TLA$^+$

Stephan Merz

October 15, 2011

Contents

1 First-order logic for TLA+

theory *PredicateLogic*
imports *Pure*
uses
~~/src/Tools/misc-legacy.ML
~~/src/Tools/intuitionistic.ML
~~/src/Provers/splitter.ML
~~/src/Provers/hypsubst.ML
~~/src/Tools/atomize-elim.ML
~~/src/Provers/classical.ML
~~/src/Provers/blast.ML
~~/src/Provers/quantifier1.ML
~~/src/Provers/clasimp.ML
~~/src/Tools/IsaPlanner/isand.ML ~~/src/Tools/IsaPlanner/rw-tools.ML ~~/src/Tools/IsaPlanner/rw-i
~~/src/Tools/IsaPlanner/zipper.ML ~~/src/Tools/eqsubst.ML
~~/src/Tools/induct.ML
(*simplifier-setup.ML*)
begin

declare [[*eta-contract* = *false*]]

We define classical first-order logic as a basis for an encoding of TLA$^+$. TLA$^+$ is untyped, to the extent that it does not even distinguish between terms and formulas. We therefore declare a single type c that represents the universe of "constants" rather than introducing the traditional types i and o of first-order logic that, for example, underly Isabelle/ZF.

setup *Pure-Thy.old-appl-syntax-setup*

setup ⟪ *Intuitionistic.method-setup* @{*binding iprover*} ⟫

typedecl c

The following (implicit) lifting from the object to the Isabelle meta level is always needed when formalizing a logic. It corresponds to judgments $\vdash F$ or $\models F$ in standard presentations, asserting that a formula is considered true (either because it is an assumption or because it is a theorem).

judgment
Trueprop :: $c \Rightarrow$ *prop* ((-) *5*)

1.1 Equality

The axioms for equality are reflexivity and a rule that asserts that equal terms are interchangeable at the meta level (this is essential for setting up Isabelle's rewriting machinery). In particular, we can derive a substitution rule.

axiomatization
  eq :: $c \Rightarrow c \Rightarrow c$ (**infixl** = *50*)
where
  refl [*intro*!]: $a = a$
and
  eq-reflection: $t = u \Longrightarrow t \equiv u$

Left and right hand sides of definitions are equal. This is the converse of axiom $?t = ?u \Longrightarrow ?t \equiv ?u$.

theorem *meta-to-obj-eq*:
  assumes *df*: $t \equiv u$
  shows $t = u$
by (*unfold df*, *rule refl*)

theorem *subst*:
  assumes *eq*: $a = b$ **and** *p*: $P(a)$
  shows $P(b)$
proof −
  from *eq* **have** *ab*: $a \equiv b$
    by (*rule eq-reflection*)
  from *p* **show** $P(b)$
    by (*unfold ab*)
qed

theorem *sym* [*sym*]:
  $a = b \Longrightarrow b = a$
proof (*elim subst*)
  show $a{=}a$ **..**
qed

theorem *trans* [*trans*]:
  $a{=}b \Longrightarrow b{=}c \Longrightarrow a{=}c$
by (*rule subst*)

theorems *ssubst* = *sym*[*THEN subst*]

LET-expressions in TLA$^+$ expressions.

Limitation: bindings cannot be unfolded selectively. Rewrite with *Let-def* in order to expand *all* bindings within an expression or a context.

definition
  Let :: $'b \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a$
where

Let(*b*,*e*) ≡ *e*(*b*)

nonterminal
letbinds **and** *letbind*

syntax
-bind :: [*pttrn*, *'a*] => *letbind* ((*2*- ==/ -) *10*)
:: *letbind* => *letbinds* (-)
-binds :: [*letbind*, *letbinds*] => *letbinds* (-/ -)
-Let :: [*letbinds*, *'a*] => *'a* ((*LET* (-)/ *IN* (-)) *10*)

syntax (*xsymbols*)
-bind :: [*pttrn*, *'a*] => *letbind* ((*2*- ≡/ -) *10*)
syntax (*xsymbols*)
-bind :: [*pttrn*, *'a*] => *letbind* ((*2*- ≜/ -) *10*)

translations
-Let(*-binds*(*b*, *bs*), *e*) ⇌ *-Let*(*b*, *-Let*(*bs*, *e*))
LET x ≜ *a IN e* ⇌ *CONST Let*(*a*, (%*x*. *e*))

1.2 Propositional logic

Propositional logic is introduced in a rather non-standard way by declaring constants *TRUE* and *FALSE* as well as conditional expressions. The Boolean connectives are defined such that they always return either *TRUE* or *FALSE*, irrespectively of their arguments. This allows us to prove many equational laws of propositional logic, which is useful for automatic reasoning based on rewriting. Note that we have equivalence as well as equality. The two relations agree over Boolean values, but equivalence may be stricter weaker than equality over non-Booleans.

consts
TRUE :: *c*
FALSE :: *c*
cond :: *c* ⇒ *c* ⇒ *c* ⇒ *c* ((*IF* (-)/ *THEN* (-)/ *ELSE* (-)) *10*)

consts
Not :: *c* ⇒ *c* (~ - [*40*] *40*)
conj :: *c* ⇒ *c* ⇒ *c* (**infixr** & *35*)
disj :: *c* ⇒ *c* ⇒ *c* (**infixr** | *30*)
imp :: *c* ⇒ *c* ⇒ *c* (**infixr** => *25*)
iff :: *c* ⇒ *c* ⇒ *c* (**infixr** <=> *25*)

abbreviation *not-equal* :: *c* ⇒ *c* ⇒ *c* (**infixl** ~= *50*)
where *x* ~= *y* ≡ ~ (*x* = *y*)

notation (*xsymbols*)
Not (¬ - [*40*] *40*)
and *conj* (**infixr** ∧ *35*)

and *disj* (**infixr** ∨ *30*)
and *imp* (**infixr** ⇒ *25*)
and *iff* (**infixr** ⇔ *25*)
and *not-equal* (**infix** ≠ *50*)

notation (*HTML* **output**)
Not (¬ - [*40*] *40*)
and *conj* (**infixr** ∧ *35*)
and *disj* (**infixr** ∨ *30*)
and *imp* (**infixr** ⇒ *25*)
and *iff* (**infixr** ⇔ *25*)
and *not-equal* (**infix** ≠ *50*)

defs
not-def: ¬ *A* ≡ *IF A THEN FALSE ELSE TRUE*
conj-def: *A* ∧ *B* ≡ *IF A THEN* (*IF B THEN TRUE ELSE FALSE*) *ELSE FALSE*
disj-def: *A* ∨ *B* ≡ *IF A THEN TRUE ELSE IF B THEN TRUE ELSE FALSE*
imp-def: *A* ⇒ *B* ≡ *IF A THEN* (*IF B THEN TRUE ELSE FALSE*) *ELSE TRUE*
iff-def: *A* ⇔ *B* ≡ (*A* ⇒ *B*) ∧ (*B* ⇒ *A*)

We adopt the following axioms of propositional logic:

1. *A* is a theorem if and only if it equals *TRUE*.
2. *FALSE* (more precisely, ¬ *TRUE*) implies anything.
3. Conditionals are reasoned about by case distinction.

We also assert that the equality predicate returns either *TRUE* or *FALSE*.

axiomatization where
trueI [*intro*!]: *TRUE*
and
eqTrueI: *A* ⟹ *A* = *TRUE*
and
notTrueE [*elim*!]: ¬ *TRUE* ⟹ *A*
and
condI: ⟦ *A* ⟹ *P*(*t*); ¬ *A* ⟹ *P*(*e*) ⟧ ⟹ *P*(*IF A THEN t ELSE e*)
and
eqBoolean : *x* ≠ *y* ⟹ (*x*=*y*) = *FALSE*

We now derive the standard proof rules of propositional logic. The first lemmas are about *TRUE*, *FALSE*, and conditional expressions.

lemma *eqTrueD*: — converse of eqTrueI
assumes *a*: *A* = *TRUE*
shows *A*
by (*unfold a*[*THEN eq-reflection*], *rule trueI*)

Assumption *TRUE* is useless and can be deleted.

```
lemma TrueAssumption: (TRUE ⟹ PROP P) == PROP P
proof
  assume h: TRUE ⟹ PROP P show PROP P
    by (rule h, rule trueI)
next
  assume PROP P thus PROP P .
qed

lemma condT: (IF TRUE THEN t ELSE e) = t
proof (rule condI)
  show t = t ..
next
  assume ¬ TRUE thus e = t ..
qed

lemma notTrue: (¬ TRUE) = FALSE
by (unfold not-def, rule condT)

theorem falseE [elim!]:
  assumes f: FALSE
  shows A
proof (rule notTrueE)
  have FALSE = (¬ TRUE)
    by (rule sym[OF notTrue])
  hence r: FALSE ≡ ¬ TRUE
    by (rule eq-reflection)
  from f show ¬ TRUE
    by (unfold r)
qed

lemma condF: (IF FALSE THEN t ELSE e) = e
proof (rule condI)
  assume FALSE thus t = e ..
next
  show e = e ..
qed

lemma notFalse: (¬ FALSE) = TRUE
by (unfold not-def, rule condF)

lemma condThen:
  assumes a: A
  shows (IF A THEN t ELSE e) = t
proof −
  from a have A = TRUE
    by (rule eqTrueI)
  hence r: A ≡ TRUE
    by (rule eq-reflection)
  show ?thesis
```

by (*unfold r*, *rule condT*)
qed

lemma *condD1* [*elim 2*]:
assumes *c*: *IF A THEN P ELSE Q* (**is** *?if*) **and** *a*: *A*
shows *P*
proof (*rule eqTrueD*)
from *c* **have** *?if* = *TRUE* **by** (*rule eqTrueI*)
hence *TRUE* = *?if* **by** (*rule sym*)
also from *a* **have** *?if* = *P* **by** (*rule condThen*)
finally show *P* = *TRUE* **by** (*rule sym*)
qed

lemma *condElse*:
assumes *na*: ¬ *A*
shows (*IF A THEN t ELSE e*) = *e*
proof (*rule condI*)
assume *A* **hence** *A* = *TRUE*
by (*rule eqTrueI*)
hence *r*: *A* ≡ *TRUE*
by (*rule eq-reflection*)
from *na* **have** ¬ *TRUE*
by (*unfold r*)
thus *t* = *e* **..**
next
show *e* = *e* **..**
qed

lemma *condD2* [*elim 2*]:
assumes *c*: *IF A THEN P ELSE Q* (**is** *?if*) **and** *a*: ¬ *A*
shows *Q*
proof (*rule eqTrueD*)
from *c* **have** *?if* = *TRUE* **by** (*rule eqTrueI*)
hence *TRUE* = *?if* **by** (*rule sym*)
also from *a* **have** *?if* = *Q* **by** (*rule condElse*)
finally show *Q* = *TRUE* **by** (*rule sym*)
qed

The following theorem shows that we have a classical logic.

lemma *cond-id*: (*IF A THEN t ELSE t*) = *t*
proof (*rule condI*)
show *t*=*t* **..**
show *t*=*t* **..**
qed

theorem *case-split*:
assumes *p*: *P* ⟹ *Q*
and *np*: ¬ *P* ⟹ *Q*
shows *Q*

proof −
  from *p np* **have** *IF P THEN Q ELSE Q* **by** (*rule condI*)
  thus *Q* **by** (*unfold eq-reflection*[*OF cond-id*])
qed

theorem *condE*:
  — use conditionals in hypotheses
  assumes *p*: $P(IF\ A\ THEN\ t\ ELSE\ e)$
  and *pos*: $[\![\ A;\ P(t)\ ]\!] \Longrightarrow B$
  and *neg*: $[\![\ \neg\ A;\ P(e)\ ]\!] \Longrightarrow B$
  shows *B*
proof (*rule case-split*[*of A*])
  assume *a*: *A*
  hence *r*: $IF\ A\ THEN\ t\ ELSE\ e \equiv t$
    by (*unfold eq-reflection*[*OF condThen*])
  with *p* **have** $P(t)$
    by (*unfold r*)
  with *a* **show** *B*
    by (*rule pos*)
next
  assume *a*: ¬ *A*
  hence *r*: $IF\ A\ THEN\ t\ ELSE\ e \equiv e$
    by (*unfold eq-reflection*[*OF condElse*])
  with *p* **have** $P(e)$
    by (*unfold r*)
  with *a* **show** *B*
    by (*rule neg*)
qed

Theorems *condI* and *condE* require higher-order unification and are therefore unsuitable for automatic tactics (in particular the `blast` method). We now derive some special cases that can be given to these methods.

— $[\![A \Longrightarrow t;\ \neg A \Longrightarrow e]\!] \Longrightarrow IF\ A\ THEN\ t\ ELSE\ e$
lemmas *cond-boolI* [*intro*!] = *condI*[**where** $P{=}\lambda\ Q.\ Q$]

lemma *cond-eqLI* [*intro*!]:
  assumes *1*: $A \Longrightarrow t = v$ **and** *2*: $\neg A \Longrightarrow u = v$
  shows $(IF\ A\ THEN\ t\ ELSE\ u) = v$
proof (*rule condI*)
  show $A \Longrightarrow t{=}v$ **by** (*rule 1*)
next
  show $\neg A \Longrightarrow u{=}v$ **by** (*rule 2*)
qed

lemma *cond-eqRI* [*intro*!]:
  assumes *1*: $A \Longrightarrow v = t$ **and** *2*: $\neg A \Longrightarrow v = u$
  shows $v = (IF\ A\ THEN\ t\ ELSE\ u)$
proof (*rule condI*)
  show $A \Longrightarrow v = t$ **by** (*rule 1*)

next
  show $\neg A \Longrightarrow v = u$ **by** (*rule 2*)
qed

— $[\![$*IF A THEN t ELSE e*; $[\![A;t]\!] \Longrightarrow B$; $[\![\neg A;e]\!] \Longrightarrow B]\!] \Longrightarrow B$
lemmas *cond-boolE* [*elim*!] = *condE*[**where** $P=\lambda\ Q.\ Q$]

lemma *cond-eqLE* [*elim*!]:
  assumes *maj*: (*IF A THEN t ELSE e*) = *u*
  and *1*: $[\![\ A;\ t=u\ ]\!] \Longrightarrow B$ **and** *2*: $[\![\ \neg A;\ e=u\ ]\!] \Longrightarrow B$
  shows *B*
using *maj*
proof (*rule condE*)
  show $[\![\ A;\ t=u\ ]\!] \Longrightarrow B$ **by** (*rule 1*)
next
  show $[\![\ \neg A;\ e=u\ ]\!] \Longrightarrow B$ **by** (*rule 2*)
qed

lemma *cond-eqRE* [*elim*!]:
  assumes *maj*: *u* = (*IF A THEN t ELSE e*)
  and *1*: $[\![\ A;\ u=t\ ]\!] \Longrightarrow B$ **and** *2*: $[\![\ \neg A;\ u=e\ ]\!] \Longrightarrow B$
  shows *B*
using *maj*
proof (*rule condE*)
  show $[\![\ A;\ u=t\ ]\!] \Longrightarrow B$ **by** (*rule 1*)
next
  show $[\![\ \neg A;\ u=e\ ]\!] \Longrightarrow B$ **by** (*rule 2*)
qed

Derive standard propositional proof rules, based on the operator definitions in terms of *IF - THEN - ELSE -*.

theorem *notI* [*intro*!]:
  assumes *hyp*: $A \Longrightarrow$ *FALSE*
  shows $\neg$ *A*
proof (*unfold not-def*, *rule condI*)
  assume *A* **thus** *FALSE*
    by (*rule hyp*)
next
  show *TRUE* **..**
qed

lemma *false-neq-true*: *FALSE* $\neq$ *TRUE*
proof
  assume *FALSE* = *TRUE*
  thus *FALSE* **by** (*rule eqTrueD*)
qed

lemma *false-eq-trueE*:
  assumes *ft*: *FALSE* = *TRUE*

shows B
proof ($rule\ falseE$)
from ft **show** $FALSE$
by ($rule\ eqTrueD$)
qed

lemmas $true\text{-}eq\text{-}falseE = sym[THEN\ false\text{-}eq\text{-}trueE]$

lemma $notFalseI$: $\neg\ FALSE$
by $iprover$

lemma $A\text{-}then\text{-}notA\text{-}false$:
assumes a: A
shows $(\neg\ A) = FALSE$
using a
by ($unfold\ not\text{-}def$, $rule\ condThen$)

The following is an alternative introduction rule for negation that is useful when we know that A is Boolean.

lemma $eq\text{-}false\text{-}not$:
assumes a: $A = FALSE$
shows $\neg\ A$
proof ($rule\ eqTrueD$)
show $(\neg\ A) = TRUE$ **by** ($unfold\ eq\text{-}reflection[OF\ a]$, $rule\ notFalse$)
qed

Note that we do not have $\neg\ A \Longrightarrow A = FALSE$: this is true only for Booleans, not for arbitrary values. However, we have the following theorem, which is just the ordinary elimination rule for negation.

theorem $notE$:
assumes $notA$: $\neg\ A$ **and** a: A
shows B
proof ($rule\ false\text{-}eq\text{-}trueE$)
from a **have** $(\neg\ A) = FALSE$
by ($rule\ A\text{-}then\text{-}notA\text{-}false$)
hence $FALSE = (\neg\ A)$
by ($rule\ sym$)
also from $notA$ **have** $(\neg\ A) = TRUE$
by ($rule\ eqTrueI$)
finally show $FALSE = TRUE$ **.**
qed

theorem $notE'$ [$elim\ 2$]:
assumes $notA$: $\neg\ A$ **and** r: $\neg\ A \Longrightarrow A$
shows B
using $notA$
proof ($rule\ notE$)
from $notA$ **show** A **by** ($rule\ r$)
qed

lemma *notnotI*:
  assumes *a*: A
  shows $\neg\neg\ A$
proof
  assume $\neg\ A$
  from *this a* **show** *FALSE* **..**
qed

theorem *not-sym* [*sym*]:
  assumes *hyp*: $a \neq b$
  shows $b \neq a$
proof
  assume $b = a$
  hence $a = b$ **..**
  with *hyp* **show** *FALSE* **..**
qed

Some derived proof rules of classical logic.

theorem *contradiction*:
  assumes *hyp*: $\neg\ A \Longrightarrow FALSE$
  shows A
proof (*rule case-split*[*of A*])
  assume $\neg\ A$ **hence** *FALSE*
    by (*rule hyp*)
  thus A **..**
qed — the other case is trivial

theorem *classical*:
  assumes *c*: $\neg\ A \Longrightarrow A$
  shows A
proof (*rule contradiction*)
  assume *na*: $\neg\ A$ **hence** A **by** (*rule c*)
  with *na* **show** *FALSE* **..**
qed

theorem *swap*:
  assumes *a*: $\neg\ A$ **and** *r*: $\neg\ B \Longrightarrow A$
  shows B
proof (*rule contradiction*)
  assume $\neg\ B$
  with *r* **have** A **.**
  with *a* **show** *FALSE* **..**
qed

theorem *notnotD* [*dest*]:
  assumes *nn*: $\neg\neg\ A$ **shows** A
proof (*rule contradiction*)
  assume $\neg\ A$

with *nn* **show** *FALSE* **..**
qed

Note again: A and $\neg\neg\ A$ are inter-derivable (and hence equivalent), but not equal!

lemma *contrapos*:
  assumes *b*: $\neg\ B$ **and** *ab*: $A \Longrightarrow B$
  shows $\neg\ A$
proof
  assume A
  hence B **by** (*rule ab*)
  with *b* **show** *FALSE* **..**
qed

lemma *contrapos2*:
  assumes *b*: B **and** *ab*: $\neg A \Longrightarrow \neg B$
  shows A
proof $-$
  have $\neg\neg\ A$
  proof
    assume $\neg\ A$
    hence $\neg\ B$ **by** (*rule ab*)
    from *this b* **show** *FALSE* **..**
  qed
  thus A **..**
qed

theorem *conjI* [*intro*!]:
  assumes *a*: A **and** *b*: B
  shows $A \wedge B$
proof (*rule eqTrueD*)
  from *a* **have** $(A \wedge B) = (\mathit{IF}\ B\ \mathit{THEN}\ \mathit{TRUE}\ \mathit{ELSE}\ \mathit{FALSE})$
    by (*unfold conj-def*, *rule condThen*)
  also from *b* **have** $\ldots = \mathit{TRUE}$ **by** (*rule condThen*)
  finally show $(A \wedge B) = \mathit{TRUE}$ **.**
qed

theorem *conjD1*:
  assumes *ab*: $A \wedge B$
  shows A
proof (*rule contradiction*)
  assume $\neg\ A$
  with *ab* **show** *FALSE*
    by (*unfold conj-def*, *elim condD2*)
qed

theorem *conjD2*:
  assumes *ab*: $A \wedge B$
  shows B

proof (*rule contradiction*)
  assume *b*: ¬ *B*
  from *ab* **have** *A* **by** (*rule conjD1*)
  with *ab* **have** *IF B THEN TRUE ELSE FALSE*
    by (*unfold conj-def*, *elim condD1*)
  with *b* **show** *FALSE* **by** (*elim condD2*)
qed

theorem *conjE* [*elim*!]:
  assumes *ab*: $A \wedge B$ **and** *c*: $A \Longrightarrow B \Longrightarrow C$
  shows *C*
proof (*rule c*)
  from *ab* **show** *A* **by** (*rule conjD1*)
  from *ab* **show** *B* **by** (*rule conjD2*)
qed

Disjunction

theorem *disjI1* [*elim*]:
  assumes *a*: *A*
  shows $A \vee B$
proof (*unfold disj-def*, *rule*)
  show *TRUE* **..**
next
  assume ¬ *A*
  from *this a* **show** *IF B THEN TRUE ELSE FALSE* **..**
qed

theorem *disjI2* [*elim*]:
  assumes *b*: *B*
  shows $A \vee B$
proof (*unfold disj-def*, *rule*)
  show *TRUE* **..**
next
  show *IF B THEN TRUE ELSE FALSE*
  proof
    show *TRUE* **..**
  next
    assume ¬ *B*
    from *this b* **show** *FALSE* **..**
  qed
qed

theorem *disjI* [*intro*!]: — classical introduction rule
  assumes *ab*: $\neg A \Longrightarrow B$
  shows $A \vee B$
proof (*unfold disj-def*, *rule*)
  show *TRUE* **..**
next
  assume ¬ *A*

```
  hence b: B by (rule ab)
  show IF B THEN TRUE ELSE FALSE
  proof
    show TRUE ..
  next
    assume ¬ B
    from this b show FALSE..
  qed
qed

theorem disjE [elim!]:
  assumes ab: A ∨ B and ac: A ⟹ C and bc: B ⟹ C
  shows C
proof (rule case-split[where P=A])
  assume A thus C by (rule ac)
next
  assume nota: ¬ A
  have B
  proof (rule contradiction)
    assume notb: ¬ B
    from nota have IF B THEN TRUE ELSE FALSE
      by (rule ab[unfolded disj-def, THEN condD2])
    from this notb show FALSE by (rule condD2)
  qed
  thus C by (rule bc)
qed

theorem excluded-middle: A ∨ ¬ A
proof
  assume ¬ A thus ¬ A .
qed
```

