ y \in D$
 and $\langle x, y \rangle \in R$
 shows $\langle y, x \rangle \in R$
 using *assms* **unfolding** *symmetric-def* **by** *blast*

end

25.8 Transitive

```
theory SBinary-Relations-Transitive
  imports
    Pairs
begin
```

definition *transitive* $D R \equiv \forall x y z \in D. \langle x, y \rangle \in R \wedge \langle y, z \rangle \in R \longrightarrow \langle x, z \rangle \in R$

lemma *transitiveI* [intro]:
 assumes
 $\bigwedge x y z. x \in D \implies y \in D \implies z \in D \implies \langle x, y \rangle \in R \implies \langle y, z \rangle \in R \implies \langle x, z \rangle$
 $\in R$
 shows *transitive* $D R$
 using *assms* **unfolding** *transitive-def* **by** *blast*

lemma *transitiveD*:
 assumes *transitive* $D R$
 and $x \in D \ y \in D \ z \in D$
 and $\langle x, y \rangle \in R \ \langle y, z \rangle \in R$
 shows $\langle x, z \rangle \in R$
 using *assms* **unfolding** *transitive-def* **by** *blast*

end

25.9 Basic Properties

```
theory SBinary-Relation-Properties
  imports
    SBinary-Relations-Antisymmetric
    SBinary-Relations-Connected
    SBinary-Relations-Injective
    SBinary-Relations-Irreflexive
    SBinary-Relations-Left-Total
    SBinary-Relations-Reflexive
    SBinary-Relations-Right-Unique
    SBinary-Relations-Surjective
    SBinary-Relations-Symmetric
    SBinary-Relations-Transitive
begin
```

end

26 Set-Theoretic Binary Relations

```
theory SBinary-Relations
  imports
    SBinary-Relation-Properties
    SBinary-Relation-Functions
begin
```

end

26.1 Evaluation of Functions

```
theory SFunctions-Base
  imports
    SBinary-Relations-Right-Unique
    SBinary-Relations-Left-Total
begin
```

definition *eval* $f\ x \equiv THE\ y.\ \langle x, y \rangle \in f$

```
bundle hotg-eval-syntax begin notation eval (( $\cdot$ ) [999, 1000] 999) end
bundle no-hotg-eval-syntax begin no-notation eval (( $\cdot$ ) [999, 1000] 999) end
unbundle hotg-eval-syntax
```

lemma *eval-eqI*:

assumes *set-right-unique-on* P f
and P x
and $\langle x, y \rangle \in f$
shows $f'x = y$
using *assms* **unfolding** *eval-def* **by** (*auto dest: set-right-unique-onD*)

lemma *eval-eqI'*:
assumes *set-right-unique-on* $\{x\}$ f
and $\langle x, y \rangle \in f$
shows $f'x = y$
using *assms* **by** (*auto intro: eval-eqI*)

lemma *pair-eval-mem-if-ex1-pair-mem*:
assumes $\exists!y. \langle x, y \rangle \in f$
shows $\langle x, f'x \rangle \in f$
using *assms* **unfolding** *eval-def* **by** (*rule theI'*)

lemma *pair-eval-mem-if-mem-dom-if-set-right-unique-on*:
assumes *set-right-unique-on* $\{x\}$ f
and $x \in \text{dom } f$
shows $\langle x, f'x \rangle \in f$
using *assms*
by (*intro pair-eval-mem-if-ex1-pair-mem*) (*auto dest: set-right-unique-onD*)

lemma *eval-singleton-eq* [*simp*]: $\{\langle x, y \rangle\}'x = y$
by (*rule eval-eqI*) *auto*

lemma *eval-repl-eq* [*iff*]: $x \in A \implies \{\langle a, f a \rangle \mid a \in A\}'x = f x$
by (*auto intro: eval-eqI*)

lemma *extend-eval-eq* [*simp*]: $x \notin \text{dom } f \implies (\text{extend } x \ y \ f)'x = y$
by (*auto intro!: eval-eqI' set-right-unique-onI*)

lemma *extend-eval-eq'* [*simp*]:
 $x \neq y \implies (\text{extend } y \ z \ f)'x = f'x$
unfolding *extend-def eval-def* **by** (*auto iff: mem-insert-iff*)

lemma *bin-union-eval-eq-left-eval* [*simp*]:
 $x \notin \text{dom } g \implies (f \cup g)'x = f'x$
unfolding *eval-def* **by** (*cases* $\exists y. \langle x, y \rangle \in g$) *auto*

lemma *bin-union-eval-eq-right-eval* [*simp*]:
 $x \notin \text{dom } f \implies (f \cup g)'x = g'x$
unfolding *eval-def* **by** (*cases* $\exists y. \langle x, y \rangle \in f$) *auto*

lemma *restriction-eval-eq* [*simp*]:
assumes P x
shows $(f \restriction P)'x = f'x$
using *assms* **unfolding** *eval-def set-restrict-left-pred-def* **by** *auto*

lemma *glue-eval-eqI*:
assumes $\bigwedge f f'. f \in F \implies f' \in F \implies \text{set-right-unique-on } \{x\} (\text{glue } \{f, f'\})$
and $f \in F$
and $x \in \text{dom } f$
shows $(\text{glue } F)'x = f'x$
proof (*rule eval-eqI* [**where** $?P = \text{mem-of } \{x\}$], *fold set-right-unique-on-set-def*)
from *assms(1)* **show** $\text{set-right-unique-on } \{x\} (\text{glue } F)$
by (*auto intro: set-right-unique-on-glueI*)
from *assms(1)* [*OF assms(2) assms(2)*] **have** $\text{set-right-unique-on } \{x\} f$ **by** *auto*
with *assms(3)* **have** $\langle x, f'x \rangle \in f$
by (*intro pair-eval-mem-if-mem-dom-if-set-right-unique-on*)
with *assms(2)* **show** $\langle x, f'x \rangle \in (\text{glue } F)$ **by** *auto*
qed *simp*