Implication

```
theorem impI [intro!]:
  assumes ab: A ⟹ B
  shows A ⇒ B
proof (unfold imp-def, rule)
  assume A
  hence b: B by (rule ab)
  show IF B THEN TRUE ELSE FALSE
  proof
    show TRUE ..
  next
    assume ¬ B
    from this b show FALSE ..
  qed
next
  show TRUE ..
qed
```

theorem *mp* :
  assumes *ab*: $A \Rightarrow B$ **and** *a*: A
  shows B
proof (*rule contradiction*)
  assume *notb*: $\neg B$
  from *a* **have** *IF B THEN TRUE ELSE FALSE*
    by (*rule ab*[*unfolded imp-def*, *THEN condD1*])
  from *this notb* **show** *FALSE* **by** (*rule condD2*)
qed

theorem *rev-mp* :
  assumes *a*: A **and** *ab*: $A \Rightarrow B$
  shows B
using *ab a* **by** (*rule mp*)

theorem *impE*:
  assumes *ab*: $A \Rightarrow B$ **and** *a*: A **and** *bc*: $B \Longrightarrow C$
  shows C
proof −
  from *ab a* **have** B **by** (*rule mp*)
  thus C **by** (*rule bc*)
qed

theorem *impCE* [*elim*]:
  assumes *ab*: $A \Rightarrow B$ **and** *b*: $B \Longrightarrow P$ **and** *a*: $\neg A \Longrightarrow P$
  shows P
proof (*rule classical*)
  assume *contra*: $\neg P$
  have A
  proof (*rule contradiction*)
    assume $\neg A$ **hence** P **by** (*rule a*)
    with *contra* **show** *FALSE* **..**
  qed
  with *ab* **have** B **by** (*rule mp*)
  thus P **by** (*rule b*)
qed

theorem *impCE′*:
  assumes *ab*: $A \Rightarrow B$ **and** *a*: $\neg C \Longrightarrow A$ **and** *b*: $B \Longrightarrow C$
  shows C
proof (*rule classical*)
  assume $\neg C$
  hence A **by** (*rule a*)
  with *ab* **have** B **by** (*rule mp*)
  thus C **by** (*rule b*)
qed

Equivalence

theorem *iffI* [*intro*!]:
  assumes *ab*: $A \Longrightarrow B$ **and** *ba*: $B \Longrightarrow A$
  shows $A \Leftrightarrow B$
proof (*unfold iff-def*, *rule*)
  from *ab* **show** $A \Rightarrow B$ **..**
  from *ba* **show** $B \Rightarrow A$ **..**
qed

lemma *iff-refl*: $A \Leftrightarrow A$
by *iprover*

lemma *meta-eq-to-iff*:
  assumes *mt*: $A \equiv B$ **shows** $A \Leftrightarrow B$
by (*unfold mt*, *rule iff-refl*)

lemma *eqThenIff*:
  assumes *eq*: $A = B$ **shows** $A \Leftrightarrow B$
proof −
  from *eq* **have** $A \equiv B$ **by** (*rule eq-reflection*)
  thus *?thesis* **by** (*rule meta-eq-to-iff*)
qed

theorem *iffD1* [*elim 2*]:
  assumes *ab*: $A \Leftrightarrow B$ **and** *a*: A
  shows B
using *ab*
proof (*unfold iff-def*, *elim conjE*)
  assume $A \Rightarrow B$
  from *this a* **show** B **by** (*rule mp*)
qed

theorem *iffD2* [*elim 2*]:
  assumes *ab*: $A \Leftrightarrow B$ **and** *b*: B
  shows A
using *ab*
proof (*unfold iff-def*, *elim conjE*)
  assume $B \Rightarrow A$
  from *this b* **show** A **by** (*rule mp*)
qed

theorem *iffE*:
  assumes *ab*: $A \Leftrightarrow B$ **and** *r*: $[\![A \Rightarrow B; B \Rightarrow A]\!] \Longrightarrow C$
  shows C
proof (*rule r*)
  from *ab* **show** $A \Rightarrow B$ **by** (*unfold iff-def*, *elim conjE*)
  from *ab* **show** $B \Rightarrow A$ **by** (*unfold iff-def*, *elim conjE*)
qed

theorem *iffCE* [*elim*!]:

```
  assumes ab: A ⇔ B
  and pos: ⟦A; B⟧ ⟹ C and neg: ⟦¬ A; ¬ B⟧ ⟹ C
  shows C
proof (rule case-split[of A])
  assume a: A
  with ab have B ..
  with a show C by (rule pos)
next
  assume a: ¬ A
  have ¬ B
  proof
    assume B
    with ab have A ..
    with a show FALSE ..
  qed
  with a show C by (rule neg)
qed

theorem iff-trans [trans]:
  assumes ab: A ⇔ B and bc: B ⇔ C
  shows A ⇔ C
proof
  assume A
  with ab have B ..
  with bc show C ..
next
  assume C
  with bc have B ..
  with ab show A ..
qed
```

1.3 Predicate Logic

We take Hilbert's ε as the basic binder and define the other quantifiers as derived connectives. Again, we make sure that quantified formulas evaluate to *TRUE* or *FALSE*.

Observe that quantification is allowed at arbitrary types. Although TLA$^+$ formulas are purely first-order formulas, and may only contain quantification over values of type c, we sometimes need to reason about formula schemas, for example for induction, and automatic provers such as `blast` rely on reflection to the object level for reasoning about meta-connectives, which would not be possible with purely first-order quantification.

```
consts
  Choice  :: ('a ⇒ c) ⇒ 'a
  Ex      :: ('a ⇒ c) ⇒ c
  All     :: ('a ⇒ c) ⇒ c
```

Concrete syntax: several variables as in $\forall\, x,y : P(x,y)$.

nonterminal *cidts*

syntax
:: *idt* ⇒ *cidts* (- [*100*] *100*)
@*cidts* :: [*idt*, *cidts*] ⇒ *cidts* (-,/ - [*100*,*100*] *100*)

syntax

@*Choice* :: [*idt*, *c*] ⇒ *c* ((*3CHOOSE* - :/ -) [*100*,*10*] *10*)
@*Ex* :: [*cidts*, *c*] ⇒ *c* ((*3**E* - :/ -) [*100*,*10*] *10*)
@*All* :: [*cidts*, *c*] ⇒ *c* ((*3**A* - :/ -) [*100*,*10*] *10*)
syntax (*xsymbols*)
@*Ex* :: [*cidts*, *c*] ⇒ *c* ((*3*∃ - :/ -) [*100*, *10*] *10*)
@*All* :: [*cidts*, *c*] ⇒ *c* ((*3*∀ - :/ -) [*100*, *10*] *10*)
translations
CHOOSE x: *P* ⇌ *CONST Choice*(λ*x*. *P*)

∃ *x*,*xs* : *P* ⇀ *CONST Ex*(λ*x*. ∃ *xs* : *P*)

∃ *x* : *P* ⇌ *CONST Ex*(λ*x*. *P*)
∀ *x*,*xs* : *P* ⇀ *CONST All*(λ*x*. ∀ *xs* : *P*)

∀ *x* : *P* ⇌ *CONST All*(λ*x*. *P*)

axiomatization where
chooseI: $P(t) \Longrightarrow P(CHOOSE\ x\colon P(x))$

axiomatization where
choose-det : $(\bigwedge x.\ P(x) \Leftrightarrow Q(x)) \Longrightarrow (CHOOSE\ x\colon P(x)) = (CHOOSE\ x\colon Q(x))$

defs
Ex-def: $Ex(P) \ \equiv\ P(CHOOSE\ x : P(x) = TRUE) = TRUE$
All-def: $All(P) \ \equiv\ \neg(\exists\, x : \neg\ P(x))$

We introduce two constants *arbitrary* and *default* that correspond to unconstrained and overconstrained choice, respectively.

definition *arbitrary*::*c* **where**
arbitrary ≡ *CHOOSE x* : *TRUE*

definition *default*::*c* **where**
default ≡ *CHOOSE x* : *FALSE*

theorem *exI* [*intro*]:
assumes *hyp*: $P(t)$
shows $\exists\, x : P(x)$
proof −

from *hyp* **have** $P(t) = TRUE$ **by** (*rule eqTrueI*)
thus *?thesis* **by** (*unfold Ex-def*, *rule chooseI*)
qed

theorem *exE* [*elim*!]:
assumes *hyp*: $\exists\, x : P(x)$ **and** *r*: $\bigwedge x.\ P(x) \Longrightarrow Q$
shows Q
proof −
from *hyp* **have** $P(CHOOSE\ x : P(x) = TRUE) = TRUE$ **by** (*unfold Ex-def*)
hence $P(CHOOSE\ x : P(x) = TRUE)$ **by** (*rule eqTrueD*)
thus Q **by** (*rule r*)
qed

theorem *allI* [*intro*!]:
assumes *hyp*: $\bigwedge x.\ P(x)$
shows $\forall\, x : P(x)$
proof (*unfold All-def*, *rule*)
assume $\exists\, x : \neg\ P(x)$
then obtain x **where** $\neg\ P(x)$ **..**
from *this hyp* **show** *FALSE* **by** (*rule notE*)
qed

theorem *spec*:
assumes *hyp*: $\forall\, x : P(x)$
shows $P(x)$
proof (*rule contradiction*)
assume *contra*: $\neg\ P(x)$
hence $\exists\, x : \neg\ P(x)$ **by** (*rule exI*)
with *hyp* **show** *FALSE* **by** (*unfold All-def*, *elim notE*)
qed

theorem *allE* [*elim*]:
assumes *hyp*: $\forall\, x : P(x)$ **and** *r*: $P(x) \Longrightarrow Q$
shows Q
proof (*rule r*)
from *hyp* **show** $P(x)$ **by** (*rule spec*)
qed

theorem *all-dupE*:
assumes *hyp*: $\forall\, x : P(x)$ **and** *r*: $[\![\ P(x); \forall\, x : P(x)\ ]\!] \Longrightarrow Q$
shows Q
proof (*rule r*)
from *hyp* **show** $P(x)$ **by** (*rule spec*)
qed (*rule hyp*)

lemma *chooseI-ex*: $\exists\, x : P(x) \Longrightarrow P(CHOOSE\ x : P(x))$
by (*elim exE chooseI*)

lemma *chooseI2*:

assumes p: $P(a)$ **and** q:$\bigwedge x.\ P(x) \Longrightarrow Q(x)$
shows $Q(CHOOSE\ x : P(x))$
proof ($rule\ q$)
from p **show** $P(CHOOSE\ x : P(x))$ **by** ($rule\ chooseI$)
qed

lemma *chooseI2-ex*:
assumes p: $\exists\, x : P(x)$ **and** q:$\bigwedge x.\ P(x) \Longrightarrow Q(x)$
shows $Q(CHOOSE\ x : P(x))$
proof ($rule\ q$)
from p **show** $P(CHOOSE\ x : P(x))$ **by** (*rule chooseI-ex*)
qed

lemma *choose-equality* [*intro*]:
assumes $P(t)$ **and** $\bigwedge x.\ P(x) \Longrightarrow x{=}a$
shows $(CHOOSE\ x : P(x)) = a$
using *assms* **by** (*rule chooseI2*[**where** $Q{=}\lambda x.\ x{=}a$])

lemmas *choose-equality'* = *sym*[*OF choose-equality*, *standard*, *intro*]

Skolemization rule: note that the existential quantifier in the conclusion introduces an operator (of type $c \Rightarrow c$), not a value; second-order quantification is necessary here.

lemma *skolemI*:
assumes h: $\forall\, x : \exists\, y$: $P(x,y)$ **shows** $\exists f : \forall\, x : P(x, f(x))$
proof −
have $\forall\, x : P(x,\ CHOOSE\ y : P(x,y))$
proof
fix x
from *h*[*THEN spec*] **show** $P(x,\ CHOOSE\ y$: $P(x,y))$ **by** (*rule chooseI-ex*)
qed
thus *?thesis* **by** *iprover*
qed

lemma *skolem*:
$(\forall\, x : \exists\, y : P(x,y)) \Leftrightarrow (\exists f : \forall\, x : P(x, f(x)))$ (**is** *?lhs* ⇔ *?rhs*)
proof
assume *?lhs* **thus** *?rhs* **by** (*rule skolemI*)
next
assume *?rhs* **thus** *?lhs* **by** *iprover*
qed

1.4 Setting up the automatic proof methods

1.4.1 Reflection of meta-level to object-level

Our next goal is to getting Isabelle's automated tactics to work for TLA$^+$. We follow the setup chosen for Isabelle/HOL as far as it applies to TLA$^+$.

The following lemmas, when used as rewrite rules, replace meta- by object-level connectives.

lemma *atomize-all* [*atomize*]: $(\bigwedge x.\ P(x)) \equiv \mathit{Trueprop}\ (\forall\, x : P(x))$
proof
  assume $\bigwedge x.\ P(x)$ **thus** $\forall\, x : P(x)$ **..**
next
  assume $\forall\, x : P(x)$ **thus** $\bigwedge x.\ P(x)$ **..**
qed

lemma *atomize-imp* [*atomize*]: $(A \Longrightarrow B) \equiv \mathit{Trueprop}\ (A \Rightarrow B)$
proof
  assume $A \Longrightarrow B$ **thus** $A \Rightarrow B$ **..**
next
  assume $A \Rightarrow B$ **and** A **thus** B **by** (*rule mp*)
qed

lemma *atomize-not* [*atomize*]: $(A \Longrightarrow \mathit{FALSE}) \equiv \mathit{Trueprop}(\neg A)$
proof
  assume $A \Longrightarrow \mathit{FALSE}$ **thus** $\neg A$ **by** (*rule notI*)
next
  assume $\neg A$ **and** A **thus** FALSE **by** (*rule notE*)
qed

lemma *atomize-eq* [*atomize*]: $(x \equiv y) \equiv \mathit{Trueprop}\ (x = y)$
proof
  assume *1*: $x \equiv y$
  show $x = y$ **by** (*unfold 1*, *rule refl*)
next
  assume $x = y$
  thus $x \equiv y$ **by** (*rule eq-reflection*)
qed

lemma *atomize-conj* [*atomize*]: $(A\ \&\&\&\ B) \equiv \mathit{Trueprop}\ (A \wedge B)$
proof
  assume *conj*: $A\ \&\&\&\ B$
  show $A \wedge B$
  proof
    from *conj* **show** A **by** (*rule conjunctionD1*)
    from *conj* **show** B **by** (*rule conjunctionD2*)
  qed
next
  assume *conj*: $A \wedge B$
  show $A\ \&\&\&\ B$
  proof −
    from *conj* **show** A **..**
    from *conj* **show** B **..**
  qed
qed

lemmas [*symmetric*,*rulify*] = *atomize-all atomize-imp*
and [*symmetric*,*defn*] = *atomize-all atomize-imp atomize-eq*

setup *AtomizeElim.setup*

lemma *atomize-exL*[*atomize-elim*]: $(\bigwedge x.\ P(x) \Longrightarrow Q) \equiv ((\exists\, x : P(x)) \Longrightarrow Q)$
by *rule iprover+*

lemma *atomize-conjL*[*atomize-elim*]: $(A \Longrightarrow B \Longrightarrow C) \equiv (A \wedge B \Longrightarrow C)$
by *rule iprover+*

lemma *atomize-disjL*[*atomize-elim*]: $((A \Longrightarrow C) \Longrightarrow (B \Longrightarrow C) \Longrightarrow C) \equiv ((A \vee B \Longrightarrow C) \Longrightarrow C)$
by *rule iprover+*

lemma *atomize-elimL*[*atomize-elim*]: $(\bigwedge B.\ (A \Longrightarrow B) \Longrightarrow B) \equiv Trueprop(A)$ **..**

1.4.2 Setting up the classical reasoner

We now instantiate Isabelle's classical reasoner for TLA$^+$. This includes methods such as `fast` and `blast`.

lemma *thin-refl*: $[\![x{=}x;\ PROP\ W]\!] \Longrightarrow PROP\ W$ **.**

```
ML ‹

  (* functions to take apart judgments and formulas, see
     Isabelle reference manual, section 9.3 *)
  fun dest-Trueprop (Const(@{const-name Trueprop}, -) $ P) = P
    | dest-Trueprop t = raise TERM (dest-Trueprop, [t]);

  fun dest-eq (Const(@{const-name eq}, -) $ t $ u) = (t,u)
    | dest-eq t = raise TERM (dest-eq, [t]);

  fun dest-imp (Const(@{const-name imp}, -) $ A $ B) = (A, B)
    | dest-imp t = raise TERM (dest-imp, [t]);

(**
  structure Hypsubst-Data =
    struct
    structure Simplifier = Simplifier
    val dest-Trueprop = dest-Trueprop
    val dest-eq = dest-eq
    val dest-imp = dest-imp
    val eq-reflection = @{thm eq-reflection}
    val rev-eq-reflection = @{thm meta-to-obj-eq}
    val imp-intr = @{thm impI}
```

```
  val rev-mp = @{thm rev-mp}
  val subst = @{thm subst}
  val sym = @{thm sym}
  val thin-refl = @{thm thin-refl}
  val prop-subst = @{lemma PROP P(t) ==> PROP prop (x = t ==> PROP
P(x))
                     by (unfold prop-def) (drule eq-reflection, unfold)}
  end;
 structure Hypsubst = HypsubstFun(Hypsubst-Data);
 open Hypsubst;
**)

 structure Hypsubst = Hypsubst(
  val dest-Trueprop = dest-Trueprop
  val dest-eq = dest-eq
  val dest-imp = dest-imp
  val eq-reflection = @{thm eq-reflection}
  val rev-eq-reflection = @{thm meta-to-obj-eq}
  val imp-intr = @{thm impI}
  val rev-mp = @{thm rev-mp}
  val subst = @{thm subst}
  val sym = @{thm sym}
  val thin-refl = @{thm thin-refl}
 );
 open Hypsubst;

(**
 structure Classical-Data =
  struct
  val imp-elim       = @{thm impCE'}
  val not-elim       = @{thm notE}
  val swap           = @{thm swap}
  val classical      = @{thm classical}
  val sizef          = Drule.size-of-thm
  val hyp-subst-tacs = [Hypsubst.hyp-subst-tac]
  end;
 structure Classical = ClassicalFun(Classical-Data);
**)

 structure Classical = Classical(
  val imp-elim       = @{thm impCE'}
  val not-elim       = @{thm notE}
  val swap           = @{thm swap}
  val classical      = @{thm classical}
  val sizef          = Drule.size-of-thm
  val hyp-subst-tacs = [Hypsubst.hyp-subst-tac]
 );
 structure BasicClassical : BASIC-CLASSICAL = Classical;
 open BasicClassical;
```

⟫

setup *hypsubst-setup*
setup *Classical.setup*

declare *refl* [*intro*!]
and *trueI* [*intro*!]
and *conjI* [*intro*!]
and *disjI* [*intro*!]
and *impI* [*intro*!]
and *notI* [*intro*!]
and *iffI* [*intro*!]
and *cond-boolI* [*intro*!]
and *cond-eqLI* [*intro*!]
and *cond-eqRI* [*intro*!]

and *conjE* [*elim*!]
and *disjE* [*elim*!]
and *impCE* [*elim*!]
and *falseE* [*elim*!]
and *iffCE* [*elim*!]
and *cond-boolE* [*elim*!]
and *cond-eqLE* [*elim*!]
and *cond-eqRE* [*elim*!]

and *allI* [*intro*!]
and *exI* [*intro*]
and *exE* [*elim*!]
and *allE* [*elim*]
and *choose-equality* [*intro*]
and *sym*[*OF choose-equality*, *intro*]

ML ⟪
(**
structure Blast = *Blast*
(*struct*
val thy = @{*theory*}
type claset = *Classical.claset*
val equality-name = @{*const-name PredicateLogic.eq*}
val not-name = @{*const-name PredicateLogic.Not*}
val notE = @{*thm notE*}
val ccontr = @{*thm contradiction*}
val contr-tac = *Classical.contr-tac*
val dup-intr = *Classical.dup-intr*
val hyp-subst-tac = *Hypsubst.blast-hyp-subst-tac*
val rep-cs = *Classical.rep-cs*
val cla-modifiers = *Classical.cla-modifiers*;
val cla-meth′ = *Classical.cla-meth′*

```
  end );
**)

 structure Blast = Blast
 (
  structure Classical = Classical
  val Trueprop-const = dest-Const @{const Trueprop}
  val equality-name = @{const-name PredicateLogic.eq}
  val not-name = @{const-name PredicateLogic.Not}
  val notE = @{thm notE}
  val ccontr = @{thm contradiction}
  val hyp-subst-tac = Hypsubst.blast-hyp-subst-tac
 );
⟫
```

setup *Blast.setup*

1.4.3 Setting up the simplifier

We instantiate the simplifier, Isabelle's generic rewriting machinery. Equational laws for predicate logic will be proven below; they automate much of the purely logical reasoning.

lemma *if-bool-eq-conj*:
 $(IF\ A\ THEN\ B\ ELSE\ C) \Leftrightarrow ((A \Rightarrow B) \wedge (\neg A \Rightarrow C))$
by *fast*

A copy of Isabelle's meta-level implication is introduced, which is used internally by the simplifier for fine-tuning congruence rules by simplifying their premises.