26.1.1 Dependent Functions

definition *dep-functions* $A B \equiv$
 $\{f \in \text{powerset } (\sum x \in A. B x) \mid \text{set-left-total-on } A f \wedge \text{set-right-unique-on } A f\}$

abbreviation *functions* $A B \equiv \text{dep-functions } A (\lambda x. B)$

bundle *hotg-functions-syntax*
begin
syntax
 $\text{-set-functions-telescope} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \text{ (infixr } \rightarrow_s 55)$
end
bundle *no-hotg-functions-syntax*
begin
no-syntax
 $\text{-set-functions-telescope} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \text{ (infixr } \rightarrow_s 55)$
end
unbundle *hotg-functions-syntax*

translations

$(x y \in A) \rightarrow_s B \rightarrow (x \in A)(y \in A) \rightarrow_s B$
 $(x \in A) \text{ args } \rightarrow_s B \rightarrow (x \in A) \rightarrow_s \text{args} \rightarrow_s B$
 $(x \in A) \rightarrow_s B \equiv \text{CONST } \text{dep-functions } A (\lambda x. B)$
 $A \rightarrow_s B \equiv \text{CONST } \text{functions } A B$

lemma *mem-dep-functionsI* [*intro*]:
assumes $f \subseteq (\sum x \in A. (B x))$
and $\text{set-left-total-on } A f$
and $\text{set-right-unique-on } A f$
shows $f \in (x \in A) \rightarrow_s (B x)$
using *assms* **unfolding** *dep-functions-def* **by** *auto*

lemma *mem-dep-functionsE* [*elim*]:
assumes $f \in (x \in A) \rightarrow_s (B x)$

obtains $f \subseteq \sum x \in A. (B\ x)$ *set-left-total-on A f set-right-unique-on A f*
using *assms* **unfolding** *dep-functions-def* **by** *blast*

lemma *dep-functions-cong* [*cong*]:
 $\llbracket A = A'; \bigwedge x. x \in A' \implies B\ x = B'\ x \rrbracket \implies (x \in A) \rightarrow_s (B\ x) = (x \in A') \rightarrow_s (B'\ x)$
unfolding *dep-functions-def* **by** *simp*

lemma *mem-functions-if-mem-dep-functions*:
 $f \in (x \in A) \rightarrow_s (B\ x) \implies f \in (A \rightarrow_s (\bigcup x \in A. B\ x))$
unfolding *dep-functions-def* **by** *auto*

lemma *dom-eq-if-mem-dep-functions* [*simp*]:
assumes $f \in (x \in A) \rightarrow_s (B\ x)$
shows $\text{dom } f = A$
using *assms* **by** (*elim mem-dep-functionsE, intro eq-if-subset-if-subset*) *auto*

lemma *rng-subset-if-mem-dep-functions* [*simp*]:
assumes $f \in (x \in A) \rightarrow_s (B\ x)$
shows $\text{rng } f \subseteq (\bigcup x \in A. B\ x)$
proof –
from *assms* **have** $f \subseteq \sum x \in A. (B\ x)$ **by** (*elim mem-dep-functionsE*)
then **have** $\text{rng } f \subseteq \text{rng } (\sum x \in A. (B\ x))$ **by** *blast*
also **have** $\dots \subseteq (\bigcup x \in A. B\ x)$ **by** *simp*
finally **show** *?thesis* .
qed

lemma *fst-snd-eq-pair-if-mem-dep-function* [*simp*]:
assumes $f \in (x \in A) \rightarrow_s (B\ x)$
and $p \in f$
shows $\langle \text{fst } p, \text{snd } p \rangle = p$
using *assms* **by** (*auto elim!: mem-dep-functionsE*)

lemma *pair-eval-mem-if-mem-if-mem-dep-functions* [*elim*]:
assumes $f \in (x \in A) \rightarrow_s (B\ x)$
and $x \in A$
shows $\langle x, f'x \rangle \in f$
proof –
from *assms* **have** $x \in \text{dom } f$ **by** *simp*
then **show** *?thesis* **using** *assms*
by (*elim mem-dep-functionsE mem-domE, intro pair-eval-mem-if-ex1-pair-mem*)
(auto dest: set-right-unique-onD)
qed

lemma *pair-mem-iff-eval-eq-if-mem-dom-dep-function*:
assumes $f \in (x \in A) \rightarrow_s (B\ x)$
and $x \in A$
shows $\langle x, y \rangle \in f \longleftrightarrow f'x = y$
proof

assume $\langle x, y \rangle \in f$
moreover have $\langle x, f'x \rangle \in f$ **using** *assms* **by** *auto*
ultimately show $f'x = y$ **using** *assms*
by (*auto dest: set-right-unique-onD*)
qed (*insert assms, auto*)

lemma *fst-mem-if-mem-dep-function*:
 $\llbracket f \in (x \in A) \rightarrow_s (B\ x); p \in f \rrbracket \implies \text{fst } p \in A$
by (*auto elim!: mem-dep-functionsE*)

lemma *snd-mem-if-mem-dep-function*:
 $\llbracket f \in (x \in A) \rightarrow_s (B\ x); p \in f \rrbracket \implies \text{snd } p \in B\ (\text{fst } p)$
by (*auto elim!: mem-dep-functionsE*)

lemma *mem-dom-if-pair-mem-dep-function*:
 $\llbracket f \in (x \in A) \rightarrow_s (B\ x); \langle x, y \rangle \in f \rrbracket \implies x \in A$
using *fst-mem-if-mem-dep-function* [**where** $?p = \langle x, y \rangle$] **by** *auto*

lemma *mem-codom-if-pair-mem-dep-function*:
 $\llbracket f \in (x \in A) \rightarrow_s (B\ x); \langle x, y \rangle \in f \rrbracket \implies y \in B\ x$
using *snd-mem-if-mem-dep-function* [**where** $?p = \langle x, y \rangle$] **by** *auto*

lemma *eval-mem-if-mem-if-mem-dep-functions* [*elim*]:
 $\llbracket f \in (x \in A) \rightarrow_s (B\ x); x \in A \rrbracket \implies f'x \in B\ x$
using *mem-codom-if-pair-mem-dep-function*
by (*blast dest: pair-eval-mem-if-mem-if-mem-dep-functions*)

lemma *eval-eq-if-pair-mem-dep-function* [*simp*]:
assumes $f \in (x \in A) \rightarrow_s (B\ x)$
and $\langle x, y \rangle \in f$
shows $f'x = y$
using *assms fst-mem-if-mem-dep-function* [*OF assms*]
by (*auto iff: pair-mem-iff-eval-eq-if-mem-dom-dep-function*)

lemma *mem-dom-dep-functionE*:
assumes $f \in (x \in A) \rightarrow_s (B\ x)$
and $x \in A$
obtains y **where** $f'x = y$ $y \in B\ x$
using *assms eval-mem-if-mem-if-mem-dep-functions* **by** *auto*

lemma *mem-dep-functionE* [*elim*]:
assumes $f \in (x \in A) \rightarrow_s (B\ x)$
and $p \in f$
obtains $x\ y$ **where** $p = \langle x, y \rangle$ $x \in A$ $y \in B\ x$ $f'x = y$
proof –
assume *hyp*: $\bigwedge x\ y. p = \langle x, y \rangle \implies x \in A \implies y \in B\ x \implies f'x = y \implies \text{thesis}$
obtain $x\ y$ **where** [*simp*]: $p = \langle x, y \rangle$ **using** *assms*
by (*auto elim!: mem-dep-functionsE*)
show *thesis*

```

proof (intro hyp[of x y])
  from fst-mem-if-mem-dep-function[OF assms] show  $x \in A$  by simp
  from snd-mem-if-mem-dep-function[OF assms] show  $y \in B\ x$  by simp
  from assms show  $f'x = y$  by auto
qed fact
qed