definition *simp-implies* :: [*prop*, *prop*] ⇒ *prop* (**infixr** =*simp*=> *1*) **where**
 simp-implies ≡ *op* ⟹

lemma *simp-impliesI*:
 assumes *PQ*: (*PROP P* ==> *PROP Q*)
 shows *PROP P* =*simp*=> *PROP Q*
 unfolding *simp-implies-def* **by** (*rule PQ*)

lemma *simp-impliesE*:
 assumes *PQ*: *PROP P* =*simp*=> *PROP Q*
 and *P*: *PROP P* **and** *QR*: *PROP Q* ==> *PROP R*
 shows *PROP R*
proof −
 from *P* **have** *PROP Q* **by** (*rule PQ* [*unfolded simp-implies-def*])
 thus *PROP R* **by** (*rule QR*)
qed

lemma *simp-implies-cong*:
 assumes *PP′* :*PROP P* == *PROP P′*

and *P'QQ'*: *PROP P' ==> (PROP Q == PROP Q')*
shows *(PROP P =simp=> PROP Q) == (PROP P' =simp=> PROP Q')*
unfolding *simp-implies-def* **proof** (*rule equal-intr-rule*)
assume *PQ*: *PROP P ==> PROP Q* **and** *P'*: *PROP P'*
from *PP'* [*symmetric*] **and** *P'* **have** *PROP P*
by (*rule equal-elim-rule1*)
hence *PROP Q* **by** (*rule PQ*)
with *P'QQ'* [*OF P'*] **show** *PROP Q'* **by** (*rule equal-elim-rule1*)
next
assume *P'Q'*: *PROP P' ==> PROP Q'* **and** *P*: *PROP P*
from *PP'* **and** *P* **have** *P'*: *PROP P'* **by** (*rule equal-elim-rule1*)
hence *PROP Q'* **by** (*rule P'Q'*)
with *P'QQ'* [*OF P'*, *symmetric*] **show** *PROP Q* **by** (*rule equal-elim-rule1*)
qed

use *simplifier-setup.ML*

setup ⟪
Simplifier.map-simpset-global (K Simpdata.PL-basic-ss)
#> Simplifier.method-setup Splitter.split-modifiers
#> Splitter.setup
#> Simpdata.clasimp-setup
#> EqSubst.setup
⟫

lemma *trueprop-eq-true*: *Trueprop(A = TRUE) ≡ Trueprop(A)*
proof
assume *A = TRUE* **thus** *A* **by** (*rule eqTrueD*)
next
assume *A* **thus** *A = TRUE* **by** (*rule eqTrueI*)
qed

lemma *trueprop-true-eq*: *Trueprop(TRUE = A) ≡ Trueprop(A)*
proof
assume *TRUE = A*
hence *A = TRUE* **by** (*rule sym*)
thus *A* **by** (*rule eqTrueD*)
next
assume *A*
hence *A = TRUE* **by** (*rule eqTrueI*)
thus *TRUE = A* **by** (*rule sym*)
qed

lemmas [*simp*] =
triv-forall-equality
TrueAssumption

trueprop-eq-true *trueprop-true-eq*
refl[*THEN* *eqTrueI*] — $(x = x) \equiv$ *TRUE*
condT *notTrue*
condF *notFalse*
cond-id
false-neq-true[*THEN* *eqBoolean*]
not-sym[*OF* *false-neq-true*, *THEN* *eqBoolean*]
iff-refl

lemmas [*cong*] = *simp-implies-cong*

1.4.4 Reasoning by cases

The next bit of code sets up reasoning by cases as a proper Isar method, so we can write "proof cases" etc. Following the development of FOL, we introduce a set of "shadow connectives" that will only be used for this purpose.

theorems *cases* = *case-split* [*case-names True False*]

definition *cases-equal* **where** *cases-equal* ≡ *eq*
definition *cases-implies* **where** *cases-implies* ≡ *imp*
definition *cases-conj* **where** *cases-conj* ≡ *conj*
definition *cases-forall* **where** *cases-forall*$(P) \equiv \forall x \colon P(x)$
definition *cases-true* **where** *cases-true* ≡ *TRUE*
definition *cases-false* **where** *cases-false* ≡ *FALSE*

lemma *cases-equal-eq*: $(x \equiv y) \equiv$ *Trueprop*(*cases-equal*(x, y))
unfolding *atomize-eq cases-equal-def* **.**

lemma *cases-implies-eq*: $(A \Longrightarrow B) \equiv$ *Trueprop* (*cases-implies*(A,B))
unfolding *atomize-imp cases-implies-def* **.**

lemma *cases-conj-eq*: $(A$ &&& $B) \equiv$ *Trueprop* (*cases-conj*(A,B))
unfolding *atomize-conj cases-conj-def* **.**

lemma *cases-forall-eq*: $(\bigwedge x.\ P(x)) \equiv$ *Trueprop* (*cases-forall* $(\lambda x.\ P(x))$)
unfolding *atomize-all cases-forall-def* **.**

lemma *cases-trueI*: *cases-true*
unfolding *cases-true-def* **..**

lemmas *cases-atomize'* = *cases-implies-eq cases-conj-eq cases-forall-eq*
lemmas *cases-atomize* = *cases-atomize' cases-equal-eq*
lemmas *cases-rulify'* [*symmetric*, *standard*] = *cases-atomize'*
lemmas *cases-rulify* [*symmetric*, *standard*] = *cases-atomize*
lemmas *cases-rulify-fallback* =
cases-equal-def cases-implies-def cases-conj-def cases-forall-def
cases-true-def cases-false-def

lemma *cases-forall-conj*: *cases-forall* $(\lambda x.$ *cases-conj*$(A(x), B(x)))$ ⇔

```
    cases-conj(cases-forall(A), cases-forall(B))
  by (unfold cases-forall-def cases-conj-def) iprover

lemma cases-implies-conj: cases-implies(C, cases-conj(A, B)) ⇔
    cases-conj(cases-implies(C, A), cases-implies(C, B))
  by (unfold cases-implies-def cases-conj-def) iprover

lemma cases-conj-curry: (cases-conj(A, B) ⟹ PROP C) ≡ (A ⟹ B ⟹ PROP
C)
proof
  assume r: cases-conj(A, B) ==> PROP C and ab: A B
  from ab show PROP C
    by (intro r[unfolded cases-conj-def], fast)
next
  assume r: A ==> B ==> PROP C and ab: cases-conj(A, B)
  from ab[unfolded cases-conj-def] show PROP C
    by (intro r, fast, fast)
qed

lemmas cases-conj = cases-forall-conj cases-implies-conj cases-conj-curry

ML ⟪
  structure Induct = Induct
  (
    val cases-default = @{thm cases}
    val atomize = @{thms cases-atomize}
    val rulify = @{thms cases-rulify'}
    val rulify-fallback = @{thms cases-rulify-fallback}
    val equal-def = @{thm cases-equal-def}
    fun dest-def (Const (@{const-name cases-equal}, -) $ t $ u) = SOME (t, u)
      | dest-def - = NONE
    val trivial-tac = match-tac @{thms cases-trueI}
  )
⟫

setup ⟪
  Induct.setup #>
  Context.theory-map (Induct.map-simpset (fn ss => ss
    setmksimps (fn ss => Simpdata.mksimps Simpdata.mksimps-pairs ss #>
    map (Simplifier.rewrite-rule (map Thm.symmetric @{thms cases-rulify-fallback})))
    addsimprocs
      [Simplifier.simproc-global @{theory} swap-cases-false
        [cases-false ==> PROP P ==> PROP Q]
        (fn - => fn - =>
           (fn - $ (P as - $ @{const cases-false}) $ (- $ Q $ -) =>
                 if P <> Q then SOME Drule.swap-prems-eq else NONE
             | - => NONE)),
       Simplifier.simproc-global @{theory} cases-equal-conj-curry
        [cases-conj(P, Q) ==> PROP R]
```

```
      (fn _ => fn _ =>
        (fn _ $ (_ $ P) $ _ =>
            let
              fun is_conj (@{const cases_conj} $ P $ Q) =
                    is_conj P andalso is_conj Q
                | is_conj (Const (@{const_name cases_equal}, _) $ _ $ _) = true
                | is_conj @{const cases_true} = true
                | is_conj @{const cases_false} = true
                | is_conj _ = false
            in if is_conj P then SOME @{thm cases_conj_curry} else NONE end
          | _ => NONE))]))
⟩⟩
```

Pre-simplification of induction and cases rules

lemma [*induct-simp*]: $(\bigwedge x.\ \textit{cases-equal}(x,\ t) \Longrightarrow \textit{PROP}\ P(x)) \equiv \textit{PROP}\ P(t)$
 unfolding *cases-equal-def*
proof
 assume R: $\bigwedge x.\ x = t \Longrightarrow \textit{PROP}\ P(x)$
 show $\textit{PROP}\ P(t)$ **by** (*rule R* [*OF refl*])
next
 fix x **assume** $\textit{PROP}\ P(t)\ x = t$
 then show $\textit{PROP}\ P(x)$ **by** *simp*
qed

lemma [*induct-simp*]: $(\bigwedge x.\ \textit{cases-equal}(t,\ x) \Longrightarrow \textit{PROP}\ P(x)) \equiv \textit{PROP}\ P(t)$
 unfolding *cases-equal-def*
proof
 assume R: $\bigwedge x.\ t = x \Longrightarrow \textit{PROP}\ P(x)$
 show $\textit{PROP}\ P(t)$ **by** (*rule R* [*OF refl*])
next
 fix x **assume** $\textit{PROP}\ P(t)\ t = x$
 then show $\textit{PROP}\ P(x)$ **by** *simp*
qed

lemma [*induct-simp*]: $(\textit{cases-false} \Longrightarrow P) \equiv \textit{Trueprop}(\textit{cases-true})$
unfolding *cases-false-def cases-true-def* **by** (*iprover intro*: *equal-intr-rule*)

lemma [*induct-simp*]: $(\textit{cases-true} \Longrightarrow \textit{PROP}\ P) \equiv \textit{PROP}\ P$
 unfolding *cases-true-def*
proof
 assume R: $\textit{TRUE} \Longrightarrow \textit{PROP}\ P$
 from *trueI* **show** $\textit{PROP}\ P$ **by** (*rule R*)
next
 assume $\textit{PROP}\ P$ **thus** $\textit{PROP}\ P$ **.**
qed

lemma [*induct-simp*]: $(\textit{PROP}\ P \Longrightarrow \textit{cases-true}) \equiv \textit{Trueprop}(\textit{cases-true})$
unfolding *cases-true-def* **by** (*iprover intro*: *equal-intr-rule*)

lemma [*induct-simp*]: $(\bigwedge x.\ \textit{cases-true}) \equiv \textit{Trueprop}(\textit{cases-true})$
unfolding *cases-true-def* **by** (*iprover intro*: *equal-intr-rule*)

lemma [*induct-simp*]: $\textit{Trueprop}(\textit{cases-implies}(\textit{cases-true}, P)) \equiv \textit{Trueprop}(P)$
unfolding *cases-implies-def cases-true-def* **by** (*iprover intro*: *equal-intr-rule*)

lemma [*induct-simp*]: $(x = x) = \textit{TRUE}$
by *simp*

hide-const *cases-forall cases-implies cases-equal cases-conj cases-true cases-false*

1.5 Propositional simplification

1.5.1 Conversion to Boolean values

Because TLA$^+$ is untyped, equivalence is different from equality, and one has to be careful about stating the laws of propositional logic. For example, although the equivalence $(\textit{TRUE} \wedge A) \Leftrightarrow A$ is valid, we cannot state the law $(\textit{TRUE} \wedge A) = A$ because we have no way of knowing the value of, e.g., $\textit{TRUE} \wedge 3$. These equalities are valid only if the connectives are applied to Boolean operands. For automatic reasoning, we are interested in equations that can be used by Isabelle's simplifier. We therefore introduce an auxiliary predicate that is true precisely of Boolean arguments, and an operation that converts arbitrary arguments to an equivalent (in the sense of $\Leftrightarrow$) Boolean expression.

We will prove below that propositional formulas return a Boolean value when applied to arbitrary arguments.

definition *boolify* :: $c \Rightarrow c$ **where**
$\textit{boolify}(x) \equiv \textit{IF } x \textit{ THEN TRUE ELSE FALSE}$

definition *isBool* :: $c \Rightarrow c$ **where**
$\textit{isBool}(x) \equiv \textit{boolify}(x) = x$

The formulas P and $\textit{boolify}(P)$ are inter-derivable (but need of course not be equal, unless P is a Boolean).

lemma *boolifyI* [*intro*!]: $P \Longrightarrow \textit{boolify}(P)$
unfolding *boolify-def* **by** *fast*

lemma *boolifyE* [*elim*!]:
$[\![\ \textit{boolify}(P);\ P \Longrightarrow Q\ ]\!] \Longrightarrow Q$
unfolding *boolify-def* **by** *fast*

lemma *TruepropBoolify* [*simp*]: $\textit{Trueprop}(\textit{boolify}(A)) \equiv \textit{Trueprop}(A)$
by (*rule*, *fast*+)

lemma *boolify-cases*:
assumes $P(\textit{TRUE})$ **and** $P(\textit{FALSE})$

shows $P(\mathit{boolify}(x))$
unfolding *boolify-def* **using** *assms* **by** (*fast intro*: *condI*)

boolify can be defined as $x = \mathit{TRUE}$. For automatic reasoning, we rewrite the latter to the former, and derive calculational rules for *boolify*.

lemma [*simp*]: $(x = \mathit{TRUE}) = \mathit{boolify}(x)$
proof (*cases* x)
case *True* **thus** *?thesis* **by** (*simp add*: *boolify-def*)
next
case *False*
hence $x \neq \mathit{TRUE}$ **by** *fast*
thus *?thesis* **by** (*auto simp add*: *boolify-def*)
qed

lemma [*simp*]: $(\mathit{TRUE} = x) = \mathit{boolify}(x)$
proof (*cases* x)
case *True* **thus** *?thesis* **by** (*simp add*: *boolify-def*)
next
case *False*
hence $\mathit{TRUE} \neq x$ **by** *fast*
thus *?thesis* **by** (*auto simp add*: *boolify-def*)
qed

lemma *boolifyTrue* [*simp*]: $\mathit{boolify}(\mathit{TRUE}) = \mathit{TRUE}$
by (*simp add*: *boolify-def*)

lemma *trueIsBool* [*intro*!,*simp*]: *isBool*(*TRUE*)
by (*unfold isBool-def*, *rule boolifyTrue*)

lemma *boolifyTrueI* [*intro*]: $A \Longrightarrow \mathit{boolify}(A) = \mathit{TRUE}$
by (*simp add*: *boolify-def*)

lemma *boolifyFalse* [*simp*]: $\mathit{boolify}(\mathit{FALSE}) = \mathit{FALSE}$
by (*auto simp add*: *boolify-def*)

lemma *falseIsBool* [*intro*!,*simp*]: *isBool*(*FALSE*)
by (*unfold isBool-def*, *rule boolifyFalse*)

The following lemma is used to turn hypotheses $\neg A$ into rewrite rules $A = \mathit{FALSE}$.

lemma *boolifyFalseI* [*intro*]: $\neg\ A \Longrightarrow \mathit{boolify}(A) = \mathit{FALSE}$
by (*auto simp add*: *boolify-def*)

idempotence of *boolify*

lemma *boolifyBoolify* [*simp*]: $\mathit{boolify}(\mathit{boolify}(x)) = \mathit{boolify}(x)$
by (*auto simp add*: *boolify-def*)

lemma *boolifyIsBool* [*intro*!,*simp*]: *isBool*(*boolify*(x))

by (*unfold isBool-def*, *rule boolifyBoolify*)

lemma *boolifyEquivalent*: *boolify*(x) ⇔ x
by (*auto simp add*: *boolify-def*)

lemma *boolifyTrueFalse*: (*boolify*(x) = *TRUE*) ∨ (*boolify*(x) = *FALSE*)
by (*auto simp add*: *boolify-def*)

lemma *isBoolTrueFalse*:
  assumes *hyp*: *isBool*(x)
  shows (x = *TRUE*) ∨ (x = *FALSE*)
proof −
  from *hyp* **have** *boolify*(x) = x **by** (*unfold isBool-def*)
  hence *bx*: *boolify*(x) ≡ x **by** (*rule eq-reflection*)
  from *boolifyTrueFalse*[*of x*]
  show *?thesis* **by** (*unfold bx*)
qed

lemmas *isBoolE* [*elim*!] = *isBoolTrueFalse*[*THEN disjE*, *standard*]

lemma *boolifyEq* [*simp*]: *boolify*(t=u) = (t=u)
proof (*cases t*=*u*)
  case *True*
  hence (t=u) = *TRUE* **by** (*rule eqTrueI*)
  hence *tu*: (t=u) ≡ *TRUE* **by** (*rule eq-reflection*)
  show *?thesis* **by** (*unfold tu*, *rule boolifyTrue*)
next
  case *False*
  hence (t=u) = *FALSE* **by** (*rule eqBoolean*)
  hence *tu*: (t=u) ≡ *FALSE* **by** (*rule eq-reflection*)
  show *boolify*(t=u) = (t=u) **by** (*unfold tu*, *rule boolifyFalse*)
qed

lemma *eqIsBool* [*intro*!,*simp*]: *isBool*(t=u)
unfolding *isBool-def* **by** (*rule boolifyEq*)

lemma *boolifyCond* [*simp*]:
  boolify(*IF A THEN t ELSE u*) = (*IF A THEN boolify*(t) *ELSE boolify*(u))
by (*auto simp add*: *boolify-def*)

lemma *isBoolCond*[*intro*!,*simp*]:
  ⟦ *isBool*(t); *isBool*(e) ⟧ ⟹ *isBool*(*IF A THEN t ELSE e*)
by (*simp add*: *isBool-def*)

lemma *boolifyNot* [*simp*]: *boolify*(¬ A) = (¬ A)
by (*simp add*: *not-def*)

lemma *notIsBool* [*intro*!,*simp*]: *isBool*(¬ A)
unfolding *isBool-def* **by** (*rule boolifyNot*)

lemma *notBoolIsFalse*:
  assumes *isBool*(A)
  shows $(\neg A) = (A = \mathit{FALSE})$
using *assms* **by** *auto*

lemma *boolifyAnd* [*simp*]: *boolify*$(A \wedge B) = (A \wedge B)$
by (*simp add*: *conj-def*)

lemma *andIsBool* [*intro*!,*simp*]: *isBool*$(A \wedge B)$
unfolding *isBool-def* **by** (*rule boolifyAnd*)

lemma *boolifyOr* [*simp*]: *boolify*$(A \vee B) = (A \vee B)$
by (*simp add*: *disj-def*)

lemma *orIsBool* [*intro*!,*simp*]: *isBool*$(A \vee B)$
unfolding *isBool-def* **by** (*rule boolifyOr*)

lemma *boolifyImp* [*simp*]: *boolify*$(A \Rightarrow B) = (A \Rightarrow B)$
by (*simp add*: *imp-def*)

lemma *impIsBool* [*intro*!,*simp*]: *isBool*$(A \Rightarrow B)$
unfolding *isBool-def* **by** (*rule boolifyImp*)

lemma *boolifyIff* [*simp*]: *boolify*$(A \Leftrightarrow B) = (A \Leftrightarrow B)$
by (*simp add*: *iff-def*)

lemma *iffIsBool* [*intro*!,*simp*]: *isBool*$(A \Leftrightarrow B)$
unfolding *isBool-def* **by** (*rule boolifyIff*)

We can now rewrite equivalences to equations between "boolified" arguments, and this gives rise to a technique for proving equations between formulas.

lemma *boolEqual*:
  assumes $P \Leftrightarrow Q$ **and** *isBool*(P) **and** *isBool*(Q)
  shows $P = Q$
using *assms* **by** *auto*

The following variant converts equivalences to equations. It might be useful as a (non-conditional) simplification rule, but I suspect that for goals it is more useful to use the standard introduction rule reducing an equivalence to two implications.