```

```

lemma repl-eval-eq-dep-function [simp]:
  assumes  $f \in (x \in A) \rightarrow_s (B\ x)$ 
  shows  $\{\langle x, f'x \rangle \mid x \in A\} = f$ 
  using assms by (intro eqI) auto

```

Note: functions are not contravariant on their domain.

```

lemma mem-dep-functions-covariant-codom:
  assumes  $f \in (x \in A) \rightarrow_s (B\ x)$ 
  and  $\bigwedge x. x \in A \implies f'x \in B\ x \implies f'x \in B'\ x$ 
  shows  $f \in (x \in A) \rightarrow_s (B'\ x)$ 
  by (rule mem-dep-functionsE[OF assms(1)], intro mem-dep-functionsI)
  (insert assms, auto)

```

```

corollary mem-dep-functions-covariant-codom-subset:
  assumes  $f \in (x \in A) \rightarrow_s (B\ x)$ 
  and  $\bigwedge x. x \in A \implies B\ x \subseteq B'\ x$ 
  shows  $f \in (x \in A) \rightarrow_s (B'\ x)$ 
  using assms(2) by (intro mem-dep-functions-covariant-codom[OF assms(1)])
  auto

```

```

lemma eq-if-mem-if-mem-agree-if-mem-dep-functions:
  assumes mem-dep-functions:  $\bigwedge f. f \in F \implies \exists B. f \in (x \in A) \rightarrow_s (B\ x)$ 
  and agree A F
  and  $f \in F$ 
  and  $g \in F$ 
  shows  $f = g$ 
  using assms
proof –
  have  $\bigwedge f. f \in F \implies \exists B. f \subseteq \sum x \in A. (B\ x)$  by (blast dest: mem-dep-functions)
  with assms show ?thesis by (intro eq-if-subset-dep-pairs-if-agree)
qed

```

```

lemma subset-if-agree-if-mem-dep-functions:
  assumes  $f \in (x \in A) \rightarrow_s (B\ x)$ 
  and  $f \in F$ 
  and agree A F
  and  $g \in F$ 
  shows  $f \subseteq g$ 
  using assms
  by (elim mem-dep-functionsE subset-if-agree-if-subset-dep-pairs) auto

```

```

lemma agree-if-eval-eq-if-mem-dep-functions:

```

assumes *mem-dep-functions*: $\bigwedge f. f \in F \implies \exists B. f \in (x \in A) \rightarrow_s (B \ x)$
and $\bigwedge f \ g \ x. f \in F \implies g \in F \implies x \in A \implies f'x = g'x$
shows *agree A F*
proof (*subst agree-set-set-iff-agree-set, rule agreeI*)
fix $x \ y \ f \ g$ **assume** *hyps*: $f \in F \ g \in F \ x \in A$ **and** $\langle x, y \rangle \in f$
then have $y = f'x$ **using** *assms(1)* **by** *auto*
also have $\dots = g'x$ **by** (*fact assms(2)[OF hyps]*)
finally have *y-eq*: $y = g'x$.
from *assms(1)[OF $\langle g \in F \rangle$]* **obtain** B **where** $g \in (x \in A) \rightarrow_s (B \ x)$ **by** *blast*
with *y-eq pair-mem-iff-eval-eq-if-mem-dom-dep-function $\langle x \in A \rangle$*
show $\langle x, y \rangle \in g$ **by** *blast*
qed

lemma *eq-if-agree-if-mem-dep-functions*:
assumes $f \in (x \in A) \rightarrow_s (B \ x) \ g \in (x \in A) \rightarrow_s (B \ x)$
and *agree A {f, g}*
shows $f = g$
using *assms*
by (*intro eq-if-mem-if-mem-agree-if-mem-dep-functions[of {f, g}]*) *auto*

lemma *dep-functions-ext*:
assumes $f \in (x \in A) \rightarrow_s (B \ x) \ g \in (x \in A) \rightarrow_s (B \ x)$
and $\bigwedge x. x \in A \implies f'x = g'x$
shows $f = g$
using *assms*
by (*intro eq-if-agree-if-mem-dep-functions*)
(auto intro:
agree-if-eval-eq-if-mem-dep-functions[unfolded agree-set-set-iff-agree-set])

lemma *dep-functions-eval-eqI*:
assumes $f \in (x \in A) \rightarrow_s (B \ x) \ g \in (x \in A') \rightarrow_s (B' \ x)$
and $f \subseteq g$
and $x \in A \cap A'$
shows $f'x = g'x$
proof –
from *assms* **have** $\langle x, f'x \rangle \in g$ **and** $\langle x, g'x \rangle \in g$ **by** *auto*
then show *?thesis* **using** *assms* **by** *auto*
qed

lemma *dep-functions-eq-if-subset*:
assumes *f-mem*: $f \in (x \in A) \rightarrow_s (B \ x)$
and *g-mem*: $g \in (x \in A) \rightarrow_s (B' \ x)$
and $f \subseteq g$
shows $f = g$
proof (*rule eqI*)
fix p **assume** $p \in g$
with *g-mem* **obtain** $x \ y$ **where** [*simp*]: $p = \langle x, y \rangle \ g'x = y \ x \in A$ **by** *auto*
with *assms* **have** [*simp*]: $f'x = g'x$ **by** (*intro dep-functions-eval-eqI*) *auto*
show $p \in f$ **using** *f-mem*

by (auto iff: pair-mem-iff-eval-eq-if-mem-dom-dep-function)
qed (insert assms, auto)

lemma *ex-dom-mem-dep-functions-iff*:
 $(\exists A. f \in (x \in A) \rightarrow_s (B\ x)) \longleftrightarrow f \in (x \in \text{dom } f) \rightarrow_s (B\ x)$
 by auto

lemma *mem-dep-functions-empty-dom-iff-eq-empty* [iff]:
 $(f \in (x \in \{\}) \rightarrow_s (B\ x)) \longleftrightarrow f = \{\}$
 by auto