For assumptions we can use lemma *boolEqual* for turning equivalences into conditional rewrites.

lemma *iffIsBoolifyEqual*: $(A \Leftrightarrow B) = (\mathit{boolify}(A) = \mathit{boolify}(B))$
proof (*rule boolEqual*)
  show $(A \Leftrightarrow B) \Leftrightarrow (\mathit{boolify}(A) = \mathit{boolify}(B))$ **by** (*auto simp*: *boolifyFalseI*)
qed (*simp-all*)

lemma *iffThenBoolifyEqual*:
  assumes $A \Leftrightarrow B$ **shows** $boolify(A) = boolify(B)$
using *assms* **by** (*simp add*: *iffIsBoolifyEqual*)

lemma *boolEqualIff*:
  assumes *isBool*(P) **and** *isBool*(Q)
  shows $(P = Q) = (P \Leftrightarrow Q)$
using *assms* **by** (*auto intro*: *boolEqual*)

```
ML ⟪
  structure Simpdata =
  struct
    open Simpdata;
    val mksimps-pairs = [(@{const-name Not}, (@{thms boolifyFalseI}, true)),
                         (@{const-name iff}, (@{thms iffThenBoolifyEqual}, true))]
                        @ mksimps-pairs;
  end;

  open Simpdata;
⟫

declaration ⟪ fn - =>
  Simplifier.map-ss (fn ss => ss setmksimps (mksimps mksimps-pairs))
⟫
```

The following code rewrites $x = y$ to *FALSE* in the presence of a premise $y \neq x$ or $(y = x) = FALSE$. The simplifier is set up so that $y = x$ is already simplified to *FALSE*, this code adds symmetry of disequality to simplification.

lemma *symEqLeft*: $(x = y) = b \Longrightarrow (y = x) = b$
by (*auto simp*: *boolEqualIff*)

```
simproc-setup neq (x = y) = ⟪ fn - =>
let
  val neq-to-EQ-False = @{thm not-sym} RS @{thm eqBoolean} RS @{thm eq-reflection};
  val symEqLeft-to-symEQLeft = @{thm symEqLeft} RS @{thm eq-reflection};
  fun is-neq lhs rhs thm =
    (case Thm.prop-of thm of
      - $ (Not' $ (eq' $ l' $ r')) =>
        Not' = @{const Not} andalso eq' = @{const eq} andalso
        r' aconv lhs andalso l' aconv rhs
    | - $ (eq' $ (eq'' $ l' $ r') $ f') =>
        eq' = @{const eq} andalso eq'' = @{const eq} andalso
        f' = @{const FALSE} andalso r' aconv lhs andalso l' aconv rhs
    | - => false);
  fun proc ss ct =
    (case Thm.term-of ct of
```

```
      eq $ lhs $ rhs =>
        (case find-first (is-neq lhs rhs) (Simplifier.prems-of ss) of
          SOME thm => SOME ((thm RS symEqLeft-to-symEQLeft)
                            handle - => thm RS neq-to-EQ-False)
        | NONE => NONE)
     | - => NONE);
in proc end;
⟫
```

lemma *boolifyEx* [*simp*]: $boolify(Ex(P)) = Ex(P)$
by (*simp add*: *Ex-def*)

lemma *exIsBool* [*intro*!,*simp*]: $isBool(Ex(P))$
unfolding *isBool-def* **by** (*rule boolifyEx*)

lemma *boolifyAll* [*simp*]: $boolify(All(P)) = All(P)$
by (*simp add*: *All-def*)

lemma *allIsBool* [*intro*!,*simp*]: $isBool(All(P))$
unfolding *isBool-def* **by** (*rule boolifyAll*)

lemma [*intro*!]:
$[\![isBool(P); Q \Leftrightarrow P]\!] \Longrightarrow boolify(Q) = P$
$[\![isBool(P); P \Leftrightarrow Q]\!] \Longrightarrow P = boolify(Q)$
$P \Longrightarrow TRUE = P$
$P \Longrightarrow P = TRUE$
$[\![isBool(P); P \Longrightarrow FALSE]\!] \Longrightarrow FALSE = P$
$[\![isBool(P); P \Longrightarrow FALSE]\!] \Longrightarrow P = FALSE$
$[\![isBool(P); t=u \Leftrightarrow P]\!] \Longrightarrow (t=u) = P$
$[\![isBool(P); P \Leftrightarrow t=u]\!] \Longrightarrow P = (t=u)$
$[\![isBool(P); \neg Q \Leftrightarrow P]\!] \Longrightarrow (\neg Q) = P$
$[\![isBool(P); P \Leftrightarrow \neg Q]\!] \Longrightarrow P = (\neg Q)$
$[\![isBool(P); P \Leftrightarrow (Q \land R)]\!] \Longrightarrow P = (Q \land R)$
$[\![isBool(P); (Q \land R) \Leftrightarrow P]\!] \Longrightarrow (Q \land R) = P$
$[\![isBool(P); P \Leftrightarrow (Q \lor R)]\!] \Longrightarrow P = (Q \lor R)$
$[\![isBool(P); (Q \lor R) \Leftrightarrow P]\!] \Longrightarrow (Q \lor R) = P$
$[\![isBool(P); P \Leftrightarrow (Q \Rightarrow R)]\!] \Longrightarrow P = (Q \Rightarrow R)$
$[\![isBool(P); (Q \Rightarrow R) \Leftrightarrow P]\!] \Longrightarrow (Q \Rightarrow R) = P$
$[\![isBool(P); P \Leftrightarrow (Q \Leftrightarrow R)]\!] \Longrightarrow P = (Q \Leftrightarrow R)$
$[\![isBool(P); (Q \Leftrightarrow R) \Leftrightarrow P]\!] \Longrightarrow (Q \Leftrightarrow R) = P$
$[\![isBool(P); All(A) \Leftrightarrow P]\!] \Longrightarrow All(A) = P$
$[\![isBool(P); P \Leftrightarrow All(A)]\!] \Longrightarrow P = All(A)$
$[\![isBool(P); Ex(A) \Leftrightarrow P]\!] \Longrightarrow Ex(A) = P$
$[\![isBool(P); P \Leftrightarrow Ex(A)]\!] \Longrightarrow P = Ex(A)$
by (*auto simp*: *boolEqualIff*)

lemma *notBoolifyFalse* [*simp*]: $(\neg A) = (boolify(A) = FALSE)$

by *auto*

Orient equations with Boolean constants such that the constant appears on the right-hand side.

lemma *boolConstEqual* [*simp*]:
  $(TRUE = P) = (P = TRUE)$
  $(FALSE = P) = (P = FALSE)$
by *blast+*

1.5.2 Simplification laws for conditionals

lemma *splitCond* [*split*]:
  assumes *q*: $\bigwedge x.\ isBool(Q(x))$
  shows $Q(IF\ P\ THEN\ t\ ELSE\ e) = ((P \Rightarrow Q(t)) \wedge (\neg P \Rightarrow Q(e)))$
proof (*cases P*)
  case *True* **thus** *?thesis* **by** (*auto intro*: *q*)
next
  case *False*
  hence $(IF\ P\ THEN\ t\ ELSE\ e) = e$ **by** (*rule condElse*)
  thus *?thesis* **by** (*auto intro*: *q*)
qed

lemma *splitCondAsm*: — useful with conditionals in hypotheses
  assumes $\bigwedge x.\ isBool(Q(x))$
  shows $Q(IF\ P\ THEN\ t\ ELSE\ e) = (\neg((P \wedge \neg Q(t)) \vee (\neg P \wedge \neg Q(e))))$
using *assms* **by** (*simp only*: *splitCond*, *blast*)

lemma *condCong* :
  $P = Q \Longrightarrow (IF\ P\ THEN\ t\ ELSE\ e) = (IF\ Q\ THEN\ t\ ELSE\ e)$
by *simp*

lemma *condFullCong*: — not active by default, because too expensive
  $[\![P = Q;\ Q \Longrightarrow t{=}t';\ \neg Q \Longrightarrow e{=}e']\!] \Longrightarrow (IF\ P\ THEN\ t\ ELSE\ e) = (IF\ Q\ THEN\ t'\ ELSE\ e')$
by *auto*

lemma *substCond* [*intro*]:
  assumes $A \Leftrightarrow B$
  and $[\![\ A;B\ ]\!] \Longrightarrow t{=}v$ **and** $[\![\ \neg A;\ \neg B\ ]\!] \Longrightarrow e{=}f$
  shows $(IF\ A\ THEN\ t\ ELSE\ e) = (IF\ B\ THEN\ v\ ELSE\ f)$
using *assms* **by** *auto*

lemma *cond-simps* [*simp*]:
  $(IF\ x = y\ THEN\ y\ ELSE\ x) = x$
  $(IF\ (IF\ A\ THEN\ B\ ELSE\ C)\ THEN\ t\ ELSE\ e) =$
   $(IF\ (A \wedge B) \vee (\neg A \wedge C)\ THEN\ t\ ELSE\ e)$
  $(IF\ A\ THEN\ (IF\ B\ THEN\ t\ ELSE\ u)\ ELSE\ v) =$
   $(IF\ A \wedge B\ THEN\ t\ ELSE\ IF\ A \wedge \neg B\ THEN\ u\ ELSE\ v)$
by *auto*

1.5.3 Simplification laws for conjunction

lemma *conj-simps* [*simp*]:
$(P \wedge TRUE) = boolify(P)$
$(TRUE \wedge P) = boolify(P)$
$(P \wedge FALSE) = FALSE$
$(FALSE \wedge P) = FALSE$

$(P \wedge P) = boolify(P)$
$(P \wedge P \wedge Q) = (P \wedge Q)$
$((P \wedge Q) \wedge R) = (P \wedge Q \wedge R)$
by *auto*

The congruence rule for conjunction is occasionally useful, but not active by default.

lemma *conj-cong*:
assumes $P = P'$ **and** $P' \Longrightarrow Q = Q'$
shows $(P \wedge Q) = (P' \wedge Q')$
using *assms* **by** *auto*

Commutativity laws are not active by default

lemma *conj-comms*:
$(P \wedge Q) = (Q \wedge P)$
$(P \wedge Q \wedge R) = (Q \wedge P \wedge R)$
by *auto*

1.5.4 Simplification laws for disjunction

lemma *disj-simps* [*simp*]:
$(P \vee TRUE) = TRUE$
$(TRUE \vee P) = TRUE$
$(P \vee FALSE) = boolify(P)$
$(FALSE \vee P) = boolify(P)$

$(P \vee P) = boolify(P)$
$(P \vee P \vee Q) = (P \vee Q)$
$((P \vee Q) \vee R) = (P \vee Q \vee R)$
by *auto*

Congruence rule, not active by default

lemma *disj-cong*:
assumes $P = P'$ **and** $\neg P' \Longrightarrow Q = Q'$
shows $(P \vee Q) = (P' \vee Q')$
using *assms* **by** *auto*

Commutativity laws are not active by default

lemma *disj-comms*:
$(P \vee Q) = (Q \vee P)$
$(P \vee Q \vee R) = (Q \vee P \vee R)$
by *auto*

1.5.5 Simplification laws for negation

Negated formulas create simplifications of the form $A = FALSE$, we therefore prove two versions of the following lemmas to complete critical pairs.

lemma *not-simps* [*simp*]:
$(\neg(P \vee Q)) = (\neg P \wedge \neg Q)$
$(\neg(P \wedge Q)) = (\neg P \vee \neg Q)$
$(\neg(P \Rightarrow Q)) = (P \wedge \neg Q)$
$(\neg(P \Leftrightarrow Q)) = (P \Leftrightarrow \neg Q)$
$(\neg P \Leftrightarrow \neg Q) = (P \Leftrightarrow Q)$
$(\neg\neg P) = \mathit{boolify}(P)$
$(x \neq x) = \mathit{FALSE}$
$\bigwedge P.\ (\neg(\forall x : P(x))) = (\exists x : \neg P(x))$
$\bigwedge P.\ (\neg(\exists x : P(x))) = (\forall x : \neg P(x))$
by (*auto simp del*: *notBoolifyFalse*)

declare *not-simps* [*simplified*,*simp*]

lemma *eqFalse-eqFalse* [*simp*]: $\mathit{isBool}(P) \Longrightarrow ((P = \mathit{FALSE}) = \mathit{FALSE}) = P$
by *auto*

1.5.6 Simplification laws for implication

lemma *imp-simps* [*simp*]:
$(P \Rightarrow \mathit{FALSE}) = (\neg P)$
$(P \Rightarrow \mathit{TRUE}) = \mathit{TRUE}$
$(\mathit{FALSE} \Rightarrow P) = \mathit{TRUE}$
$(\mathit{TRUE} \Rightarrow P) = \mathit{boolify}(P)$
$(P \Rightarrow P) = \mathit{TRUE}$
$(P \Rightarrow \neg P) = (\neg P)$
by *auto*

lemma *imp-cong* [*cong*]:
$(P = P') \Longrightarrow (P' \Longrightarrow (Q = Q')) \Longrightarrow ((P \Rightarrow Q) = (P' \Rightarrow Q'))$
by *auto*

1.5.7 Simplification laws for equivalence

lemma *iff-simps* [*simp*]:
$(\mathit{TRUE} \Leftrightarrow P) = \mathit{boolify}(P)$
$(P \Leftrightarrow \mathit{TRUE}) = \mathit{boolify}(P)$
$(P \Leftrightarrow P) = \mathit{TRUE}$
$(\mathit{FALSE} \Leftrightarrow P) = (\neg P)$
$(P \Leftrightarrow \mathit{FALSE}) = (\neg P)$
by *auto*

lemma *iff-cong* [*cong*]:
$P = P' \Longrightarrow Q = Q' \Longrightarrow (P \Leftrightarrow Q) = (P' \Leftrightarrow Q')$
by *auto*

1.5.8 Simplification laws for quantifiers

lemma *quant-simps* [*simp*]:
  $\bigwedge P.\ (\exists\, x : P) = boolify(P)$
  $\bigwedge P.\ (\forall\, x : P) = boolify(P)$
  $\exists\, x : x{=}t$
  $\exists\, x : t{=}x$
  $\bigwedge P.\ (\exists\, x : x{=}t \wedge P(x)) = boolify(P(t))$
  $\bigwedge P.\ (\exists\, x : t{=}x \wedge P(x)) = boolify(P(t))$
  $\bigwedge P.\ (\forall\, x : x{=}t \Rightarrow P(x)) = boolify(P(t))$
  $\bigwedge P.\ (\forall\, x : t{=}x \Rightarrow P(x)) = boolify(P(t))$
by *auto*

Miniscoping of quantifiers.

lemma *miniscope-ex* [*simp*]:
  $\bigwedge P\ Q.\ (\exists\, x : P(x) \wedge Q) = ((\exists\, x : P(x)) \wedge Q)$
  $\bigwedge P\ Q.\ (\exists\, x : P \wedge Q(x)) = (P \wedge (\exists\, x : Q(x)))$
  $\bigwedge P\ Q.\ (\exists\, x : P(x) \vee Q) = ((\exists\, x : P(x)) \vee Q)$
  $\bigwedge P\ Q.\ (\exists\, x : P \vee Q(x)) = (P \vee (\exists\, x : Q(x)))$
  $\bigwedge P\ Q.\ (\exists\, x : P(x) \Rightarrow Q) = ((\forall\, x : P(x)) \Rightarrow Q)$
  $\bigwedge P\ Q.\ (\exists\, x : P \Rightarrow Q(x)) = (P \Rightarrow (\exists\, x : Q(x)))$
by *auto*

lemma *miniscope-all* [*simp*]:
  $\bigwedge P\ Q.\ (\forall\, x : P(x) \wedge Q) = ((\forall\, x : P(x)) \wedge Q)$
  $\bigwedge P\ Q.\ (\forall\, x : P \wedge Q(x)) = (P \wedge (\forall\, x : Q(x)))$
  $\bigwedge P\ Q.\ (\forall\, x : P(x) \vee Q) = ((\forall\, x : P(x)) \vee Q)$
  $\bigwedge P\ Q.\ (\forall\, x : P \vee Q(x)) = (P \vee (\forall\, x : Q(x)))$
  $\bigwedge P\ Q.\ (\forall\, x : P(x) \Rightarrow Q) = ((\exists\, x : P(x)) \Rightarrow Q)$
  $\bigwedge P\ Q.\ (\forall\, x : P \Rightarrow Q(x)) = (P \Rightarrow (\forall\, x : Q(x)))$
by *auto*

lemma *choose-trivial* [*simp*]: $(CHOOSE\ x : x = t) = t$
by (*rule chooseI*, *rule refl*)

declare *choose-det* [*cong*]

A *CHOOSE* expression evaluates to *default* if the only possible value satisfying the predicate equals *default*. Note that the reverse implication is not necessarily true: there could be several values satisfying $P(x)$, including *default*, and *CHOOSE* may return *default*. This rule can be useful for reasoning about CASE expressions where none of the guards is true.

lemma *equal-default* [*intro*!]:
  assumes *p*: $\forall\, x : P(x) \Rightarrow x = default$
  shows $(CHOOSE\ x : P(x)) = default$
proof (*cases* $\exists\, x : P(x)$)
  case *True*
  then obtain *a* **where** *a*: $P(a)$ **..**
  thus *?thesis* **proof** (*rule chooseI2*[**where** $P{=}P$])

fix x
assume $P(x)$
with p **show** $x = default$ **by** $blast$
qed
next
case $False$ **thus** $?thesis$
unfolding $default\text{-}def$ **by** ($blast$ $intro$: $choose\text{-}det$)
qed

lemmas [$intro!$] = sym[OF $equal\text{-}default$, $standard$]

Similar lemma for *arbitrary*.

lemma $equal\text{-}arbitrary$:
assumes p: $\forall x : P(x)$
shows $(CHOOSE\ x : P(x)) = arbitrary$
unfolding $arbitrary\text{-}def$ **proof** ($rule$ $choose\text{-}det$)
fix x
from p **show** $P(x) \Leftrightarrow TRUE$ **by** $blast$
qed

1.5.9 Distributive laws

Not active by default.

lemma $prop\text{-}distrib$:
$(P \wedge (Q \vee R)) = ((P \wedge Q) \vee (P \wedge R))$
$((Q \vee R) \wedge P) = ((Q \wedge P) \vee (R \wedge P))$
$(P \vee (Q \wedge R)) = ((P \vee Q) \wedge (P \vee R))$
$((Q \wedge R) \vee P) = ((Q \vee P) \wedge (R \vee P))$
$(P \Rightarrow (Q \wedge R)) = ((P \Rightarrow Q) \wedge (P \Rightarrow R))$
$((P \wedge Q) \Rightarrow R) = (P \Rightarrow (Q \Rightarrow R))$
$((P \vee Q) \Rightarrow R) = ((P \Rightarrow R) \wedge (Q \Rightarrow R))$
by $auto$

lemma $quant\text{-}distrib$:
$\bigwedge P\ Q.\ (\exists x : P(x) \vee Q(x)) = ((\exists x : P(x)) \vee (\exists x : Q(x)))$
$\bigwedge P\ Q.\ (\forall x : P(x) \wedge Q(x)) = ((\forall x : P(x)) \wedge (\forall x : Q(x)))$
by $auto$

1.5.10 Further calculational laws

lemma $cases\text{-}simp$: $((P \Rightarrow Q) \wedge (\neg P \Rightarrow Q)) = boolify(Q)$
by $auto$

end

2 TLA$^+$ Set Theory

theory $SetTheory$

imports *PredicateLogic*
begin

This theory defines the version of Zermelo-Fränkel set theory that underlies TLA$^+$.

2.1 Basic syntax and axiomatic basis of set theory.

We take the set-theoretic constructs of TLA$^+$, but add generalized intersection for symmetry and convenience. (Note that *INTER* {} = {}.)

consts
emptyset :: *c* ({} *100*) — empty set
upair :: [*c*, *c*] ⇒ *c* — unordered pairs
addElt :: [*c*, *c*] ⇒ *c* — add element to set
infinity :: *c* — infinity set
SUBSET :: *c* ⇒ *c* (*SUBSET* - [*100*]*90*) — power set
UNION :: *c* ⇒ *c* (*UNION* - [*100*]*90*) — generalized union
INTER :: *c* ⇒ *c* (*INTER* - [*100*]*90*) — generalized intersection
cup :: [*c*, *c*] ⇒ *c* (**infixl** *cup 65*) — binary union
cap :: [*c*, *c*] ⇒ *c* (**infixl** *cap 70*) — binary intersection
setminus:: [*c*, *c*] ⇒ *c* (**infixl** \ *65*) — binary set difference
in :: [*c*, *c*] ⇒ *c* (**infixl** *in 50*) — membership relation
subseteq:: [*c*, *c*] ⇒ *c* (**infixl** *subseteq 50*) — subset relation
subsetOf :: [*c*, *c* ⇒ *c*] ⇒ *c* — $subsetOf(S,p) = \{x \in S : p\}$
setOfAll :: [*c*, *c* ⇒ *c*] ⇒ *c* — $setOfAll(S,e) = \{e : x \in S\}$
bChoice :: [*c*, *c* ⇒ *c*] ⇒ *c* — bounded choice
bAll :: [*c*, *c* ⇒ *c*] ⇒ *c* — bounded universal quantifier
bEx :: [*c*, *c* ⇒ *c*] ⇒ *c* — bounded existential quantifier

notation (*xsymbols*)
cup (**infixl** ∪ *65*) **and**
cap (**infixl** ∩ *70*) **and**
setminus (**infixl** \ *65*) **and**
in (**infixl** ∈ *50*) **and**
subseteq (**infixl** ⊆ *50*)

notation (*HTML* **output**)
cup (**infixl** ∪ *65*) **and**
cap (**infixl** ∩ *70*) **and**
setminus (**infixl** \ *65*) **and**
in (**infixl** ∈ *50*) **and**
subseteq (**infixl** ⊆ *50*)

abbreviation $notin(a, S) \equiv \neg(a \in S)$ — negated membership

notation
notin (**infixl** *notin 50*)
notation (*xsymbols*)

notin (**infixl** $\notin$ *50*)
notation (*HTML* **output**)
notin (**infixl** $\notin$ *50*)

abbreviation (*input*) *supseteq*(S, T) $\equiv$ $T \subseteq S$
notation
supseteq (**infixl** *supseteq 50*)
notation (*xsymbols*)
supseteq (**infixl** $\supseteq$ *50*)
notation (*HTML* **output**)
supseteq (**infixl** $\supseteq$ *50*)

Concrete syntax: proper sub and superset

definition *psubset* :: [c, c] $\Rightarrow$ c (**infixl** *subset 50*)
where S *subset* $T \equiv S \subseteq T \wedge S \neq T$

notation (*xsymbols*)
psubset (**infixl** $\subset$ *50*)
notation (*HTML* **output**)
psubset (**infixl** $\subset$ *50*)

abbreviation (*input*) *psupset*(S,T) $\equiv$ $T \subset S$
notation
psupset (**infix** *supset 50*)
notation (*xsymbols*)
psupset (**infix** $\supset$ *50*)
notation (*HTML* **output**)
psupset (**infix** $\supset$ *50*)

lemma *psubset-intro* [*intro*!]:
$[\![S \subseteq T ; S \neq T]\!] \Longrightarrow S \subset T$
unfolding *psubset-def* **by** *safe*

lemma *psubset-elim* [*elim*!]:
$[\![S \subset T ; [\![S \subseteq T ; S \neq T]\!] \Longrightarrow C]\!] \Longrightarrow C$
unfolding *psubset-def* **by** *safe*

Concrete syntax: set enumerations

nonterminal *cs*

syntax
:: $c \Rightarrow cs$ (-)
@*cs* :: [c, cs] $\Rightarrow$ cs (-,/ -)
@*enumset* :: $cs \Rightarrow c$ ({(-)})

translations
$\{x, xs\}$ $\rightleftharpoons$ *CONST addElt*(x, $\{xs\}$)
$\{x\}$ $\rightleftharpoons$ *CONST addElt*(x, $\{\}$)

abbreviation *BOOLEAN* :: *c* **where**
BOOLEAN ≡ {*TRUE*, *FALSE*}

Concrete syntax: bounded quantification

syntax

```
@bChoice :: [idt, c, c] ⇒ c      ((3CHOOSE - \in - :/ -) [100,0,0] 10)
@bEx     :: [cidts, c, c] ⇒ c    ((3EX - in - :/ -) [100,0,0] 10)
@bAll    :: [cidts, c, c] ⇒ c    ((3ALL - in - :/ -) [100,0,0] 10)
```

syntax (*xsymbols*)

```
@bChoice :: [idt, c, c] ⇒ c      ((3CHOOSE - ∈ - :/ -) [100,0,0]10)
@bEx     :: [cidts, c, c] ⇒ c    ((3∃ - ∈ - :/ -) [100,0,0] 10)
@bAll    :: [cidts, c, c] ⇒ c    ((3∀ - ∈ - :/ -) [100,0,0] 10)
```

translations

```
CHOOSE x ∈ S : P      ⇌  CONST bChoice(S, λx. P)

∃ x, xs ∈ S : P       ⇀  CONST bEx(S, λx. ∃ xs ∈ S : P)
∃ x ∈ S : P           ⇀  CONST bEx(S, λx. P)
∀ x, xs ∈ S : P       ⇀  CONST bAll(S, λx. ∀ xs ∈ S : P)
∀ x ∈ S : P           ⇀  CONST bAll(S, λx. P)
```

print-translation ⟪

```
let
    fun bEx-tr′ [S, Abs(x, T, P as (Const (@{const-syntax bEx},-) $ S′ $ Q))] =
        (∗ bEx(S, bEx(S′, Q)) => \\E x,y \\in S : Q if S = S′ ∗)
        let val (y,Q′) = Syntax-Trans.atomic-abs-tr′ (x,T,Q)
            val (- $ xs $ set $ Q′′) = bEx-tr′ [S′, Q′]
  in if S = S′
      then Syntax.const @bEx $ (Syntax.const @cidts $ y $ xs)
          $ set $ Q′′
      else Syntax.const @bEx $ y $ S $
        (Syntax.const @bEx $ xs $ set $ Q′′)
  end
| bEx-tr′ [S, Abs(x, T, P)] =
    let val (x′,P′) = Syntax-Trans.atomic-abs-tr′ (x,T,P)
    in (Syntax.const @bEx) $ x′ $ S $ P′
    end
| bEx-tr′ - = raise Match;
     fun bAll-tr′ [S, Abs(x, T, P as (Const (@{const-syntax bAll},-) $ S′ $ Q))]
=
        (∗ bAll(S, bAll(S′, Q)) => \\A x,y \\in S : Q if S = S′ ∗)
        let val (y,Q′) = Syntax-Trans.atomic-abs-tr′ (x,T,Q)
            val (- $ xs $ set $ Q′′) = bAll-tr′ [S′, Q′]
  in if S = S′
      then Syntax.const @bAll $ (Syntax.const @cidts $ y $ xs)
          $ set $ Q′′
      else Syntax.const @bAll $ y $ S $
        (Syntax.const @bAll $ xs $ set $ Q′′)
  end
```

```
 | bAll-tr' [S, Abs(x, T, P)] =
     let val (x',P') = Syntax-Trans.atomic-abs-tr' (x,T,P)
     in (Syntax.const @bAll) $ x' $ S $ P'
     end
 | bAll-tr' - = raise Match;
  in [(@{const-syntax bEx}, bEx-tr'), (@{const-syntax bAll}, bAll-tr')]
  end
⟫
```