lemma *empty-mem-dep-functions*: $\{\} \in (x \in \{\}) \rightarrow_s (B\ x)$ by simp

lemma *eq-singleton-if-mem-functions-singleton* [simp]:
 $f \in \{a\} \rightarrow_s \{b\} \implies f = \{\langle a, b \rangle\}$
 by auto

lemma *singleton-mem-functionsI* [intro]: $y \in B \implies \{\langle x, y \rangle\} \in \{x\} \rightarrow_s B$
 by auto

lemma *mem-dep-functions-collectI*:
 assumes *f-mem*: $f \in (x \in A) \rightarrow_s (B\ x)$
 and $\bigwedge x. x \in A \implies P\ x\ (f'x)$
 shows $f \in (x \in A) \rightarrow_s \{y \in B\ x \mid P\ x\ y\}$
 by (rule mem-dep-functions-covariant-codom) (insert assms, auto)

lemma *mem-dep-functions-collectD*:
 assumes $f \in (x \in A) \rightarrow_s \{y \in B\ x \mid P\ x\ y\}$
 shows $f \in (x \in A) \rightarrow_s (B\ x)$ and $\bigwedge x. x \in A \implies P\ x\ (f'x)$
proof –
 from assms show $f \in (x \in A) \rightarrow_s (B\ x)$
 by (rule mem-dep-functions-covariant-codom-subset) auto
 fix *x* assume $x \in A$
 with assms show $P\ x\ (f'x)$
 by (auto dest: pair-eval-mem-if-mem-if-mem-dep-functions)
 qed

end

26.2 Lambda Abstractions

theory *SFunctions-Lambda*
 imports *SFunctions-Base*
 begin

definition *lambda* $A\ f \equiv \{\langle x, f\ x \rangle \mid x \in A\}$

bundle *hotg-lambda-syntax*

```

begin
syntax
  -lam :: [pttrns, set, set  $\Rightarrow$  set]  $\Rightarrow$  set ((2λ- ∈ -./ -) 60)
  -lam2 :: [pttrns, set, set  $\Rightarrow$  set]  $\Rightarrow$  set
end
bundle no-hotg-lambda-syntax
begin
no-syntax
  -lam :: [pttrns, set, set  $\Rightarrow$  set]  $\Rightarrow$  set ((2λ- ∈ -./ -) 60)
  -lam2 :: [pttrns, set, set  $\Rightarrow$  set]  $\Rightarrow$  set
end
unbundle hotg-lambda-syntax

translations
  λx xs ∈ A. f  $\rightarrow$  CONST lambda A (λx. -lam2 xs A f)
  -lam2 x A f  $\rightarrow$  λx ∈ A. f
  λx ∈ A. f  $\Rightarrow$  CONST lambda A (λx. f)

lemma mem-lambdaE [elim!]:
  assumes p ∈ λx ∈ A. f x
  obtains x y where p = ⟨x, y⟩ x ∈ A y = f x
  using assms unfolding lambda-def by auto

lemma mem-lambdaD [dest]: ⟨a, b⟩ ∈ λx ∈ A. f x  $\Longrightarrow$  b = f a
  by auto

lemma lambda-cong [cong]:
  [A = A';  $\bigwedge x. x \in A \Longrightarrow f x = f' x$ ]  $\Longrightarrow$  (λx ∈ A. f x) = λx ∈ A'. f' x
  unfolding lambda-def by auto

lemma eval-lambda-eq [simp]: a ∈ A  $\Longrightarrow$  (λx ∈ A. f x) 'a = f a
  unfolding lambda-def by auto

lemma eval-lambda-uncurry-eq [simp]:
  assumes x ∈ A y ∈ B x
  shows (λp ∈  $\sum x \in A. (B x). \text{uncurry } f p$ ) '⟨x, y⟩ = f x y
  using assms by auto

lemma lambda-dep-pairs-eq-lambda-uncurry:
  (λp ∈  $\sum x \in A. (B x). f p$ ) = (λ⟨a, b⟩ ∈  $\sum x \in A. (B x). f \langle a, b \rangle$ )
  by (rule lambda-cong) auto

lemma lambda-pair-mem-if-mem [intro]: a ∈ A  $\Longrightarrow$  ⟨a, f a⟩ ∈ λx ∈ A. f x
  unfolding lambda-def by auto

lemma lambda-dom-eq [simp]: dom (λx ∈ A. f x) = A
  unfolding lambda-def by simp

lemma lambda-rng-eq [simp]: rng (λx ∈ A. f x) = {f x | x ∈ A}

```

unfolding *lambda-def* **by** *simp*

lemma *app-eq-if-mem-if-lambda-eq*:

$\llbracket (\lambda x \in A. f\ x) = \lambda x \in A. g\ x; a \in A \rrbracket \implies f\ a = g\ a$
by *auto*

lemma *lambda-mem-dep-functions* [*iff*]: $(\lambda x \in A. f\ x) \in (x \in A) \rightarrow_s \{f\ x\}$

by *auto*

lemma *lambda-mem-dep-functions-contravariant*:

assumes $f \in (x \in A) \rightarrow_s (B\ x)$

and $A' \subseteq A$

shows $(\lambda a \in A'. f'a) \in (x \in A') \rightarrow_s (B\ x)$

proof

show $(\lambda a \in A'. f'a) \subseteq \sum x \in A'. (B\ x)$

proof

fix p **assume** $p \in \lambda a \in A'. f'a$

then obtain $x\ y$ **where** $x \in A' \ y \in \{f'x\} \ p = \langle x, y \rangle$ **by** *auto*

moreover with *assms* **have** $y \in B\ x$ **by** *auto*

ultimately show $p \in \sum x \in A'. (B\ x)$ **by** *auto*

qed

qed *auto*

lemma *lambda-bin-inter-mem-dep-functionsI*:

assumes $f \in (x \in A) \rightarrow_s (B\ x)$

shows $(\lambda x \in A \cap A'. f'x) \in (x \in A \cap A') \rightarrow_s (B\ x)$

using *assms* **by** (*rule* *lambda-mem-dep-functions-contravariant*) *auto*

lemma *lambda-ext*:

assumes $f \in (x \in A) \rightarrow_s (B\ x)$

and $\bigwedge a. a \in A \implies g\ a = f'a$

shows $(\lambda a \in A. g\ a) = f$

using *assms* **by** (*intro* *eqI*) *auto*

lemma *lambda-eta* [*simp*]: $f \in (x \in A) \rightarrow_s (B\ x) \implies (\lambda x \in A. f'x) = f$

by (*rule* *dep-functions-ext*,

rule *mem-dep-functions-covariant-codom*[*OF* *lambda-mem-dep-functions*]) *auto*

Every element of *dep-functions* $A\ B$ may be expressed as a lambda abstraction

lemma *eq-lambdaE-if-mem-dep-functions*:

assumes $f \in (x \in A) \rightarrow_s (B\ x)$

obtains g **where** $f = (\lambda x \in A. g\ x)$

proof

let $?g = (\lambda x. f'x)$

from *assms* **show** $f = (\lambda x \in A. (\lambda x. f'x)\ x)$ **by** *auto*

qed

lemma *mono-lambda-set*: *mono* $(\lambda A. \lambda x \in A. f\ x)$

```

    by (intro monoI) auto

end



### 26.3 Composition



theory SFunctions-Composition
  imports SFunctions-Lambda
begin

lemma comp-mem-dep-functionsI:
  assumes f-mem:  $f \in (x \in B) \rightarrow_s (C\ x)$ 
  and g-mem:  $g \in A \rightarrow_s B$ 
  shows  $f \circ g \in (x \in A) \rightarrow_s (C\ (g'x))$ 
proof
  show  $f \circ g \subseteq \sum x \in A. (C\ (g'x))$ 
  proof
    fix p assume  $p \in f \circ g$ 
    then obtain  $x\ y\ z$  where  $\langle x, y \rangle \in g$   $\langle y, z \rangle \in f$   $p = \langle x, z \rangle$  by auto
    moreover with assms have  $x \in A$   $z \in C\ (g'x)$  by auto
    ultimately show  $p \in \sum x \in A. (C\ (g'x))$  by auto
  qed
next
  show set-right-unique-on  $A$   $(f \circ g)$ 
  proof (subst set-right-unique-on-set-iff-set-right-unique-on,
    intro set-right-unique-on-compI)
    let  $?C = \text{rng } g \upharpoonright_{\lambda x. x \in A \cap \text{dom } f}$ 
    from f-mem have  $\text{mem-of } ?C \leq \text{mem-of } B$  by auto
    moreover have set-right-unique-on  $(\text{mem-of } B)$   $f$  using f-mem by blast
    ultimately have set-right-unique-on  $(\text{mem-of } ?C)$   $f$ 
      using antimonoD[OF antimono-set-right-unique-on-pred] by auto
    then show set-right-unique-on  $?C$   $f$  by simp
  qed (insert g-mem, auto)
  from g-mem have  $\text{rng } g \subseteq B$  by auto
  then show set-left-total-on  $A$   $(f \circ g)$ 
    using assms by (subst set-left-total-on-set-iff-set-left-total-on,
      intro set-left-total-on-compI)
  auto
qed

lemma comp-eval-eq-if-mem-dep-functions [simp]:
  assumes f-mem:  $f \in (x \in B) \rightarrow_s (C\ x)$ 
  and g-mem:  $g \in A \rightarrow_s B$ 
  and x-mem:  $x \in A$ 
  shows  $(f \circ g)'x = f'(g'x)$ 
proof -
  have  $f \circ g \in (x \in A) \rightarrow_s (C\ (g'x))$ 
    using f-mem g-mem comp-mem-dep-functionsI by auto

```


with $x\text{-mem}$ **have** $\langle x, (f \circ g)'x \rangle \in f \circ g$
using $\text{pair-eval-mem-if-mem-if-mem-dep-functions}$ **by** auto
then show $(f \circ g)'x = f'(g'x)$ **using** $g\text{-mem } f\text{-mem}$ **by** auto
qed

definition $\text{set-id } A \equiv \lambda x \in A. x$

lemma $\text{set-id-eq } [\text{simp}]$: $\text{set-id } A = \lambda x \in A. x$
unfolding set-id-def **by** simp

lemma $\text{set-id-mem-dep-functions } [\text{iff}]$: $\text{set-id } A \in (x \in A) \rightarrow_s \{x\}$
by auto

lemma $\text{comp-set-id-eq } [\text{simp}]$:
assumes $f \in (x \in A) \rightarrow_s (B \ x)$
shows $f \circ \text{set-id } A = f$
proof –
from assms **have** $f \circ \text{set-id } A \in (x \in A) \rightarrow_s (B((\text{set-id } A)'x))$
by $(\text{elim comp-mem-dep-functionsI}) \text{ auto}$
then have $f \circ \text{set-id } A \in (x \in A) \rightarrow_s (B \ x)$
by $(\text{rule mem-dep-functions-covariant-codom}) \text{ auto}$
from this assms show $?thesis$
by $(\text{rule dep-functions-ext, subst comp-eval-eq-if-mem-dep-functions}) \text{ auto}$
qed

lemma $\text{set-id-comp-eq } [\text{simp}]$:
assumes $f \in A \rightarrow_s B$
shows $\text{set-id } B \circ f = f$
proof –
have $\text{set-id } B \circ f \in A \rightarrow_s B$
by $(\text{rule comp-mem-dep-functionsI}[OF - \text{assms}]) \text{ auto}$
from this assms show $?thesis$
by $(\text{rule dep-functions-ext, subst comp-eval-eq-if-mem-dep-functions})$
 $(\text{auto intro: eval-lambda-eq})$
qed

end

26.4 Extending Functions

theory $S\text{Functions-Extend-Restrict}$
imports $S\text{Functions-Base}$
begin

lemma $\text{extend-mem-dep-functionsI}$:
assumes $f\text{-dep-fun: } f \in (x \in A) \rightarrow_s (B \ x)$
and $x \notin A$
shows $\text{extend } x \ y \ f \in (x' \in \text{insert } x \ A) \rightarrow_s (\text{if } x' = x \text{ then } \{y\} \text{ else } B \ x')$

```

  (is ?lhs ∈ dep-functions ?rhs-dom ?rhs-fun)
proof
  show set-left-total-on (insert x A) (extend x y f)
proof (subst set-left-total-on-set-iff-subset-dom, rule subsetI)
  fix x' assume x' ∈ insert x A
  then show x' ∈ dom (extend x y f)
proof (rule mem-insertE)
  assume x' ∈ A
  with assms have ⟨x', f'x'⟩ ∈ f by auto
  then show x' ∈ dom (extend x y f) by auto
qed auto
qed
show set-right-unique-on (insert x A) (extend x y f) using assms by blast
qed (insert assms, auto elim!: mem-dep-functionE)