Concrete syntax: set comprehension

syntax
  @*setOfAll* :: [*c*, *idt*, *c*] ⇒ *c*   ((*1*{- : - \in -}))
  @*subsetOf* :: [*idt*, *c*, *c*] ⇒ *c*   ((*1*{- \in - : -}))
syntax (*xsymbols*)
  @*setOfAll* :: [*c*, *idt*, *c*] ⇒ *c*   ((*1*{- : - ∈ -}))
  @*subsetOf* :: [*idt*, *c*, *c*] ⇒ *c*   ((*1*{- ∈ - : -}))
translations
  $\{e : x \in S\}$ ⇌ *CONST setOfAll*(S, $\lambda x.\ e$)
  $\{x \in S : P\}$ ⇌ *CONST subsetOf*(S, $\lambda x.\ P$)

The following definitions make the axioms of set theory more readable. Observe that ∈ is treated as an uninterpreted predicate symbol.

defs
  bChoose-def: *bChoice*(A,P) ≡ *CHOOSE* $x : x \in A \wedge P(x)$
  bEx-def: *bEx*(A, P) ≡ $\exists\, x : x \in A \wedge P(x)$
  bAll-def: *bAll*(A, P) ≡ $\forall\, x : x \in A \Rightarrow P(x)$
  subset-def: $A \subseteq B$ ≡ $\forall\, x \in A : x \in B$

We now state a first batch of axioms of set theory: extensionality and the definitions of *UNION*, *SUBSET*, and *setOfAll*. Membership is also asserted to be produce Boolean values—in traditional presentations of ZF set theory this is ensured by distinguishing sorts of terms and formulas.

axiomatization where
  inIsBool [*intro*!,*simp*]: *isBool*($x \in A$)
and
  extension: $(A = B) \Leftrightarrow (A \subseteq B) \wedge (B \subseteq A)$
and
  UNION: $(A \in \mathit{UNION}\ S) \Leftrightarrow (\exists\, B{\in}S : A \in B)$
and
  SUBSET: $(A \in \mathit{SUBSET}\ S) \Leftrightarrow (A \subseteq S)$
and
  setOfAll: $(y \in \{\ e(x) : x \in S\ \}) \Leftrightarrow (\exists\, x{\in}S : y = e(x))$
and
  subsetOf: $(y \in \{\ x \in S : P(x)\ \}) \Leftrightarrow (y \in S \wedge P(y))$

Armed with this understanding, we can now define the remaining operators of set theory.

defs

upair-def: $upair(a,b) \equiv \{ IF\ x=\{\}\ THEN\ a\ ELSE\ b : x \in SUBSET\ (SUBSET\ \{\}) \}$
cup-def: $A \cup B \equiv UNION\ upair(A,B)$
addElt-def: $addElt(a, A) \equiv upair(a,a) \cup A$
cap-def: $A \cap B \equiv subsetOf(A, \lambda x.\ x \in B)$
diff-def: $A \setminus B \equiv \{x \in A : x \notin B\}$
INTER-def: $INTER\ A \equiv \{ x \in UNION\ A : \forall B \in A : x \in B \}$

The following two axioms complete our presentation of set theory.

axiomatization where
— *infinity* is some infinite set, but it is not uniquely defined.
infinity: $(\{\} \in infinity) \wedge (\forall x \in infinity : \{x\} \cup x \in infinity)$
and
— The foundation axiom rules out sets that are "too big".
foundation: $(A = \{\}) \vee (\exists x \in A : \forall y \in x : y \notin A)$

2.2 Boolean operators

The following lemmas assert that certain operators always return Boolean values; these are helpful for the automated reasoning methods.

lemma *boolifyIn* [*simp*]: $boolify(x \in A) = (x \in A)$
by (*rule inIsBool*[*unfolded isBool-def*])

lemma *notIn-inFalse*: $a \notin A \Longrightarrow (a \in A) = FALSE$
by *auto*

lemma *boolifyBAll* [*simp*]: $boolify(\forall x{\in}A : P(x)) = (\forall x{\in}A : P(x))$
by (*simp add*: *bAll-def*)

lemma *bAllIsBool* [*intro*!,*simp*]: $isBool(\forall x{\in}A : P(x))$
by (*unfold isBool-def*, *rule boolifyBAll*)

lemma *boolifyBEx* [*simp*]: $boolify(\exists x{\in}A : P(x)) = (\exists x{\in}A : P(x))$
by (*simp add*: *bEx-def*)

lemma *bExIsBool* [*intro*!,*simp*]: $isBool(\exists x{\in}A : P(x))$
by (*unfold isBool-def*, *rule boolifyBEx*)

lemma *boolifySubset* [*simp*]: $boolify(A \subseteq B) = (A \subseteq B)$
by (*simp add*: *subset-def*)

lemma *subsetIsBool* [*intro*!,*simp*]: $isBool(A \subseteq B)$
by (*unfold isBool-def*, *rule boolifySubset*)

lemma [*intro*!]:
$[\![isBool(P);\ x{\in}S \Leftrightarrow P]\!] \Longrightarrow (x{\in}S) = P$
$[\![isBool(P);\ P \Leftrightarrow x{\in}S]\!] \Longrightarrow P = (x{\in}S)$
$[\![isBool(P);\ bAll(S,A) \Leftrightarrow P]\!] \Longrightarrow bAll(S,A) = P$

$[\![isBool(P);\ P \Leftrightarrow bAll(S,A)]\!] \Longrightarrow P = bAll(S,A)$
$[\![isBool(P);\ bEx(S,A) \Leftrightarrow P]\!] \Longrightarrow bEx(S,A) = P$
$[\![isBool(P);\ P \Leftrightarrow bEx(S,A)]\!] \Longrightarrow P = bEx(S,A)$
$[\![isBool(P);\ S \subseteq T \Leftrightarrow P]\!] \Longrightarrow (S \subseteq T) = P$
$[\![isBool(P);\ P \Leftrightarrow S \subseteq T]\!] \Longrightarrow P = (S \subseteq T)$
by *auto*

2.3 Substitution rules

lemma *subst-elem* [*trans*]:
assumes $b \in A$ **and** $a=b$
shows $a \in A$
using *assms* **by** *simp*

lemma *subst-elem-rev* [*trans*]:
assumes $a=b$ **and** $b \in A$
shows $a \in A$
using *assms* **by** *simp*

lemma *subst-set* [*trans*]:
assumes $a \in B$ **and** $A=B$
shows $a \in A$
using *assms* **by** *simp*

lemma *subst-set-rev* [*trans*]:
assumes $A=B$ **and** $a \in B$
shows $a \in A$
using *assms* **by** *simp*

2.4 Bounded quantification

lemma *bAllI* [*intro*!]:
assumes $\bigwedge x.\ x \in A \Longrightarrow P(x)$
shows $\forall x{\in}A : P(x)$
using *assms* **unfolding** *bAll-def* **by** *blast*

lemma *bspec* [*dest?*]:
assumes $\forall x{\in}A : P(x)$ **and** $x \in A$
shows $P(x)$
using *assms* **unfolding** *bAll-def* **by** *blast*

lemma *bAllE* [*elim*]:
assumes $\forall x{\in}A : P(x)$ **and** $x \notin A \Longrightarrow Q$ **and** $P(x) \Longrightarrow Q$
shows Q
using *assms* **unfolding** *bAll-def* **by** *blast*

lemma *bAllTriv* [*simp*]: $(\forall x{\in}A : P) = ((\exists x : x \in A) \Rightarrow P)$
unfolding *bAll-def* **by** *blast*

lemma *bAllCong* [*cong*]:
  assumes $A=B$ **and** $\bigwedge x.\ x \in B \Longrightarrow P(x) \Leftrightarrow Q(x)$
  shows $(\forall x{\in}A : P(x)) = (\forall x{\in}B : Q(x))$
using *assms* **by** (*auto simp*: *bAll-def*)

lemma *bExI* [*intro*]:
  assumes $x \in A$ **and** $P(x)$
  shows $\exists x{\in}A : P(x)$
using *assms* **unfolding** *bEx-def* **by** *blast*

lemma *bExCI*: — implicit proof by contradiction
  assumes $(\forall x{\in}A : \neg P(x)) \Longrightarrow P(a)$ **and** $a \in A$
  shows $\exists x{\in}A : P(x)$
using *assms* **by** *blast*

lemma *bExE* [*elim*!]:
  assumes $\exists x{\in}A : P(x)$ **and** $\bigwedge x.\ [\![\ x \in A;\ P(x)\ ]\!] \Longrightarrow Q$
  shows Q
using *assms* **unfolding** *bEx-def* **by** *blast*

lemma *bExTriv* [*simp*]: $(\exists x{\in}A : P) = ((\exists x : x \in A) \wedge P)$
unfolding *bEx-def* **by** *simp*

lemma *bExCong* [*cong*]:
  assumes $A=B$ **and** $\bigwedge x.\ x \in B \Longrightarrow P(x) \Leftrightarrow Q(x)$
  shows $(\exists x{\in}A : P(x)) = (\exists x{\in}B : Q(x))$
using *assms* **unfolding** *bEx-def* **by** *force*

lemma *bChooseI*:
  assumes *1*: $t \in A$ **and** *2*: $P(t)$
  shows $P(CHOOSE\ x \in A : P(x))$
proof −
  let $?ch = CHOOSE\ x \in A : P(x)$
  from *1 2* **have** $t \in A \wedge P(t)$ **..**
  hence $?ch \in A \wedge P(?ch)$
    by (*unfold bChoose-def*, *rule chooseI*)
  thus *?thesis* **..**
qed

lemma *bChooseInSet*:
  assumes *1*: $t \in A$ **and** *2*: $P(t)$
  shows $(CHOOSE\ x \in A : P(x)) \in A$
proof −
  let $?ch = CHOOSE\ x \in A : P(x)$
  from *1 2* **have** $t \in A \wedge P(t)$ **..**
  hence $?ch \in A \wedge P(?ch)$
    by (*unfold bChoose-def*, *rule chooseI*)
  thus *?thesis* **..**

qed

lemma *bChooseI-ex*:
  assumes *hyp*: $\exists\, x{\in}A : P(x)$
  shows $P(CHOOSE\ x{\in}A : P(x))$
proof $-$
  from *hyp* **obtain** x **where** $x \in A$ **and** $P(x)$ **by** *auto*
  thus *?thesis* **by** (*rule bChooseI*)
qed

lemma *bChooseInSet-ex*:
  assumes *hyp*: $\exists\, x{\in}A : P(x)$
  shows $(CHOOSE\ x{\in}A : P(x)) \in A$
proof $-$
  from *hyp* **obtain** x **where** $x \in A$ **and** $P(x)$ **by** *auto*
  thus *?thesis* **by** (*rule bChooseInSet*)
qed

lemma *bChooseI2*:
  assumes *1*: $\exists\, x{\in}A : P(x)$ **and** *2*: $\bigwedge x.\ [\![x \in A;\ P(x)]\!] \Longrightarrow Q(x)$
  shows $Q(CHOOSE\ x{\in}A : P(x))$
proof (*rule 2*)
  from *1* **show** $(CHOOSE\ x{\in}A : P(x)) \in A$ **by** (*rule bChooseInSet-ex*)
next
  from *1* **show** $P(CHOOSE\ x{\in}A : P(x))$ **by** (*rule bChooseI-ex*)
qed

lemma *bChooseCong* [*cong*]:
  assumes $A{=}B$ **and** $\bigwedge x.\ x \in B \Longrightarrow P(x) \Leftrightarrow Q(x)$
  shows $(CHOOSE\ x{\in}A : P(x)) = (CHOOSE\ x{\in}B : Q(x))$
unfolding *bChoose-def* **proof** (*rule choose-det*)
  fix x
  from *assms* **show** $x \in A \wedge P(x) \Leftrightarrow x \in B \wedge Q(x)$ **by** *blast*
qed

2.5 Simplification of conditional expressions

lemma *inCond* [*simp*]: $(a \in (IF\ P\ THEN\ S\ ELSE\ T)) = ((P \wedge a \in S) \vee (\neg P \wedge a \in T))$
by (*force intro*: *condI elim*: *condE*)

lemma *condIn* [*simp*]: $((IF\ P\ THEN\ a\ ELSE\ b) \in S) = ((P \wedge a \in S) \vee (\neg P \wedge b \in S))$
by (*force intro*: *condI elim*: *condE*)

lemma *inCondI* :

  assumes $P \Longrightarrow c \in A$ **and** $\neg P \Longrightarrow c \in B$
  shows $c \in (IF\ P\ THEN\ A\ ELSE\ B)$

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

lemma *condInI* :

assumes $P \Longrightarrow a \in S$ **and** $\neg P \Longrightarrow b \in S$
shows $(IF\ P\ THEN\ a\ ELSE\ b) \in S$
using *assms* **by** *auto*

lemma *inCondE*:
assumes $c \in (IF\ P\ THEN\ A\ ELSE\ B)$
and $[\![\ P;\ c \in A\ ]\!] \Longrightarrow Q$ **and** $[\![\ \neg P;\ c \in B\ ]\!] \Longrightarrow Q$
shows Q
using *assms* **by** *auto*

lemma *condInE*:
assumes $(IF\ P\ THEN\ a\ ELSE\ b) \in S$
and $[\![\ P;\ a \in S\ ]\!] \Longrightarrow Q$ **and** $[\![\ \neg P;\ b \in S\ ]\!] \Longrightarrow Q$
shows Q
using *assms* **by** *auto*

lemma *subsetCond* [*simp*]:
$(A \subseteq (IF\ P\ THEN\ S\ ELSE\ T)) = ((P \wedge A \subseteq S) \vee (\neg P \wedge A \subseteq T))$
by (*blast intro*: *condI elim*: *condE*)

lemma *condSubset* [*simp*]:
$((IF\ P\ THEN\ A\ ELSE\ B) \subseteq S) = ((P \wedge A \subseteq S) \vee (\neg P \wedge B \subseteq S))$
by (*force intro*: *condI elim*: *condE*)

2.6 Rules for subsets and set equality

lemma *subsetI* [*intro*!]:
assumes $\bigwedge x.\ x \in A \Longrightarrow x \in B$
shows $A \subseteq B$
using *assms* **by** (*auto simp*: *subset-def*)

lemma *subsetD* [*elim*,*trans*]:
assumes $A \subseteq B$ **and** $c \in A$
shows $c \in B$
using *assms* **by** (*auto simp*: *subset-def*)

lemma *rev-subsetD* [*trans*]:
$[\![\ c \in A;\ A \subseteq B\ ]\!] \Longrightarrow c \in B$
by (*rule subsetD*)

lemma *subsetCE* [*elim*]: — elimination rule for classical logic
assumes $A \subseteq B$ **and** $c \notin A \Longrightarrow P$ **and** $c \in B \Longrightarrow P$
shows P
using *assms* **unfolding** *subset-def* **by** *blast*

lemma *subsetE*:
  assumes $A \subseteq B$ **and** $\bigwedge x.\ (x \in A \Longrightarrow x \in B) \Longrightarrow P$
  shows P
using *assms* **by** *blast*

lemma *subsetNotIn*:
  assumes $A \subseteq B$ **and** $c \notin B$
  shows $c \notin A$
using *assms* **by** *blast*

lemma *rev-subsetNotIn*: $[\![\ c \notin B;\ A \subseteq B\ ]\!] \Longrightarrow c \notin A$
by (*rule subsetNotIn*)

lemma *notSubset*: $(\neg(A \subseteq B)) = (\exists\, x \in A : x \notin B)$
by *blast*

lemma *notSubsetI* :
  assumes $a \in A$ **and** $a \notin B$
  shows $\neg(A \subseteq B)$
using *assms* **by** *blast*

lemma *notSubsetE* :
  assumes $\neg(A \subseteq B)$ **and** $\bigwedge x.\ [\![\ x \in A;\ x \notin B\ ]\!] \Longrightarrow P$
  shows P
using *assms* **by** *blast*

The subset relation is a partial order.

lemma *subsetRefl* [*simp*,*intro*!]: $A \subseteq A$
by *blast*

lemma *subsetTrans* [*trans*]:
  assumes $A \subseteq B$ **and** $B \subseteq C$
  shows $A \subseteq C$
using *assms* **by** *blast*

lemma *setEqual*:
  assumes $A \subseteq B$ **and** $B \subseteq A$
  shows $A = B$
using *assms* **by** (*intro iffD2*[*OF extension*], *blast*)

lemma *setEqualI*:
  assumes $\bigwedge x.\ x \in A \Longrightarrow x \in B$ **and** $\bigwedge x.\ x \in B \Longrightarrow x \in A$
  shows $A = B$
by (*rule setEqual*, (*blast intro*: *assms*)+)

The rule *setEqualI* is too general for use as a default introduction rule: we don't want to apply it for Booleans, for example. However, instances where at least one term results from a set constructor are useful.

lemmas

setEqualI [**where** $A = addElt(a,C)$, *standard*, *intro*!]
setEqualI [**where** $B = addElt(a,C)$, *standard*, *intro*!]
setEqualI [**where** $A = SUBSET\ C$, *standard*, *intro*!]
setEqualI [**where** $B = SUBSET\ C$, *standard*, *intro*!]
setEqualI [**where** $A = UNION\ C$, *standard*, *intro*!]
setEqualI [**where** $B = UNION\ C$, *standard*, *intro*!]
setEqualI [**where** $A = INTER\ C$, *standard*, *intro*!]
setEqualI [**where** $B = INTER\ C$, *standard*, *intro*!]
setEqualI [**where** $A = C \cup D$, *standard*, *intro*!]
setEqualI [**where** $B = C \cup D$, *standard*, *intro*!]
setEqualI [**where** $A = C \cap D$, *standard*, *intro*!]
setEqualI [**where** $B = C \cap D$, *standard*, *intro*!]
setEqualI [**where** $A = C \setminus D$, *standard*, *intro*!]
setEqualI [**where** $B = C \setminus D$, *standard*, *intro*!]
setEqualI [**where** $A = subsetOf(S,P)$, *standard*, *intro*!]
setEqualI [**where** $B = subsetOf(S,P)$, *standard*, *intro*!]
setEqualI [**where** $A = setOfAll(S,e)$, *standard*, *intro*!]
setEqualI [**where** $B = setOfAll(S,e)$, *standard*, *intro*!]

lemmas *setEqualD1* = *extension*[*THEN iffD1*, *THEN conjD1*, *standard*] — $A = B \Longrightarrow A \subseteq B$
lemmas *setEqualD2* = *extension*[*THEN iffD1*, *THEN conjD2*, *standard*] — $A = B \Longrightarrow B \subseteq A$

We declare the elimination rule for set equalities as an unsafe rule to use with the classical reasoner, so it will be tried if the more obvious uses of equality fail.

lemma *setEqualE* [*elim*]:
assumes $A = B$
and $[\![c \in A;\ c \in B]\!] \Longrightarrow P$ **and** $[\![c \notin A;\ c \notin B]\!] \Longrightarrow P$
shows P
using *assms* **by** (*blast dest*: *setEqualD1 setEqualD2*)

lemma *setEqual-iff*: $(A = B) = (\forall x : x \in A \Leftrightarrow x \in B)$
by (*blast intro*: *setEqualI*)

2.7 Set comprehension: *setOfAll* and *subsetOf*

lemma *setOfAllI* :
assumes $\exists x \in S : a = e(x)$
shows $a \in \{ e(x) : x \in S \}$
using *assms* **by** (*blast intro*: *iffD2*[*OF setOfAll*])

lemma *setOfAll-eqI* [*intro*]:
assumes $a = e(x)$ **and** $x \in S$
shows $a \in \{ e(x) : x \in S \}$
using *assms* **by** (*blast intro*: *setOfAllI*)

lemma *setOfAllE* [*elim*!]:
  assumes $a \in \{ e(x) : x \in S \}$ **and** $\bigwedge x.\ [\![x \in S;\ e(x) = a]\!] \Longrightarrow P$
  shows P
using *assms* **by** (*blast dest*: *iffD1*[*OF setOfAll*])

lemma *setOfAll-iff* [*simp*]:
  $(a \in \{ e(x) : x \in S \}) = (\exists\, x{\in}S : a = e(x))$
by *blast*

lemma *setOfAll-triv* [*simp*]: $\{ x : x \in S \} = S$
by *blast*

lemma *setOfAll-cong* :
  assumes $S = T$ **and** $\bigwedge x.\ x \in T \Longrightarrow e(x) = f(x)$
  shows $\{ e(x) : x \in S \} = \{ f(y) : y \in T \}$
using *assms* **by** *auto*

The following rule for showing equality of sets defined by comprehension is probably too general to use by default with the automatic proof methods.