```

```

lemma extend-mem-dep-functionsI':
  assumes f ∈ (x ∈ A) →s (B x)
  and x ∉ A
  and y ∈ B x
  shows extend x y f ∈ (x ∈ insert x A) →s (B x)
proof (rule mem-dep-functions-covariant-codom)
  show extend x y f ∈ (x' ∈ insert x A) →s (if x' = x then {y} else B x')
  by (fact extend-mem-dep-functionsI[OF assms(1-2)])
qed (insert assms, auto)

```

```

lemma extend-mem-functionsI:
  assumes f ∈ A →s B
  and x ∉ A
  shows extend x y f ∈ functions (insert x A) (insert y B)
proof (rule mem-dep-functions-covariant-codom)
  show extend x y f ∈ (x' ∈ insert x A) →s (if x' = x then {y} else B)
  by (fact extend-mem-dep-functionsI[OF assms])
qed (insert assms, auto)

```

26.5 Gluing

```

lemma glue-mem-dep-functionsI:
  fixes F defines D ≡ ⋃ f ∈ F. dom f
  assumes all-fun: ∧ f. f ∈ F ⇒ ∃ A. f ∈ (x ∈ A) →s B x
  and F-right-unique: set-right-unique-on D (glue F)
  shows glue F ∈ (x ∈ D) →s B x
proof (rule mem-dep-functionsI)
  show set-left-total-on D (glue F) unfolding D-def by auto
  show glue F ⊆ ∑ x ∈ D. (B x)
  unfolding D-def using all-fun
  by (intro glue-subset-dep-pairsI) (auto elim!: mem-dep-functionE)
qed (fact F-right-unique)

```

```

lemma glue-upair-mem-dep-functionsI:

```

```

assumes f-dep-fun:  $f \in (x \in A) \rightarrow_s B\ x$ 
and g-dep-fun:  $g \in (x \in A') \rightarrow_s B\ x$ 
and agree-fg:  $\text{agree } (A \cap A') \{f, g\}$ 
shows  $\text{glue } \{f, g\} \in (x \in A \cup A') \rightarrow_s B\ x$ 
proof –
  have  $(\bigcup f \in \{f, g\}. \text{dom } f) = (\bigcup f \in \{f\}. \text{dom } f) \cup (\bigcup f \in \{g\}. \text{dom } f)$ 
    by (rule eqI) (auto simp only: idx-union-bin-union-dom-eq-bin-union-idx-union)
  also have  $\dots = \text{dom } f \cup \text{dom } g$  by (rule eqI) auto
  also have  $\dots = A \cup A'$  using assms by simp
  finally have  $A \cup A' = (\bigcup f \in \{f, g\}. \text{dom } f)$  by auto
  moreover have  $\text{set-right-unique-on } (A \cup A') (\text{glue } \{f, g\})$ 
proof (subst set-right-unique-on-set-iff-set-right-unique-on,
    rule set-right-unique-onI)
    fix  $x\ y\ y'$  assume  $x \in A \cup A'$ 
    and pairs-mem:  $\langle x, y \rangle \in \text{glue } \{f, g\}\ \langle x, y' \rangle \in \text{glue } \{f, g\}$ 
    show  $y = y'$ 
    proof (cases  $x \in A \cap A'$ )
      case True
        with agree-fg pairs-mem have  $\langle x, y \rangle \in f\ \langle x, y' \rangle \in f$ 
          by (auto dest: agreeD)
        with f-dep-fun show  $y = y'$  by (auto dest: set-right-unique-onD)
    qed (insert f-dep-fun g-dep-fun pairs-mem,
      auto elim!: mem-dep-functionsE dest: set-right-unique-onD)
    qed
  ultimately show ?thesis using assms by (auto intro: glue-mem-dep-functionsI)
qed

```

26.6 Restriction

lemma *restrict-left-mem-dep-functions-if-mem-dep-functions-if-agree*:

```

assumes agree A F
and  $f \in (x \in A) \rightarrow_s (B\ x)$ 
and  $f \in F$ 
and  $g \in F$ 
shows  $g \upharpoonright_A \in (x \in A) \rightarrow_s (B\ x)$ 
proof –
  from assms have  $g \upharpoonright_A = f \upharpoonright_A$ 
    by (auto elim: set-restrict-left-eq-set-restrict-left-if-agree)
  also have  $\dots = f$  using  $\langle f \in (x \in A) \rightarrow_s (B\ x) \rangle$  by auto
  finally show ?thesis using  $\langle f \in (x \in A) \rightarrow_s (B\ x) \rangle$  by simp
qed

```

lemma *restrict-left-mem-dep-functions-collectI*:

```

assumes  $f \in (x \in A) \rightarrow_s (B\ x)$ 
shows  $f \upharpoonright_P \in (x \in \{x \in A \mid P\ x\}) \rightarrow_s (B\ x)$ 
proof (rule mem-dep-functionsI)
  have  $\text{set-right-unique-on } A\ f = \text{set-right-unique-on } (\text{mem-of } A)\ f$  by simp
  also have  $\dots \leq \text{set-right-unique-on } (\text{mem-of } A \sqcap P)\ f$ 
    by (rule antimonoD[OF antimono-set-right-unique-on-pred]) auto

```

also have $\dots \leq \text{set-right-unique-on } (\text{mem-of } A \sqcap P) f \upharpoonright_P$
by $(\text{rule antimonod}[OF \text{ antimonono-set-right-unique-on-set}]) \text{ auto}$
also have $\dots = \text{set-right-unique-on } \{x \in A \mid P x\} f \upharpoonright_P$
unfolding inf-apply **by** simp
finally have $\text{set-right-unique-on } A f \leq \text{set-right-unique-on } \{x \in A \mid P x\} f \upharpoonright_P$.
moreover from assms **have** $\text{set-right-unique-on } A f$ **by** blast
ultimately show $\text{set-right-unique-on } \{x \in A \mid P x\} f \upharpoonright_P$ **by** auto
qed $(\text{insert assms, auto})$

end

27 Functions

theory $S\text{Functions}$
imports
 $S\text{Functions-Composition}$
 $S\text{Functions-Extend-Restrict}$
 $S\text{Functions-Lambda}$
begin

end

28 Set-Theoretic Orders

theory $S\text{Orders}$
imports
 $S\text{Binary-Relations-Antisymmetric}$
 $S\text{Binary-Relations-Connected}$
 $S\text{Binary-Relations-Reflexive}$
 $S\text{Binary-Relations-Transitive}$
begin

definition $\text{partial-order } D R \equiv$
 $\text{reflexive } D R \wedge \text{transitive } D R \wedge \text{antisymmetric } D R$

definition $\text{linear-order } D R \equiv \text{connected } D R \wedge \text{partial-order } D R$

definition $\text{well-founded } D R \equiv$
 $\forall X. X \subseteq D \wedge X \neq \{\} \longrightarrow (\exists a \in X. \forall x \in X. \langle x, a \rangle \in R \longrightarrow x = a)$

lemma well-foundedI :
assumes $\bigwedge X. [X \subseteq D; X \neq \{\}] \implies \exists a \in X. \forall x \in X. \langle x, a \rangle \in R \longrightarrow x = a$
shows $\text{well-founded } D R$
using assms **unfolding** well-founded-def **by** auto

definition $\text{well-order } D R \equiv \text{linear-order } D R \wedge \text{well-founded } D R$

end

29 Empty Set

theory *Empty-Set*
 imports *Equality*
begin

lemma *emptyE* [*elim*]: $x \in \{\}$ $\implies P$ **by** *auto*

lemma *eq-emptyI* [*intro*]: $\llbracket \bigwedge y. y \in A \implies \text{False} \rrbracket \implies A = \{\}$
 by *auto*

lemma *not-mem-if-empty* [*dest*]: $A = \{\} \implies a \notin A$
 by *auto*

lemma *ne-empty-if-mem*: $a \in A \implies A \neq \{\}$
 by *auto*

lemma *ex-mem-if-ne-empty*: $A \neq \{\} \implies \exists x. x \in A$
 by *auto*

lemma *ne-emptyE*:
 assumes $A \neq \{\}$
 obtains x **where** $x \in A$
 using *ex-mem-if-ne-empty* [*OF assms*]
 by *blast*

lemma *mem-trans-closed-empty* [*iff*]: *mem-trans-closed* $\{\}$
 unfolding *mem-trans-closed-def* **by** *blast*

end

30 Set Difference

theory *Set-Difference*
 imports *Union-Intersection*
begin

definition *diff* $A\ B \equiv \{x \in A \mid x \notin B\}$

bundle *hotg-diff-syntax* **begin** **notation** *diff* (*infixl* \setminus 65) **end**
bundle *no-hotg-diff-syntax* **begin** **no-notation** *diff* (*infixl* \setminus 65) **end**

unbundle *hotg-diff-syntax*

lemma *mem-diff-iff* [*iff*]: $a \in A \setminus B \longleftrightarrow (a \in A \wedge a \notin B)$
unfolding *diff-def* **by** *auto*

lemma *mem-if-mem-diff*: $a \in A \setminus B \implies a \in A$ **by** *simp*

lemma *not-mem-if-mem-diff*: $a \in A \setminus B \implies a \notin B$ **by** *simp*

lemma *diff-subset* [*iff*]: $A \setminus B \subseteq A$ **by** *blast*

lemma *subset-diff-if-inter-eq-empty-if-subset*:
 $C \subseteq A \implies C \cap B = \{\} \implies C \subseteq A \setminus B$
by *blast*

lemma *diff-self-eq* [*simp*]: $A \setminus A = \{\}$ **by** *blast*

lemma *diff-eq-left-if-inter-eq-empty*: $A \cap B = \{\} \implies A \setminus B = A$ **by** *auto*

lemma *empty-diff-eq* [*simp*]: $\{\} \setminus A = \{\}$ **by** *blast*

lemma *diff-empty-eq* [*simp*]: $A \setminus \{\} = A$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *diff-eq-empty-iff-subset*: $A \setminus B = \{\} \longleftrightarrow A \subseteq B$
unfolding *subset-def* **by** *auto*

lemma *inter-diff-eq-empty* [*simp*]: $A \cap (B \setminus A) = \{\}$ **by** *blast*

lemma *bin-union-diff-eq* [*simp*]: $A \cup (B \setminus A) = A \cup B$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *bin-union-diff-eq-if-subset*: $A \subseteq B \implies A \cup (B \setminus A) = B$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *subset-bin-union-diff*: $A \subseteq B \cup (A \setminus B)$
by *blast*

lemma *diff-diff-eq-if-subset-if-subset*: $A \subseteq B \implies B \subseteq C \implies B \setminus (C \setminus A) = A$
by *auto*

lemma *bin-union-diff-diff-eq* [*simp*]: $(A \cup B) \setminus (B \setminus A) = A$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *diff-bin-union-eq-bin-inter-diff*: $A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C)$
by (*rule eq-if-subset-if-subset*) *auto*

lemma *diff-bin-inter-eq-bin-union-diff*: $A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$
by (*rule eq-if-subset-if-subset*) *auto*

```

lemma bin-union-diff-eq-bin-union-diff:  $(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C)$ 
  by (rule eq-if-subset-if-subset) auto

lemma bin-union-diff-eq-diff-right [simp]:  $(A \cup B) \setminus B = A \setminus B$ 
  using bin-union-diff-eq-bin-union-diff by auto

lemma bin-union-diff-eq-diff-left [simp]:  $(B \cup A) \setminus B = A \setminus B$ 
  using bin-union-diff-eq-bin-union-diff by auto

lemma bin-inter-diff-eq-bin-inter-diff:  $(A \cap B) \setminus C = A \cap (B \setminus C)$ 
  by (rule eq-if-subset-if-subset) auto

lemma diff-bin-inter-eq-diff-if-subset:  $C \subseteq A \implies ((A \setminus B) \cap C) = (C \setminus B)$ 
  by auto

lemma diff-bin-inter-distrib-right:  $C \cap (A \setminus B) = (C \cap A) \setminus (C \cap B)$ 
  by (rule eq-if-subset-if-subset) auto

lemma diff-bin-inter-distrib-left:  $(A \setminus B) \cap C = (A \cap C) \setminus (B \cap C)$ 
  by (rule eq-if-subset-if-subset) auto

lemma diff-idx-union-eq-idx-union:
  assumes  $I \neq \{\}$ 
  shows  $B \setminus (\bigcup_{i \in I}. A \ i) = (\bigcap_{i \in I}. B \setminus A \ i)$ 
  using assms by (intro eq-if-subset-if-subset) auto

lemma diff-idx-inter-eq-idx-inter:
  assumes  $I \neq \{\}$ 
  shows  $B \setminus (\bigcap_{i \in I}. A \ i) = (\bigcup_{i \in I}. B \setminus A \ i)$ 
  using assms by (intro eq-if-subset-if-subset) auto

lemma collect-diff:  $\{x \in (A \setminus B) \mid P \ x\} = \{x \in A \mid P \ x\} \setminus \{x \in B \mid P \ x\}$ 
  by (rule eq-if-subset-if-subset) auto

lemma mono-diff-left: mono  $(\lambda A. A \setminus B)$ 
  by (intro monoI) auto

lemma antimono-diff-right: antimono  $(\lambda B. A \setminus B)$ 
  by (intro antimonoI) auto

end