lemma *setOfAllEqual*:
  assumes $\bigwedge x.\ x \in S \Longrightarrow \exists\, y{\in}T : e(x) = f(y)$
  and $\bigwedge y.\ y \in T \Longrightarrow \exists\, x{\in}S : f(y) = e(x)$
  shows $\{ e(x) : x \in S \} = \{ f(y) : y \in T \}$
using *assms* **by** *auto*

lemma *subsetOfI* [*intro*!]:
  assumes $a \in S$ **and** $P(a)$
  shows $a \in \{ x \in S : P(x) \}$
using *assms* **by** (*blast intro*: *iffD2*[*OF subsetOf*])

lemma *subsetOfE* [*elim*!]:
  assumes $a \in \{ x \in S : P(x) \}$ **and** $[\![a \in S;\ P(a)]\!] \Longrightarrow Q$
  shows Q
using *assms* **by** (*blast dest*: *iffD1*[*OF subsetOf*])

lemma *subsetOfD1*:
  assumes $a \in \{ x \in S : P(x) \}$
  shows $a \in S$
using *assms* **by** *blast*

lemma *subsetOfD2*:
  assumes $a \in \{ x \in S : P(x) \}$
  shows $P(a)$
using *assms* **by** *blast*

lemma *subsetOf-iff* [*simp*]:
  $(a \in \{ x \in S : P(x) \}) = (a \in S \wedge P(a))$
by *blast*

lemma *subsetOf-cong*:
assumes $S = T$ **and** $\bigwedge x.\ x \in T \Longrightarrow P(x) \Leftrightarrow Q(x)$
shows $\{x \in S : P(x)\} = \{y \in T : Q(y)\}$
using *assms* **by** *blast*

lemma *subsetOfEqual*:
assumes $\bigwedge x.\ [\![\ x \in S;\ P(x)\ ]\!] \Longrightarrow x \in T$
and $\bigwedge x.\ [\![\ x \in S;\ P(x)\ ]\!] \Longrightarrow Q(x)$
and $\bigwedge y.\ [\![\ y \in T;\ Q(y)\ ]\!] \Longrightarrow y \in S$
and $\bigwedge y.\ [\![\ y \in T;\ Q(y)\ ]\!] \Longrightarrow P(y)$
shows $\{x \in S : P(x)\} = \{y \in T : Q(y)\}$
by (*safe elim*!: *assms*)

2.8 *UNION* – basic rules for generalized union

lemma *UNIONI* [*intro*]:
assumes $B \in C$ **and** $a \in B$
shows $a \in UNION\ C$
using *assms* **by** (*blast intro*: *UNION*[*THEN iffD2*])

lemma *UNIONE* [*elim*!]:
assumes $a \in UNION\ C$ **and** $\bigwedge B.\ [\![\ a \in B;\ B \in C\ ]\!] \Longrightarrow P$
shows P
using *assms* **by** (*blast dest*: *iffD1* [*OF UNION*])

lemma *UNION-iff* [*simp*]:
$(a \in UNION\ C) = (\exists\, B{\in}C : a \in B)$
by *blast*

2.9 The empty set

Proving that the empty set has no elements is a bit tricky. We first show that the set $\{x \in \{\} : FALSE\}$ is empty and then use the foundation axiom to show that it equals the empty set.

lemma *emptysetEmpty*: $a \notin \{\}$
proof
assume *a*: $a \in \{\}$
let $?empty = \{\ x \in \{\} : FALSE\ \}$
from *foundation*[**where** $A{=}?empty$]
have $\{\} = ?empty$ **by** *blast*
from *this a* **have** $a \in ?empty$ **by** (*rule subst*)
thus *FALSE* **by** *blast*
qed

$a \in \{\} \Longrightarrow P$

lemmas *emptyE* [*elim*!] = *emptysetEmpty*[*THEN notE*, *standard*]

lemma [*simp*]: $(a \in \{\}) = FALSE$

by *blast*

lemma *emptysetI* [*intro*!]:
  assumes $\forall\, x : x \notin A$
  shows $A = \{\}$
using *assms* **by** (*blast intro*: *setEqualI*)

lemmas *emptysetI-rev* [*intro*!] = *sym*[*OF emptysetI*]

lemma *emptysetIffEmpty* : $(A = \{\}) = (\forall\, x : x \notin A)$
by *blast*

lemma *emptysetIffEmpty'* : $(\{\} = A) = (\forall\, x : x \notin A)$
by *blast*

lemma *nonEmpty* [*simp*]:
  $(A \neq \{\}) = (\exists\, x : x \in A)$
  $(\{\} \neq A) = (\exists\, x : x \in A)$
by (*blast*+)

Complete critical pairs

lemmas *nonEmpty'* [*simp*] = *nonEmpty*[*simplified*]
— $((A = \{\}) = \mathit{FALSE}) = (\exists\, x : x \in A)$, $((\{\} = A) = \mathit{FALSE}) = (\exists\, x : x \in A)$

lemma *emptysetD* :
  assumes $A = \{\}$
  shows $x \notin A$
using *assms* **by** *blast*

lemma *emptySubset* [*iff*]: $\{\} \subseteq A$
by *blast*

lemma *nonEmptyI*:
  assumes $a \in A$
  shows $A \neq \{\}$
using *assms* **by** *blast*

lemma *nonEmptyE*:
  assumes $A \neq \{\}$ **and** $\bigwedge x.\ x \in A \Longrightarrow P$
  shows P
using *assms* **by** *blast*

lemma *subsetEmpty* [*simp*]: $(A \subseteq \{\}) = (A = \{\})$
by *blast*

lemma [*simp*]:
  $\mathit{bAll}(\{\}, P) = \mathit{TRUE}$

$bEx(\{\}, P) = FALSE$
by $(blast+)$

2.10 *SUBSET* – the powerset operator

lemma *SUBSETI* [*intro*!]:
  assumes $A \subseteq B$
  shows $A \in SUBSET\ B$
using *assms* **by** (*blast intro*: *iffD2*[*OF SUBSET*])

lemma *SUBSETD* [*dest*!]:
  assumes $A \in SUBSET\ B$
  shows $A \subseteq B$
using *assms* **by** (*blast dest*: *iffD1*[*OF SUBSET*])

lemma *SUBSET-iff* [*simp*]:
  $(A \in SUBSET\ B) = (A \subseteq B)$
by *blast*

lemmas *emptySUBSET* = *emptySubset*[*THEN SUBSETI*, *standard*] — $\{\} \in SUBSET\ A$
lemmas *selfSUBSET* = *subsetRefl*[*THEN SUBSETI*, *standard*] — $A \in SUBSET\ A$

2.11 *INTER* – basic rules for generalized intersection

Generalized intersection is not officially part of TLA$^+$ but can easily be defined as above. Observe that the rules are not exactly dual to those for *UNION* because the intersection of the empty set is defined to be the empty set.

lemma *INTERI* [*intro*]:
  assumes $\bigwedge B.\ B \in C \Longrightarrow a \in B$ **and** $\exists\, B : B \in C$
  shows $a \in INTER\ C$
using *assms* **unfolding** *INTER-def* **by** *blast*

lemma *INTERE* [*elim*]:
  assumes $a \in INTER\ C$ **and** $[\![\ a \in UNION\ C;\ \bigwedge B.\ B \in C \Longrightarrow a \in B\ ]\!] \Longrightarrow P$
  shows P
using *assms* **unfolding** *INTER-def* **by** *blast*

lemma *INTER-iff* [*simp*]:
  $(a \in INTER\ C) = (a \in UNION\ C \land (\forall\, B{\in}C : a \in B))$
by *blast*

2.12 Binary union, intersection, and difference: basic rules

We begin by proving some lemmas about the auxiliary pairing operator *upair*. None of these theorems is active by default, as the operator is not part of TLA$^+$ and should not occur in actual reasoning. The dependencies between these operators are quite tricky, therefore the order of the first few lemmas in this section is tightly constrained.

lemma *upairE*:
  assumes $x \in upair(a,b)$ **and** $x=a \Longrightarrow P$ **and** $x=b \Longrightarrow P$
  shows P
using *assms* **by** (*auto simp*: *upair-def*)

lemma *cupE* [*elim*!]:
  assumes $x \in A \cup B$ **and** $x \in A \Longrightarrow P$ **and** $x \in B \Longrightarrow P$
  shows P
using *assms* **by** (*auto simp*: *cup-def elim*: *upairE*)

lemma *upairI1*: $a \in upair(a,b)$
by (*auto simp*: *upair-def*)

lemma *singleton-iff* [*simp*]: $(a \in \{b\}) = (a = b)$
proof *auto*
  assume *a*: $a \in \{b\}$
  thus $a = b$
    by (*auto simp*: *addElt-def elim*: *upairE*)
next
  show $b \in \{b\}$
    by (*auto simp*: *addElt-def cup-def intro*: *upairI1*)
qed

lemma *singletonI* : $a \in \{a\}$
by *simp*

lemma *upairI2*: $b \in upair(a,b)$
by (*auto simp*: *upair-def*)

lemma *upair-iff*: $(c \in upair(a,b)) = (c=a \vee c=b)$
by (*blast intro*: *upairI1 upairI2 elim*: *upairE*)

lemma *cupI1*:
  assumes $a \in A$
  shows $a \in A \cup B$
using *assms* **by** (*auto simp*: *cup-def upair-iff*)

lemma *cupI2*:
  assumes $a \in B$
  shows $a \in A \cup B$
using *assms* **by** (*auto simp*: *cup-def upair-iff*)

lemma *cup-iff* [*simp*]: $(c \in A \cup B) = (c \in A \vee c \in B)$
by (*auto simp*: *cup-def upair-iff*)

lemma *cupCI* [*intro*!]:

assumes $c \notin B \Longrightarrow c \in A$
shows $c \in A \cup B$
using *assms* **by** *auto*

lemma *addElt-iff* [*simp*]: $(x \in addElt(a,A)) = (x = a \vee x \in A)$
by (*auto simp*: *addElt-def upair-iff*)

lemma *addEltI* [*intro*!]:
assumes $x \neq a \Longrightarrow x \in A$
shows $x \in addElt(a,A)$
using *assms* **by** *auto*

lemma *addEltE* [*elim*!]:
assumes $x \in addElt(a,A)$ **and** $x=a \Longrightarrow P$ **and** $x \in A \Longrightarrow P$
shows P
using *assms* **by** *auto*

lemma *addEltSubsetI*:
assumes $a \in B$ **and** $A \subseteq B$
shows $addElt(a,A) \subseteq B$
using *assms* **by** *blast*

lemma *addEltSubsetE* [*elim*]:
assumes $addElt(a,A) \subseteq B$ **and** $\llbracket a \in B; A \subseteq B \rrbracket \Longrightarrow P$
shows P
using *assms* **by** *blast*

lemma *addEltSubset-iff*: $(addElt(a,A) \subseteq B) = (a \in B \wedge A \subseteq B)$
by *blast*

— Adding the two following lemmas to the simpset breaks proofs.
lemma *addEltEqual-iff*: $(addElt(a,A) = S) = (a \in S \wedge A \subseteq S \wedge S \subseteq addElt(a,A))$
by *blast*

lemma *equalAddElt-iff*: $(S = addElt(a,A)) = (a \in S \wedge A \subseteq S \wedge S \subseteq addElt(a,A))$
by *blast*

lemma *addEltEqualAddElt*:
$(addElt(a,A) = addElt(b,B)) =$
$(a \in addElt(b,B) \wedge A \subseteq addElt(b,B) \wedge b \in addElt(a,A) \wedge B \subseteq addElt(a,A))$
by (*auto simp*: *addEltEqual-iff*)

lemma *cap-iff* [*simp*]: $(c \in A \cap B) = (c \in A \wedge c \in B)$
by (*simp add*: *cap-def*)

lemma *capI* [*intro*!]:
assumes $c \in A$ **and** $c \in B$
shows $c \in A \cap B$

using *assms* **by** *simp*

lemma *capD1*:
  assumes $c \in A \cap B$
  shows $c \in A$
using *assms* **by** *simp*

lemma *capD2*:
  assumes $c \in A \cap B$
  shows $c \in B$
using *assms* **by** *simp*

lemma *capE* [*elim*!]:
  assumes $c \in A \cap B$ **and** $[\![\ c \in A;\ c \in B\ ]\!] \Longrightarrow P$
  shows P
using *assms* **by** *simp*

lemma *diff-iff* [*simp*]: $(c \in A \setminus B) = (c \in A \wedge c \notin B)$
by (*simp add*: *diff-def*)

lemma *diffI* [*intro*!]:
  assumes $c \in A$ **and** $c \notin B$
  shows $c \in A \setminus B$
using *assms* **by** *simp*

lemma *diffD1*:
  assumes $c \in A \setminus B$
  shows $c \in A$
using *assms* **by** *simp*

lemma *diffD2*:
  assumes $c \in A \setminus B$
  shows $c \notin B$
using *assms* **by** *simp*

lemma *diffE* [*elim*!]:
  assumes $c \in A \setminus B$ **and** $[\![\ c \in A;\ c \notin B\ ]\!] \Longrightarrow P$
  shows P
using *assms* **by** *simp*

lemma *subsetAddElt-iff*:
  $(B \subseteq addElt(a,A)) = (B \subseteq A \vee (\exists\, C \in SUBSET\ A : B = addElt(a,C)))$
  (**is** *?lhs* = *?rhs*)
proof −
  have *?lhs* ⇒ *?rhs*
  proof
    assume *1*: *?lhs* **show** *?rhs*
    proof (*cases* $a \in B$)
      case *True*

from *1* **have** $B \setminus \{a\} \in$ *SUBSET A* **by** *blast*
moreover
from *True 1* **have** $B = addElt(a,\ B \setminus \{a\})$ **by** *blast*
ultimately
show *?thesis* **by** *blast*
next
case *False*
with *1* **show** *?thesis* **by** *blast*
qed
qed
moreover
have *?rhs* $\Rightarrow$ *?lhs* **by** *blast*
ultimately show *?thesis* **by** *blast*
qed

lemma *subsetAddEltE* [*elim*]:
assumes $B \subseteq addElt(a,A)$ **and** $B \subseteq A \Longrightarrow P$ **and** $\bigwedge C.\ [\![\ C \subseteq A;\ B = addElt(a,C)\ ]\!] \Longrightarrow P$
shows P
using *assms* **by** (*auto simp*: *subsetAddElt-iff*)

2.13 Consequences of the foundation axiom

lemma *inAsym*:
assumes *hyps*: $a \in b\ \neg P \Longrightarrow b \in a$
shows P
proof (*rule contradiction*)
assume $\neg P$
with *foundation*[**where** $A=\{a,b\}$] *hyps* **show** *FALSE* **by** *blast*
qed

lemma *inIrrefl*:
assumes $a \in a$
shows P
using *assms* **by** (*rule inAsym*, *blast*)

lemma *inNonEqual*:
assumes $a \in A$
shows $a \neq A$
using *assms* **by** (*blast elim*: *inIrrefl*)

lemma *equalNotIn*:
assumes $A = B$
shows $A \notin B$
using *assms* **by** (*blast elim*: *inIrrefl*)

2.14 Miniscoping of bounded quantifiers

lemma *miniscope-bAll* [*simp*]:
$\bigwedge P\ Q.\ (\forall x{\in}A : P(x) \wedge Q) = ((\forall x{\in}A : P(x)) \wedge (A = \{\} \vee Q))$

$\bigwedge P\ Q.\ (\forall x{\in}A : P \wedge Q(x)) = ((A = \{\} \vee P) \wedge (\forall x{\in}A : Q(x)))$
$\bigwedge P\ Q.\ (\forall x{\in}A : P(x) \vee Q) = ((\forall x{\in}A : P(x)) \vee Q)$
$\bigwedge P\ Q.\ (\forall x{\in}A : P \vee Q(x)) = (P \vee (\forall x{\in}A : Q(x)))$
$\bigwedge P\ Q.\ (\forall x{\in}A : P(x) \Rightarrow Q) = ((\exists x{\in}A : P(x)) \Rightarrow Q)$
$\bigwedge P\ Q.\ (\forall x{\in}A : P \Rightarrow Q(x)) = (P \Rightarrow (\forall x{\in}A : Q(x)))$
$\bigwedge P.\ (\neg(\forall x{\in}A : P(x))) = (\exists x{\in}A : \neg P(x))$
$\bigwedge P.\ (\forall x \in \mathit{addElt}(a,A) : P(x)) = (P(a) \wedge (\forall x{\in}A : P(x)))$
$\bigwedge P.\ (\forall x \in \mathit{UNION}\ A : P(x)) = (\forall B \in A : \forall x{\in}B : P(x))$
$\bigwedge P.\ (\forall x \in \{e(y) : y{\in}A\} : P(x)) = (\forall y{\in}A : P(e(y)))$
$\bigwedge P\ Q.\ (\forall x \in \{y \in A : P(y)\} : Q(x)) = (\forall y{\in}A\!: P(y) \Rightarrow Q(y))$
by (*blast+*)

lemma *bAllCup* [*simp*]:
$(\forall x \in A \cup B : P(x)) = ((\forall x{\in}A : P(x)) \wedge (\forall x{\in}B : P(x)))$
by *blast*

lemma *bAllCap* [*simp*]:
$(\forall x \in A \cap B : P(x)) = (\forall x{\in}A : x \in B \Rightarrow P(x))$
by *blast*

lemma *miniscope-bEx* [*simp*]:
$\bigwedge P\ Q.\ (\exists x{\in}A : P(x) \wedge Q) = ((\exists x{\in}A : P(x)) \wedge Q)$
$\bigwedge P\ Q.\ (\exists x{\in}A : P \wedge Q(x)) = (P \wedge (\exists x{\in}A : Q(x)))$
$\bigwedge P\ Q.\ (\exists x{\in}A : P(x) \vee Q) = ((\exists x{\in}A : P(x)) \vee (A \neq \{\} \wedge Q))$
$\bigwedge P\ Q.\ (\exists x{\in}A : P \vee Q(x)) = ((A \neq \{\} \wedge P) \vee (\exists x{\in}A : Q(x)))$
$\bigwedge P\ Q.\ (\exists x{\in}A : P(x) \Rightarrow Q) = ((\forall x{\in}A : P(x)) \Rightarrow (A \neq \{\} \wedge Q))$
$\bigwedge P\ Q.\ (\exists x{\in}A : P \Rightarrow Q(x)) = ((A{=}\{\} \vee P) \Rightarrow (\exists x{\in}A : Q(x)))$
$\bigwedge P.\ (\exists x \in \mathit{addElt}(a,A) : P(x)) = (P(a) \vee (\exists x{\in}A : P(x)))$
$\bigwedge P.\ (\neg(\exists x{\in}A : P(x))) = (\forall x{\in}A : \neg P(x))$
$\bigwedge P.\ (\exists x \in \mathit{UNION}\ A : P(x)) = (\exists B \in A : \exists x{\in}B : P(x))$
$\bigwedge P.\ (\exists x \in \{e(y) : y \in S\} : P(x)) = (\exists y{\in}S : P(e(y)))$
$\bigwedge P\ Q.\ (\exists x \in \{y \in S : P(y)\} : Q(x)) = (\exists y{\in}S : P(y) \wedge Q(y))$
by (*blast+*)

— completing critical pairs for negated assumption
lemma *notbQuant'* [*simp*]:
$\bigwedge P.\ ((\forall x{\in}A : P(x)) = \mathit{FALSE}) = (\exists x{\in}A : \neg P(x))$
$\bigwedge P.\ ((\exists x{\in}A : P(x)) = \mathit{FALSE}) = (\forall x{\in}A : \neg P(x))$
by (*auto simp*: *miniscope-bAll*[*simplified*] *miniscope-bEx*[*simplified*])

lemma *bExistsCup* [*simp*]:
$(\exists x \in A \cup B : P(x)) = ((\exists x{\in}A : P(x)) \vee (\exists x{\in}B : P(x)))$
by *blast*

lemma *bExistsCap* [*simp*]:
$(\exists x \in A \cap B : P(x)) = (\exists x{\in}A : x \in B \wedge P(x))$
by *blast*

lemma *bQuant-distribs*: — not active by default

$(\forall x{\in}A : P(x) \wedge Q(x)) = ((\forall x{\in}A : P(x)) \wedge (\forall x{\in}A : Q(x)))$
$(\exists x{\in}A : P(x) \vee Q(x)) = ((\exists x{\in}A : P(x)) \vee (\exists x{\in}A : Q(x)))$
by (*blast+*)

lemma *bQuantOnePoint* [*simp*]:
$(\exists x{\in}A : x{=}a) = (a \in A)$
$(\exists x{\in}A : a{=}x) = (a \in A)$
$(\exists x{\in}A : x{=}a \wedge P(x)) = (a \in A \wedge P(a))$
$(\exists x{\in}A : a{=}x \wedge P(x)) = (a \in A \wedge P(a))$
$(\forall x{\in}A : x{=}a \Rightarrow P(x)) = (a{\in}A \Rightarrow P(a))$
$(\forall x{\in}A : a{=}x \Rightarrow P(x)) = (a{\in}A \Rightarrow P(a))$
by (*blast+*)

2.15 Simplification of set comprehensions

lemma *comprehensionSimps* [*simp*]:
$\bigwedge e.\ setOfAll(\{\}, e) = \{\}$
$\bigwedge P.\ subsetOf(\{\}, P) = \{\}$
$\bigwedge A\ P.\ \{\ x \in A : P\ \} = (IF\ P\ THEN\ A\ ELSE\ \{\})$
$\bigwedge A\ a\ e.\ \{\ e(x) : x \in addElt(a,A)\ \} = addElt(e(a), \{\ e(x) : x \in A\ \})$
$\bigwedge A\ a\ P.\ \{\ x \in addElt(a,A) : P(x)\ \} = (IF\ P(a)\ THEN\ addElt(a, \{x \in A : P(x)\})\ ELSE\ \{x \in A : P(x)\})$
$\bigwedge e\ f.\ \{\ e(x) : x \in \{\ f(y) : y \in A\ \}\ \} = \{\ e(f(y)) : y \in A\ \}$
$\bigwedge P\ Q\ A.\ \{\ x \in \{y{\in}A : P(y)\} : Q(x)\ \} = \{\ x \in A : P(x) \wedge Q(x)\ \}$
$\bigwedge P\ A.\ subsetOf(A, \lambda x.\ x \in \{\ y \in B : Q(y)\ \}) = \{\ x \in A \cap B : Q(x)\ \}$
$\bigwedge e\ A\ B.\ subsetOf(A, \lambda x.\ x \in \{\ e(y) : y \in B\ \}) = \{\ e(y) : y \in B\ \} \cap A$

$\bigwedge A\ e.\ setOfAll(A, \lambda x.\ e) = (IF\ A = \{\}\ THEN\ \{\}\ ELSE\ \{e\})$
by *auto*

The following are not active by default.