```

31 Universes

```

theory Universes
  imports

```

```

    Coproduct
    SFunctions
begin

abbreviation V :: set where V  $\equiv$  univ {}

lemma
  assumes ZF-closed U
  and X  $\in$  U
  shows ZF-closed-union [elim!]:  $\bigcup X \in U$ 
  and ZF-closed-powerset [elim!]: powerset X  $\in$  U
  and ZF-closed-repl:
    ( $\bigwedge x. x \in X \implies f x \in U$ )  $\implies$  {f x | x  $\in$  X}  $\in$  U
  using assms by (auto simp: ZF-closed-def)

lemma
  assumes A  $\in$  univ X
  shows univ-closed-union [intro!]:  $\bigcup A \in \text{univ } X$ 
  and univ-closed-powerset [intro!]: powerset A  $\in$  univ X
  and univ-closed-repl [intro]:
    ( $\bigwedge x. x \in A \implies f x \in \text{univ } X$ )  $\implies$  {f x | x  $\in$  A}  $\in$  univ X
  using ZF-closed-univ[of X]
  by (auto simp only: assms ZF-closed-repl)

  Variations on transitivity:

lemma mem-univ-if-mem-if-mem-univ: A  $\in$  univ X  $\implies$  x  $\in$  A  $\implies$  x  $\in$  univ X
  using mem-trans-closed-univ by blast

lemma mem-univ-if-mem: x  $\in$  X  $\implies$  x  $\in$  univ X
  by (rule mem-univ-if-mem-if-mem-univ) auto

lemma subset-univ-if-mem: A  $\in$  univ X  $\implies$  A  $\subseteq$  univ X
  using mem-univ-if-mem-if-mem-univ by auto

lemma empty-mem-univ [iff]: {}  $\in$  univ X
proof -
  have X  $\in$  univ X by (fact mem-univ)
  then have powerset X  $\subseteq$  univ X by (intro subset-univ-if-mem) blast
  then show {}  $\in$  univ X by auto
qed

lemma subset-univ [iff]: A  $\subseteq$  univ A
  by (auto intro: mem-univ-if-mem-if-mem-univ)

lemma univ-closed-upair [intro!]:
  [ $x \in \text{univ } X$ ; y  $\in$  univ X]  $\implies$  upair x y  $\in$  univ X
  unfolding upair-def
  by (intro univ-closed-repl, intro univ-closed-powerset) auto

```


lemma *univ-closed-insert* [intro!]:
 $x \in \text{univ } X \implies A \in \text{univ } X \implies \text{insert } x \ A \in \text{univ } X$
unfolding *insert-def* **using** *univ-closed-upair* **by** *blast*

lemma *univ-closed-pair* [intro!]:
 $\llbracket x \in \text{univ } X; y \in \text{univ } X \rrbracket \implies \langle x, y \rangle \in \text{univ } X$
unfolding *pair-def* **by** *auto*

lemma *univ-closed-extend* [intro!]:
 $x \in \text{univ } X \implies y \in \text{univ } X \implies A \in \text{univ } X \implies \text{extend } x \ y \ A \in \text{univ } X$
by (*subst insert-pair-eq-extend[symmetric]*) *auto*

lemma *univ-closed-bin-union* [intro!]:
 $\llbracket x \in \text{univ } X; y \in \text{univ } X \rrbracket \implies x \cup y \in \text{univ } X$
unfolding *bin-union-def* **by** *auto*

lemma *univ-closed-singleton* [intro!]: $x \in \text{univ } U \implies \{x\} \in \text{univ } U$
by *auto*

lemma *bin-union-univ-eq-univ-if-mem*: $A \in \text{univ } U \implies A \cup \text{univ } U = \text{univ } U$
by (*rule eq-if-subset-if-subset*) (*auto intro: mem-univ-if-mem-if-mem-univ*)

lemma *univ-closed-dep-pairs* [intro!]:
assumes *A-mem-univ*: $A \in \text{univ } U$
and *univ-B-closed*: $\bigwedge x. x \in A \implies B \ x \in \text{univ } U$
shows $\sum x \in A. (B \ x) \in \text{univ } U$
unfolding *dep-pairs-def* **using** *assms*
by (*intro univ-closed-union ZF-closed-repl*) (*auto intro: mem-univ-if-mem-if-mem-univ*)

lemma *subset-univ-if-subset-univ-pairs*: $X \subseteq \text{univ } A \times \text{univ } A \implies X \subseteq \text{univ } A$
by *auto*

lemma *univ-closed-pairs* [intro!]: $X \subseteq \text{univ } A \implies Y \subseteq \text{univ } A \implies X \times Y \subseteq \text{univ } A$
by *auto*

lemma *univ-closed-dep-functions* [intro!]:
assumes $A \in \text{univ } U$
and $\bigwedge x. x \in A \implies B \ x \in \text{univ } U$
shows $((x \in A) \rightarrow_s (B \ x)) \in \text{univ } U$
proof –
let $?P = \text{powerset } (\sum x \in A. B \ x)$
have $((x \in A) \rightarrow_s (B \ x)) \subseteq ?P$ **by** *auto*
moreover **have** $?P \in \text{univ } U$ **using** *assms* **by** *auto*
ultimately show *?thesis* **by** (*auto intro: mem-univ-if-mem-if-mem-univ*)
qed

lemma *univ-closed-inl* [intro!]: $x \in \text{univ } A \implies \text{inl } x \in \text{univ } A$
unfolding *inl-def* **by** *auto*

lemma *univ-closed-inr* [*intro!*]: $x \in \text{univ } A \implies \text{inr } x \in \text{univ } A$
unfolding *inr-def* **by** *auto*

end

References

- [1] C. E. Brown, C. Kaliszyk, and K. Pak. Higher-Order Tarski Grothendieck as a Foundation for Formal Proof. In J. Harrison, J. O’Leary, and A. Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:16, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [2] L. Kirby. Addition and multiplication of sets. *Mathematical Logic Quarterly*, 53(1):52–65, 2007.
- [3] L. C. Paulson. Zermelo fraenkel set theory in higher-order logic. *Archive of Formal Proofs*, October 2019. https://isa-afp.org/entries/ZFC_in_HOL.html, Formal proof development.