lemma *comprehensionDistribs*:
$\bigwedge e.\ \{\ e(x) : x \in A \cup B\ \} = \{e(x) : x{\in}A\} \cup \{e(x) : x{\in}B\}$
$\bigwedge P.\ \{\ x \in A \cup B : P(x)\ \} = \{x{\in}A : P(x)\} \cup \{x{\in}B : P(x)\}$
— *setOfAll* and intersection or difference do not distribute
$\bigwedge P.\ \{\ x \in A \cap B : P(x)\ \} = \{x{\in}A : P(x)\} \cap \{x{\in}B : P(x)\}$
$\bigwedge P.\ \{\ x \in A \setminus B : P(x)\ \} = \{x{\in}A : P(x)\} \setminus \{x{\in}B : P(x)\}$
by (*blast+*)

2.16 Binary union, intersection, and difference: inclusions and equalities

The following list contains many simple facts about set theory. Only the most trivial of these are included in the default set of rewriting rules.

lemma *addEltCommute*: $addElt(a, addElt(b, C)) = addElt(b, addElt(a, C))$
by *blast*

lemma *addEltAbsorb*: $a \in A \Longrightarrow addElt(a,A) = A$

by *blast*

lemma *addEltTwice*: $addElt(a,\ addElt(a,\ A)) = addElt(a,\ A)$
by *blast*

lemma *capAddEltLeft*: $addElt(a,B) \cap C = (IF\ a \in C\ THEN\ addElt(a,\ B \cap C)\ ELSE\ B \cap C)$
by (*blast intro*: *condI elim*: *condE*)

lemma *capAddEltRight*: $C \cap addElt(a,B) = (IF\ a \in C\ THEN\ addElt(a,\ C \cap B)\ ELSE\ C \cap B)$
by (*blast intro*: *condI elim*: *condE*)

lemma *addEltCap*: $addElt(a,\ B \cap C) = addElt(a,B) \cap addElt(a,C)$
by *blast*

lemma *diffAddEltLeft*: $addElt(a,B) \setminus C = (IF\ a \in C\ THEN\ B \setminus C\ ELSE\ addElt(a,\ B \setminus C))$
by (*blast intro*: *condI elim*: *condE*)

lemma *capSubset*: $(C \subseteq A \cap B) = (C \subseteq A \wedge C \subseteq B)$
by *blast*

lemma *capLB1*: $A \cap B \subseteq A$
by *blast*

lemma *capLB2*: $A \cap B \subseteq B$
by *blast*

lemma *capGLB*:
  assumes $C \subseteq A$ **and** $C \subseteq B$
  shows $C \subseteq A \cap B$
using *assms* **by** *blast*

lemma *capEmpty* [*simp*]:
  $A \cap \{\} = \{\}$
  $\{\} \cap A = \{\}$
by *blast+*

lemma *capAbsorb* [*simp*]: $A \cap A = A$
by *blast*

lemma *capLeftAbsorb*: $A \cap (A \cap B) = A \cap B$
by *blast*

lemma *capCommute*: $A \cap B = B \cap A$
by *blast*

lemma *capLeftCommute*: $A \cap (B \cap C) = B \cap (A \cap C)$

by *blast*

lemma *capAssoc*: $(A \cap B) \cap C = A \cap (B \cap C)$
by *blast*

Intersection is an AC operator: can be added to simp where appropriate

lemmas *capAC* = *capAssoc capCommute capLeftCommute capLeftAbsorb*

lemma *subsetOfCap*: $\{x{\in}A : P(x)\} \cap B = \{x \in A \cap B : P(x)\}$
by *blast*

lemma *capSubsetOf*:
$B \cap \{x{\in}A : P(x)\} = \{x \in B{\cap}A : P(x)\}$
by *blast*

lemma *subsetOfDisj*:
$\{x{\in}A : P(x) \vee Q(x)\} = \{x{\in}A : P(x)\} \cup \{x{\in}A : Q(x)\}$
by *blast*

lemma *subsetOfConj*:
$\{x{\in}A : P(x) \wedge Q(x)\} = \{x{\in}A : P(x)\} \cap \{x{\in}A : Q(x)\}$
by *blast*

lemma *subsetCup*: $(A \cup B \subseteq C) = (A \subseteq C \wedge B \subseteq C)$
by *blast*

lemma *cupUB1*: $A \subseteq A \cup B$
by *blast*

lemma *cupUB2*: $B \subseteq A \cup B$
by *blast*

lemma *cupLUB*:
assumes $A \subseteq C$ **and** $B \subseteq C$
shows $A \cup B \subseteq C$
using *assms* **by** *blast*

lemma *cupEmpty* [*simp*]:
$A \cup \{\} = A$
$\{\} \cup A = A$
by *blast+*

lemma *cupAddEltLeft*: $addElt(a,B) \cup C = addElt(a, B \cup C)$
by *blast*

lemma *cupAddEltRight*: $C \cup addElt(a,B) = addElt(a, C \cup B)$
by *blast*

lemma *addEltCup*: $addElt(a, B \cup C) = addElt(a,B) \cup addElt(a,C)$

by *blast*

lemma *cupAbsorb* [*simp*]: $A \cup A = A$
by *blast*

lemma *cupLeftAbsorb*: $A \cup (A \cup B) = A \cup B$
by *blast*

lemma *cupCommute*: $A \cup B = B \cup A$
by *blast*

lemma *cupLeftCommute*: $A \cup (B \cup C) = B \cup (A \cup C)$
by *blast*

lemma *cupAssoc*: $(A \cup B) \cup C = A \cup (B \cup C)$
by *blast*

Union is an AC operator: can be added to simp where appropriate

lemmas *cupAC* = *cupAssoc cupCommute cupLeftCommute cupLeftAbsorb*

Lemmas useful for simplifying enumerated sets are active by default

lemmas *enumeratedSetSimps* [*simp*] =
addEltSubset-iff addEltEqualAddElt addEltCommute addEltTwice
capAddEltLeft capAddEltRight cupAddEltLeft cupAddEltRight diffAddEltLeft

lemma *cupEqualEmpty* [*simp*]: $(A \cup B = \{\}) = (A = \{\} \land B = \{\})$
by *blast*

lemma *capCupDistrib*: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
by *blast*

lemma *cupCapDistrib*: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
by *blast*

lemma *diffSubset*: $A \setminus B \subseteq A$
by *blast*

lemma *subsetDiff*:
assumes $C \subseteq A$ **and** $C \cap B = \{\}$
shows $C \subseteq A \setminus B$
using *assms* **by** *blast*

lemma *diffSelf* [*simp*]: $A \setminus A = \{\}$
by *blast*

lemma *diffDisjoint*:
assumes $A \cap B = \{\}$
shows $A \setminus B = A$
using *assms* **by** *blast*

lemma *emptyDiff* [*simp*]: $\{\} \setminus A = \{\}$
by *blast*

lemma *diffAddElt*: $A \setminus addElt(a,B) = (A \setminus B) \setminus \{a\}$
by *blast*

lemma *cupDiffSelf*: $A \cup (B \setminus A) = A \cup B$
by *blast*

lemma *diffCupLeft*: $(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C)$
by *blast*

lemma *diffCupRight*: $A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C)$
by *blast*

lemma *cupDiffLeft*: $(A \setminus B) \cup C = (A \cup C) \setminus (A \cap (B \setminus C))$
by *blast*

lemma *cupDiffRight*: $C \cup (A \setminus B) = (C \cup A) \setminus (A \cap (B \setminus C))$
by *blast*

lemma *diffCapLeft*: $(A \cap B) \setminus C = A \cap (B \setminus C)$
by *blast*

lemma *diffCapRight*: $A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$
by *blast*

lemma *capDiffLeft*: $(A \setminus B) \cap C = (A \cap C) \setminus B$
by *blast*

lemma *capDiffRight*: $C \cap (A \setminus B) = (C \cap A) \setminus B$
by *blast*

lemma *isEmptySimps* [*simp*]:
$(S \cup T = \{\}) = ((S = \{\}) \wedge (T = \{\}))$
$(\{\} = S \cup T) = ((S = \{\}) \wedge (T = \{\}))$
$(S \cap T = \{\}) = (\forall x \in S : x \notin T)$
$(\{\} = S \cap T) = (\forall x \in S : x \notin T)$
$(A \setminus B = \{\}) = (A \subseteq B)$
$(\{\} = A \setminus B) = (A \subseteq B)$
$(subsetOf(S,P) = \{\}) = (\forall x \in S : \neg P(x))$
$(\{\} = subsetOf(S,P)) = (\forall x \in S : \neg P(x))$
$(setOfAll(S,e) = \{\}) = (S = \{\})$
$(\{\} = setOfAll(S,e)) = (S = \{\})$
$(addElt(a,S) = \{\}) = FALSE$
$(\{\} = addElt(a,S)) = FALSE$
$(SUBSET\ S = \{\}) = FALSE$
$(\{\} = SUBSET\ S) = FALSE$

by (*blast*+)

2.17 Generalized union: inclusions and equalities

lemma *UNIONSimps* [*simp*]:
  UNION {} = {}
  UNION *addElt*(*a*,*A*) = (*a* ∪ *UNION* *A*)
  (*UNION* *S* = {}) = (∀ *A* ∈ *S* : *A* = {})
by *blast*+

lemma *UNIONIsSubset* [*simp*]: (*UNION* *A* ⊆ *C*) = (∀ *x*∈*A* : *x* ⊆ *C*)
by *blast*

lemma *UNION-UB*:
  assumes *B* ∈ *A*
  shows *B* ⊆ *UNION* *A*
using *assms* **by** *blast*

lemma *UNION-LUB*:
  assumes ⋀*B*. *B* ∈ *A* ⟹ *B* ⊆ *C*
  shows *UNION* *A* ⊆ *C*
using *assms* **by** *blast*

lemma *UNIONCupDistrib*: *UNION* (*A* ∪ *B*) = *UNION* *A* ∪ *UNION* *B*
by *blast*

lemma *UNIONCap*: *UNION* (*A* ∩ *B*) ⊆ *UNION* *A* ∩ *UNION* *B*
by *blast*

lemma *UNIONDisjoint*: ((*UNION* *A*) ∩ *C* = {}) = (∀ *B*∈*A* : *B* ∩ *C* = {})
by *blast*

lemma *capUNION*: (*UNION* *A*) ∩ *B* = *UNION* { *C* ∩ *B* : *C* ∈ *A* }
by *blast*

lemma *diffUNIONLeft*: (*UNION* *A*) \ *B* = *UNION* { *C* \ *B* : *C* ∈ *A* }
by *blast*

lemma *UNION-mono*:
  assumes *A* ⊆ *B*
  shows *UNION* *A* ⊆ *UNION* *B*
using *assms* **by** *blast*

2.18 Generalized intersection: inclusions and equalities

lemma *INTERSimps* [*simp*]:
  INTER {} = {}
  INTER {*A*} = *A*
  A ≠ {} ⟹ *INTER* *addElt*(*a*,*A*) = (*a* ∩ *INTER* *A*)
by (*blast*+)

lemma *subsetINTER* [*simp*]:
  assumes $A \neq \{\}$
  shows $(C \subseteq \mathit{INTER}\ A) = (\forall B{\in}A : C \subseteq B)$
using *assms* **by** *blast*

lemma *INTER-LB*:
  assumes $B \in A$
  shows $\mathit{INTER}\ A \subseteq B$
using *assms* **by** *blast*

lemma *INTER-GLB*:
  assumes $A \neq \{\}$ **and** $\bigwedge B.\ B \in A \Longrightarrow C \subseteq B$
  shows $C \subseteq \mathit{INTER}\ A$
using *assms* **by** *blast*

lemma *INTERCupDistrib*:
  assumes $A \neq \{\}$ **and** $B \neq \{\}$
  shows $\mathit{INTER}\ (A \cup B) = \mathit{INTER}\ A \cap \mathit{INTER}\ B$
using *assms* **by** *auto*

lemma *capINTER*: $(\mathit{INTER}\ A) \cap B \subseteq \mathit{INTER}\ \{C \cap B : C \in A\}$
by *blast*

lemma *cupINTER*:
  assumes $A \neq \{\}$
  shows $(\mathit{INTER}\ A) \cup B = \mathit{INTER}\ \{C \cup B : C \in A\}$
using *assms* **by** *auto*

lemma *diffINTERLeft*: $(\mathit{INTER}\ A) \setminus B = \mathit{INTER}\ \{C \setminus B : C \in A\}$
by *auto*

lemma *diffINTERRight*:
  assumes $A \neq \{\}$
  shows $B \setminus (\mathit{INTER}\ A) = \mathit{UNION}\ \{B \setminus C : C \in A\}$
using *assms* **by** *auto*

lemma *bAllSubset*:
  assumes $\forall x \in A : P(x)$ **and** $B \subseteq A$ **and** $b \in B$
  shows $P(b)$
using *assms* **by** *blast*

lemma *INTER-anti-mono*:
  assumes $A \neq \{\}$ **and** $A \subseteq B$
  shows $\mathit{INTER}\ B \subseteq \mathit{INTER}\ A$
using *assms* **by** (*auto simp*: *INTER-def*)

2.19 Powerset: inclusions and equalities

lemma *SUBSETEmpty* [*simp*]: *SUBSET* {} = { {} }
by *blast*

lemma *SUBSETAddElt*:
SUBSET addElt(*a*,*A*) = *SUBSET A* ∪ {*addElt*(*a*,*X*) : *X* ∈ *SUBSET A*}
by (*rule setEqualI*, *auto*)

lemma *cupSUBSET*: (*SUBSET A*) ∪ (*SUBSET B*) ⊆ *SUBSET* (*A* ∪ *B*)
by *blast*

lemma *UNION-SUBSET* [*simp*]: *UNION* (*SUBSET A*) = *A*
by *blast*

lemma *SUBSET-UNION*: *A* ⊆ *SUBSET* (*UNION A*)
by *blast*

lemma *UNION-in-SUBSET*: (*UNION A* ∈ *SUBSET B*) = (*A* ∈ *SUBSET* (*SUBSET B*))
by *blast*

lemma *SUBSETcap*: *SUBSET* (*A* ∩ *B*) = *SUBSET A* ∩ *SUBSET B*
by *blast*

end

3 Fixed points for set-theoretical constructions

theory *FixedPoints*
imports *SetTheory*
begin

As a test for the encoding of TLA$^+$ set theory, we develop the Knaster-Tarski theorems for least and greatest fixed points in the subset lattice. Again, the proofs essentially follow Paulson's developments for Isabelle/ZF.

3.1 Monotonic operators

definition *Monotonic* :: [*c*, *c* ⇒ *c*] ⇒ *c* — monotonic operator on a domain
where *Monotonic*(*D*,*f*) ≡ *f*(*D*) ⊆ *D* ∧ (∀ *S*,*T* ∈ *SUBSET D* : *S* ⊆ *T* ⇒ *f*(*S*) ⊆ *f*(*T*))

lemma *monotonicDomain*:
Monotonic(*D*,*f*) ⟹ *f*(*D*) ⊆ *D*
by (*unfold Monotonic-def*, *blast*)

lemma *monotonicSubset*:

$[\![Monotonic(D,f);\ S \subseteq T;\ T \subseteq D]\!] \Longrightarrow f(S) \subseteq f(T)$
by (*unfold Monotonic-def*, *blast*)

lemma *monotonicSubsetDomain*:
$[\![\ Monotonic(D,f);\ S \subseteq D]\!] \Longrightarrow f(S) \subseteq D$
by (*unfold Monotonic-def*, *blast*)

lemma *monotonicCup*:
assumes *mono*: $Monotonic(D,f)$ **and** *s*: $S \subseteq D$ **and** *t*: $T \subseteq D$
shows $f(S) \cup f(T) \subseteq f(S \cup T)$
proof (*rule cupLUB*)
from *s t* **show** $f(S) \subseteq f(S \cup T)$
by (*intro monotonicSubset*[*OF mono*], *blast+*)
next
from *s t* **show** $f(T) \subseteq f(S \cup T)$
by (*intro monotonicSubset*[*OF mono*], *blast+*)
qed

lemma *monotonicCap*:
assumes *mono*: $Monotonic(D,f)$ **and** *s*: $S \subseteq D$ **and** *t*: $T \subseteq D$
shows $f(S \cap T) \subseteq f(S) \cap f(T)$
proof (*rule capGLB*)
from *s t* **show** $f(S \cap T) \subseteq f(S)$
by (*intro monotonicSubset*[*OF mono*], *blast+*)
from *s t* **show** $f(S \cap T) \subseteq f(T)$
by (*intro monotonicSubset*[*OF mono*], *blast+*)
qed

3.2 Least fixed point

The least fixed point is defined as the greatest lower bound of the set of all pre-fixed points, and the Knaster-Tarski theorem is shown for monotonic operators.

definition *lfp* :: $[c,\ c \Rightarrow c] \Rightarrow c$ — least fixed point as GLB of pre-fp's
where $lfp(D,f) \equiv INTER\ \{S \in SUBSET\ D : f(S) \subseteq S\}$

lemma *lfpLB*: — The lfp is contained in each pre-fixed point.
$[\![\ f(S) \subseteq S;\ S \subseteq D\ ]\!] \Longrightarrow lfp(D,f) \subseteq S$
by (*auto simp*: *lfp-def*)

lemma *lfpGLB*: — ... and it is the GLB of all such sets
$[\![f(D) \subseteq D;\ \bigwedge S.\ [\![f(S) \subseteq S;\ S \subseteq D]\!] \Longrightarrow A \subseteq S]\!] \Longrightarrow A \subseteq lfp(D,f)$
by (*force simp*: *lfp-def*)

lemma *lfpSubsetDomain*: $lfp(D,f) \subseteq D$
by (*auto simp*: *lfp-def*)

lemma *lfpPreFP*: — *lfp* is a pre-fixed point ...
assumes *mono*: $Monotonic(D,f)$

shows $f(lfp(D,f)) \subseteq lfp(D,f)$
proof (*rule lfpGLB*)
from *mono* **show** $f(D) \subseteq D$ **by** (*rule monotonicDomain*)
next
let $?mu = lfp(D,f)$
fix S
assume *pfp*: $f(S) \subseteq S$ **and** *dom*: $S \subseteq D$
hence $?mu \subseteq S$ **by** (*rule lfpLB*)
from *mono this dom* **have** $f(?mu) \subseteq f(S)$ **by** (*rule monotonicSubset*)
with *pfp* **show** $f(?mu) \subseteq S$ **by** *blast*
qed

lemma *lfpPostFP*: — . . . and a post-fixed point
assumes *mono*: $Monotonic(D,f)$
shows $lfp(D,f) \subseteq f(lfp(D,f))$
proof −
let $?mu = lfp(D,f)$
from *mono lfpSubsetDomain* **have** *1*: $f(?mu) \subseteq D$ **by** (*rule monotonicSubset-Domain*)
from *mono* **have** $f(?mu) \subseteq ?mu$ **by** (*rule lfpPreFP*)
from *mono this lfpSubsetDomain* **have** $f(f(?mu)) \subseteq f(?mu)$ **by** (*rule monoton-icSubset*)
from *this 1* **show** *?thesis* **by** (*rule lfpLB*)
qed

lemma *lfpFixedPoint*:
assumes *mono*: $Monotonic(D,f)$
shows $f(lfp(D,f)) = lfp(D,f)$ (**is** $?lhs = ?rhs$)
proof (*rule setEqual*)
from *mono* **show** $?lhs \subseteq ?rhs$ **by** (*rule lfpPreFP*)
next
from *mono* **show** $?rhs \subseteq ?lhs$ **by** (*rule lfpPostFP*)
qed

lemma *lfpLeastFixedPoint*:
assumes $Monotonic(D,f)$ **and** $S \subseteq D$ **and** $f(S) = S$
shows $lfp(D,f) \subseteq S$
using *assms* **by** (*intro lfpLB*, *auto*)

lemma *lfpMono*: — monotonicity of the *lfp* operator
assumes *g*: $g(D) \subseteq D$ **and** *f*: $\bigwedge S.\ S \subseteq D \Longrightarrow f(S) \subseteq g(S)$
shows $lfp(D,f) \subseteq lfp(D,g)$
using *g*
proof (*rule lfpGLB*)
fix S
assume *1*: $g(S) \subseteq S$ **and** *2*: $S \subseteq D$
with *f* **have** $f(S) \subseteq S$ **by** *blast*
from *this 2* **show** $lfp(D,f) \subseteq S$ **by** (*rule lfpLB*)
qed

3.3 Greatest fixed point

Dually, the least fixed point is defined as the least upper bound of the set of all post-fixed points, and the Knaster-Tarski theorem is again established.

definition $gfp :: [c,\ c \Rightarrow c] \Rightarrow c$ — greatest fixed point as LUB of post-fp's
where $gfp(D,f) \equiv UNION\ \{S \in SUBSET\ D : S \subseteq f(S)\}$

lemma *gfpUB*: — The gfp contains each post-fixed point ...
 $[\![S \subseteq f(S);\ S \subseteq D]\!] \Longrightarrow S \subseteq gfp(D,f)$
by (*auto simp*: *gfp-def*)

lemma *gfpLUB*: — ... and it is the LUB of all such sets.
 $[\![f(D) \subseteq D;\ \bigwedge S.\ [\![S \subseteq f(S);\ S \subseteq D]\!] \Longrightarrow S \subseteq A]\!] \Longrightarrow gfp(D,f) \subseteq A$
by (*auto simp*: *gfp-def*)

lemma *gfpSubsetDomain*: $gfp(D,f) \subseteq D$
by (*auto simp*: *gfp-def*)

lemma *gfpPostFP*: — @textgfp is a post-fixed point ...
 assumes *mono*: $Monotonic(D,f)$
 shows $gfp(D,f) \subseteq f(gfp(D,f))$
proof (*rule gfpLUB*)
 from *mono* **show** $f(D) \subseteq D$ **by** (*rule monotonicDomain*)
next
 let $?nu = gfp(D,f)$
 fix S
 assume *pfp*: $S \subseteq f(S)$ **and** *dom*: $S \subseteq D$
 hence $S \subseteq ?nu$ **by** (*rule gfpUB*)
 from *mono this gfpSubsetDomain* **have** $f(S) \subseteq f(?nu)$ **by** (*rule monotonicSub-set*)
 with *pfp* **show** $S \subseteq f(?nu)$ **by** *blast*
qed

lemma *gfpPreFP*: — ... and a pre-fixed point
 assumes *mono*: $Monotonic(D,f)$
 shows $f(gfp(D,f)) \subseteq gfp(D,f)$
proof −
 let $?nu = gfp(D,f)$
 from *mono gfpSubsetDomain* **have** *1*: $f(?nu) \subseteq D$ **by** (*rule monotonicSubset-Domain*)
 from *mono* **have** $?nu \subseteq f(?nu)$ **by** (*rule gfpPostFP*)
 from *mono this 1* **have** $f(?nu) \subseteq f(f(?nu))$ **by** (*rule monotonicSubset*)
 from *this 1* **show** *?thesis* **by** (*rule gfpUB*)
qed

lemma *gfpFixedPoint*:
 assumes *mono*: $Monotonic(D,f)$
 shows $f(gfp(D,f)) = gfp(D,f)$ (**is** $?lhs = ?rhs$)
proof (*rule setEqual*)

from *mono* **show** *?lhs* ⊆ *?rhs* **by** (*rule gfpPreFP*)
next
from *mono* **show** *?rhs* ⊆ *?lhs* **by** (*rule gfpPostFP*)
qed

lemma *gfpGreatestFixedPoint*:
assumes *Monotonic*(*D*,*f*) **and** $S \subseteq D$ **and** $f(S) = S$
shows $S \subseteq$ *gfp*(*D*,*f*)
using *assms* **by** (*intro gfpUB*, *auto*)

lemma *gfpMono*: — monotonicity of the *gfp* operator
assumes *g*: $g(D) \subseteq D$ **and** *f*: $\bigwedge S.\ S \subseteq D \Longrightarrow f(S) \subseteq g(S)$
shows *gfp*(*D*,*f*) ⊆ *gfp*(*D*,*g*)
proof (*rule gfpLUB*)
from *f g* **show** $f(D) \subseteq D$ **by** *blast*
next
fix *S*
assume *1*: $S \subseteq f(S)$ **and** *2*: $S \subseteq D$
with *f* **have** $S \subseteq g(S)$ **by** *blast*
from *this 2* **show** $S \subseteq$ *gfp*(*D*,*g*) **by** (*rule gfpUB*)
qed

end

4 TLA$^+$ Functions

theory *Functions*
imports *SetTheory*
begin

4.1 Syntax and axioms for functions

Functions in TLA$^+$ are not defined (e.g., as sets of pairs), but axiomatized, and in fact, pairs and tuples will be defined as special functions. Incidentally, this approach helps us to identify functional values, and to automate the reasoning about them. This theory considers only unary functions; functions with multiple arguments are defined as functions over products.

We follow the development of functions given in Section 16.1 of "Specifying Systems". In particular, we define the predicate *IsAFcn* that is true precisely of functional values.

consts
isAFcn :: $c \Rightarrow c$ — characteristic predicate
Fcn :: $[c,\ c \Rightarrow c] \Rightarrow c$ — function constructor
DOMAIN :: $c \Rightarrow c$ ((*DOMAIN* -) [*100*]*90*) — domain of a function
fapply :: $[c,\ c] \Rightarrow c$ ((-[-]) [*89*,*0*]*90*) — function application

FuncSet :: [*c*,*c*] ⇒ *c* (([- −> -]) *900*) — function space

syntax
@*Fcn* :: [*idt*,*c*,*c*] ⇒ *c* ((*1*[- \in - |−> -]) *900*)
syntax (*xsymbols*)
FuncSet :: [*c*,*c*] ⇒ *c* (([- → -]) *900*)
@*Fcn* :: [*idt*,*c*,*c*] ⇒ *c* ((*1*[- ∈ - ↦ -]) *900*)
syntax (*HTML* **output**)
FuncSet :: [*c*,*c*] ⇒ *c* (([- → -]) *900*)
@*Fcn* :: [*idt*,*c*,*c*] ⇒ *c* ((*1*[- ∈ - ↦ -]) *900*)
translations
[*x* ∈ *S* ↦ *e*] ⇌ *CONST Fcn*(*S*, λ*x*. *e*)

axiomatization where
fcnIsAFcn [*intro*!,*simp*]: *isAFcn*(*Fcn*(*S*,*e*)) **and**
isAFcn-def: *isAFcn*(*f*) ≡ *f* = [*x* ∈ *DOMAIN f* ↦ *f*[*x*]] **and**
DOMAIN [*simp*]: *DOMAIN Fcn*(*S*,*e*) = *S* **and**
fapply [*simp*]: *v* ∈ *S* ⟹ *Fcn*(*S*,*e*)[*v*] = *e*(*v*) **and**
fcnEqual[*elim*!]: ⟦*isAFcn*(*f*); *isAFcn*(*g*); *DOMAIN f* = *DOMAIN g*; ∀ *x* ∈ *DOMAIN g* : *f*[*x*]=*g*[*x*]⟧
⟹ *f* = *g* **and**
FuncSet: *f* ∈ [*S* → *T*] ⇔ *isAFcn*(*f*) ∧ *DOMAIN f* = *S* ∧ (∀ *x*∈*S* : *f*[*x*] ∈ *T*)

lemmas — establish set equality for domains and function spaces
setEqualI [**where** *A* = *DOMAIN f*, *standard*, *intro*!]
setEqualI [**where** *B* = *DOMAIN f*, *standard*, *intro*!]
setEqualI [**where** *A* = [*S* → *T*], *standard*, *intro*!]
setEqualI [**where** *B* = [*S* → *T*], *standard*, *intro*!]

definition *except* :: [*c*,*c*,*c*] ⇒ *c* — function override
where *except*(*f*,*v*,*e*) ≡ [*x* ∈ *DOMAIN f* ↦ (*IF x*=*v THEN e ELSE f*[*x*])]

nonterminal
xcpt
syntax
-*xcpt* :: [*c*,*c*] ⇒ *xcpt* ((![-] =/ -))
-*xcpts* :: [*c*,*c*,*xcpt*] ⇒ *xcpt* ((![-] =/ -,/ -))
-*except* :: [*c*,*xcpt*] ⇒ *c* ([- *EXCEPT*/ -] [*900*,*0*] *900*)
translations
-*except*(*f*, -*xcpts*(*v*,*e*, *xcs*)) ⇌ -*except*(*CONST except*(*f*,*v*,*e*), *xcs*)
[*f EXCEPT* ![*v*] = *e*] ⇌ *CONST except*(*f*,*v*,*e*)

The following operators are useful for representing functions with finite domains by enumeration. They are not part of basic TLA$^+$, but they are defined in the TLC module of the standard library.

definition *oneArg* :: [*c*,*c*] ⇒ *c* (**infixl** :> *75*)
where *d* :> *e* ≡ [*x* ∈ {*d*} ↦ *e*]

definition *extend* :: $[c,c] \Rightarrow c$ (**infixl** @@ *70*)
where f @@ $g \equiv [x \in (DOMAIN\ f) \cup (DOMAIN\ g) \mapsto$
$IF\ x \in DOMAIN\ f\ THEN\ f[x]\ ELSE\ g[x]]$

4.2 *isAFcn*: identifying functional values

lemma *boolifyIsAFcn* [*simp*]: $boolify(isAFcn(f)) = isAFcn(f)$
by (*simp add*: *isAFcn-def*)

lemma *isBoolIsAFcn* [*intro*!,*simp*]: $isBool(isAFcn(f))$
by (*unfold isBool-def*, *rule boolifyIsAFcn*)

lemma [*intro*!,*simp*]: $isAFcn([f\ EXCEPT\ ![c] = e])$
by (*simp add*: *except-def*)

We derive instances of axiom *fcnEqual* that help in automating proofs about equality of functions.

lemma *fcnEqual2*[*elim*!]:
$[\![isAFcn(g); isAFcn(f); DOMAIN\ f = DOMAIN\ g; \forall x \in DOMAIN\ g : f[x]=g[x]]\!]$
$\Longrightarrow f = g$
by (*rule fcnEqual*)

— possibly useful as a simplification rule, but cannot be active by default
lemma *fcnEqualIff*:
assumes $isAFcn(f)$ **and** $isAFcn(g)$
shows $(f = g) = (DOMAIN\ f = DOMAIN\ g \wedge (\forall x \in DOMAIN\ g : f[x] = g[x]))$
using *assms* **by** *auto*

lemma [*intro*!]:
$[\![isAFcn(f); DOMAIN\ f = S; \forall x \in S : f[x] = e(x)]\!] \Longrightarrow [x \in S \mapsto e(x)] = f$
$[\![isAFcn(f); DOMAIN\ f = S; \forall x \in S : f[x] = e(x)]\!] \Longrightarrow f = [x \in S \mapsto e(x)]$
by *auto*

lemma [*intro*!]:
$[\![isAFcn(f); DOMAIN\ f = DOMAIN\ g; v \in DOMAIN\ g \Longrightarrow f[v] = w;$
$\forall y \in DOMAIN\ g : y \neq v \Rightarrow f[y] = g[y]]\!]$
$\Longrightarrow [g\ EXCEPT\ ![v] = w] = f$
$[\![isAFcn(f); DOMAIN\ f = DOMAIN\ g; v \in DOMAIN\ g \Longrightarrow f[v] = w;$
$\forall y \in DOMAIN\ g : y \neq v \Rightarrow f[y] = g[y]]\!]$
$\Longrightarrow f = [g\ EXCEPT\ ![v] = w]$
by (*auto simp*: *except-def*)

4.3 Theorems about functions

lemma *fcnCong* :
assumes $S = T$ **and** $\bigwedge x.\ x \in T \Longrightarrow e(x) = f(x)$

shows $[x \in S \mapsto e(x)] = [x \in T \mapsto f(x)]$
using *assms* **by** *auto*

lemma *domainExcept* [*simp*]: $DOMAIN\ [f\ EXCEPT\ ![v] = e] = DOMAIN\ f$
by (*simp add*: *except-def*)

lemma *applyExcept* [*simp*]:
assumes $w \in DOMAIN\ f$
shows $[f\ EXCEPT\ ![v] = e][w] = (IF\ w{=}v\ THEN\ e\ ELSE\ f[w])$
using *assms* **by** (*auto simp*: *except-def*)

lemma *exceptI*:
assumes $w \in DOMAIN\ f$ **and** $v{=}w \Longrightarrow P(e)$ **and** $v \neq w \Longrightarrow P(f[w])$
shows $P([f\ EXCEPT\ ![v]{=}e][w])$
using *assms* **by** (*auto simp*: *except-def intro*: *condI*)

lemma *exceptTrivial*:
assumes $v \notin DOMAIN\ f$ **and** *isAFcn*(f)
shows $[f\ EXCEPT\ ![v]{=}e] = f$
using *assms* **by** *auto*

lemma *exceptEqual* [*simp*]:
assumes *isAFcn*(f)
shows $([f\ EXCEPT\ ![v] = e] = f) = (v \notin DOMAIN\ f \lor f[v] = e)$
using *assms* **by** (*auto simp*: *fcnEqualIff*)

A function can be defined from a predicate. Using the *CHOOSE* operator, the definition does not require the predicate to be functional.

lemma *fcnConstruct*:
assumes *hyp*: $\forall x \in S : \exists y : P(x,y)$
shows $\exists f : isAFcn(f) \land DOMAIN\ f = S \land (\forall x{\in}S : P(x, f[x]))$
(**is** $\exists f : ?F(f)$)
proof −
let $?fn = [x \in S \mapsto CHOOSE\ y : P(x,y)]$
have $?F(?fn)$
proof *auto*
fix x
assume $x \in S$
with *hyp* **have** $\exists y : P(x,y)$ **..**
thus $P(x, CHOOSE\ y : P(x,y))$ **by** (*rule chooseI-ex*)
qed
thus *?thesis* **..**
qed

4.4 Function spaces

lemma *inFuncSetIff*:
$(f \in [S \to T]) = (isAFcn(f) \land DOMAIN\ f = S \land (\forall x{\in}S : f[x] \in T))$
proof (*rule boolEqual*)

show $f \in [S \to T] \Leftrightarrow isAFcn(f) \land DOMAIN\, f = S \land (\forall\, x{\in}S : f[x] \in T)$
by (*rule FuncSet*)
qed (*auto*)

lemma *funcSetIsAFcn* [*simp*]:
assumes $f \in [S \to T]$
shows $isAFcn(f)$
using *assms* **unfolding** *inFuncSetIff* **by** *blast*

lemma *funcSetDomain* [*simp*]:
assumes $f \in [S \to T]$
shows $DOMAIN\, f = S$
using *assms* **unfolding** *inFuncSetIff* **by** *blast*

lemma *funcSetValue* [*simp*]:
assumes $f \in [S \to T]$ **and** $x \in S$
shows $f[x] \in T$
using *assms* **unfolding** *inFuncSetIff* **by** *blast*

lemma *funcSetE* :
assumes $f \in [S \to T]$ **and** $x \in S$ **and** $[\![isAFcn(f);\ DOMAIN\, f = S;\ f[x] \in T]\!]$
$\Longrightarrow P$
shows P
using *assms* **unfolding** *inFuncSetIff* **by** *blast*

lemma *funcSetE'* [*elim*]:
assumes $f \in [S \to T]$ **and** $[\![isAFcn(f);\ DOMAIN\, f = S;\ \forall\, x \in S : f[x] \in T]\!]$
$\Longrightarrow P$
shows P
using *assms* **unfolding** *inFuncSetIff* **by** *blast*

lemma *funcSetFcnEqual* [*elim*!]:
assumes $f \in [S \to T]$ **and** $isAFcn(g)$ **and** $DOMAIN\ g = S$
and $\forall\, x \in S$: $f[x] = g[x]$
shows $f = g$
using *assms* **by** *auto*

declare *funcSetFcnEqual*[*symmetric*, *elim*]

lemma *inFuncSet* [*intro*!]:
assumes $isAFcn(f)$ **and** $DOMAIN\, f = S$ **and** $\forall\, x \in S : f[x] \in T$
shows $f \in [S \to T]$
using *assms* **unfolding** *inFuncSetIff* **by** *blast*

lemma *funcSetSubRange*:
assumes $f \in [S \to T]$ **and** $T \subseteq U$
shows $f \in [S \to U]$
using *assms* **by** *auto*

lemma *funcSetEmpty* [*simp*]:
$([S \to T] = \{\}) = ((S \neq \{\}) \wedge (T = \{\}))$
$(\{\} = [S \to T]) = ((S \neq \{\}) \wedge (T = \{\}))$
by *auto*

lemma *isAFcnFuncSet*:
assumes *hyp*: $isAFcn(f)$
shows $\exists\, S,\ T : f \in [S \to T]$
using *assms* **by** *blast*

lemma *functionInFuncSet*:
assumes $\forall\, x \in S : e(x) \in T$
shows $[x{\in}S \mapsto e(x)] \in [S \to T]$
using *assms* **by** *auto*

lemma *exceptInFuncSet*[*elim*!]:
assumes *1*: $f \in [S \to U]$ **and** *2*: $U \subseteq T$
and *3*: $[\![v \in S;\ isAFcn(f);\ DOMAIN\, f = S;\ \forall\, x \in S : f[x] \in U]\!] \Longrightarrow e \in T$
shows $[f\ EXCEPT\ ![v]{=}e] \in [S \to T]$ (**is** $?exc \in [S \to T]$)
proof
from *1* **show** $DOMAIN\ ?exc = S$ **by** *auto*
next
from *assms* **show** $\forall\, x \in S : ?exc[x] \in T$ **by** *auto*
qed (*simp*)

The following special case is useful for invariant proofs where one proves type correctness. The additional hypotheses make the type of f available and are useful, for example, when the expression e is of the form $f[u]$ for some $u \in S$.

lemma *exceptInFuncSetSame*:
assumes $f \in [S \to T]$
and $[\![v \in S;\ isAFcn(f);\ DOMAIN\, f = S;\ \forall\, x \in S : f[x] \in T]\!] \Longrightarrow e \in T$
shows $[f\ EXCEPT\ ![v]{=}e] \in [S \to T]$
using *assms* **by** *auto*

4.5 Finite functions and extension

lemma *oneArgIsAFcn* [*simp*, *intro*!]: $isAFcn(d :> e)$
by (*simp add*: *oneArg-def*)

lemma *oneArgDomain* [*simp*]: $DOMAIN\ (d :> e) = \{d\}$
by (*simp add*: *oneArg-def*)

lemma *oneArgVal* [*simp*]: $(d{:>}e)[d] = e$
by (*simp add*: *oneArg-def*)

lemma *oneArgFuncSet*: $(d :> e) \in [\{d\} \to \{e\}]$
by *auto*

lemma *extendIsAFcn* [*simp*, *intro*!]: *isAFcn* (f @@ g)
by (*simp add*: *extend-def*)

lemma *extendDomain* [*simp*]: *DOMAIN* (f @@ g) = (*DOMAIN* f) $\cup$ (*DOMAIN* g)
by (*simp add*: *extend-def*)

lemma *extendVal* [*simp*]:
  assumes $x \in$ (*DOMAIN* f) $\cup$ (*DOMAIN* g)
  shows (f @@ g)[x] = (*IF* $x \in$ *DOMAIN* f *THEN* $f[x]$ *ELSE* $g[x]$)
using *assms* **by** (*simp add*: *extend-def*)

lemma *extendFuncSet*:
  assumes $f \in [S \to U]$ **and** $g \in [T \to V]$
  shows f @@ $g \in [S \cup T \to U \cup V]$
using *assms* **by** *auto*

lemma *oneArgEqual* [*intro*!]:
  ⟦*isAFcn*(f); *DOMAIN* $f = \{d\}$; $f[d] = e$⟧ $\Longrightarrow$ (d :> e) = f
  ⟦*isAFcn*(f); *DOMAIN* $f = \{d\}$; $f[d] = e$⟧ $\Longrightarrow$ f = (d :> e)
by *force*+

lemma *oneArgEqualIff* [*simp*]:
  isAFcn(f) $\Longrightarrow$ (f = (d :> e)) = ((*DOMAIN* $f = \{d\}$) $\wedge$ $f[d] = e$)
  isAFcn(f) $\Longrightarrow$ ((d :> e) = f) = ((*DOMAIN* $f = \{d\}$) $\wedge$ $f[d] = e$)
by *auto*

— infer equalities $f = g$ @@ h
lemmas
  fcnEqual[**where** $f = f$ @@ h, *standard*, *intro*!]
  fcnEqual[**where** $g = g$ @@ h, *standard*, *intro*!]

lemma *extendEqualIff* [*simp*]:
  isAFcn(f) $\Longrightarrow$ ($f = g$ @@ h) =
        (*DOMAIN* f = (*DOMAIN* g) $\cup$ (*DOMAIN* h) $\wedge$
        ($\forall x \in$ *DOMAIN* g : $f[x] = g[x]$) $\wedge$
        ($\forall x \in$ *DOMAIN* h $\setminus$ *DOMAIN* g : $f[x] = h[x]$))
  isAFcn(f) $\Longrightarrow$ (g @@ $h = f$) =
        (*DOMAIN* f = (*DOMAIN* g) $\cup$ (*DOMAIN* h) $\wedge$
        ($\forall x \in$ *DOMAIN* g : $g[x] = f[x]$) $\wedge$
        ($\forall x \in$ *DOMAIN* h $\setminus$ *DOMAIN* g : $h[x] = f[x]$))
by *auto*

4.6 Notions about functions

4.6.1 Image and Range

Image of a set under a function, and range of a function. Because the application of a function to an argument outside of its domain usually leads to silliness, we restrict to the domain when defining the image.

definition *Image*
where $Image(f,A) \equiv \{ f[x] : x \in A \cap DOMAIN\, f \}$

The range of a function, introduced as an abbreviation (macro). To reason about the range, apply the theorems about *Image*, or simply rewrite with *Image-def*.

abbreviation *Range*
where $Range(f) \equiv Image(f,\ DOMAIN\, f)$

lemma *imageI* [*intro*]:
  assumes $x \in A$ **and** $x \in DOMAIN\, f$
  shows $f[x] \in Image(f,A)$
using *assms* **by** (*auto simp*: *Image-def*)

lemma *imageI-eq*:
  assumes $x \in A$ **and** $x \in DOMAIN\, f$ **and** $y = f[x]$
  shows $y \in Image(f,A)$
using *assms* **by** (*auto simp*: *Image-def*)

lemma *imageI-exEq* [*intro*]:
  assumes $\exists\, x \in A \cap DOMAIN\, f : y = f[x]$
  shows $y \in Image(f,A)$
using *assms* **by** (*auto intro*: *imageI-eq*)

lemma *rangeI*: — useful special case
  assumes $\exists\, x \in DOMAIN\, f\colon y = f[x]$
  shows $y \in Range(f)$
using *assms* **by** *auto*

lemma *imageE* [*elim*]:
  assumes $y \in Image(f,A)$ **and** $\bigwedge x.\ [\![x \in A;\ x \in DOMAIN\, f;\ y = f[x]]\!] \Longrightarrow P$
  shows P
using *assms* **by** (*auto simp*: *Image-def*)

lemma *imageEqualI* [*intro*!]:
  assumes $\bigwedge y.\ y \in B \Leftrightarrow (\exists\, x \in A \cap DOMAIN\, f : y = f[x])$
  shows $Image(f,A) = B$
using *assms* **by** (*intro setEqualI*, *auto simp*: *Image-def*)

declare *imageEqualI* [*symmetric*,*intro*!]

lemma *inImageiff* [*simp*]:
  $(y \in Image(f,A)) = (\exists\, x \in A \cap DOMAIN\, f : y = f[x])$
by *blast*

lemma *imageEmpty* [*simp*]:
  $(Image(f,A) = \{\}) = (A \cap DOMAIN\, f = \{\})$
  $(\{\} = Image(f,A)) = (A \cap DOMAIN\, f = \{\})$

by *auto*

4.6.2 Injective functions

definition *InjectiveOn*
where $InjectiveOn(f,A) \equiv \forall\, x,y \in A \cap DOMAIN\, f : f[x] = f[y] \Rightarrow x = y$

abbreviation *Injective* — special case: injective function
where $Injective(f) \equiv InjectiveOn(f,\ DOMAIN\, f)$

definition *Injections*
where $Injections(S,T) \equiv \{\ f \in [S \to T] : Injective(f)\ \}$

lemmas
  setEqualI [**where** $A = Injections(S,T)$, *standard*, *intro*!]
  setEqualI [**where** $B = Injections(S,T)$, *standard*, *intro*!]

lemma *injectiveOnIsBool* [*intro*!,*simp*]:
  $isBool(InjectiveOn(f,A))$
by (*simp add*: *InjectiveOn-def*)

lemma *boolifyInjectiveOn* [*simp*]:
  $boolify(InjectiveOn(f,A)) = InjectiveOn(f,A)$
by *auto*

For the moment, no support by default for automatic reasoning.

lemma *injectiveOnI*:
  assumes $\bigwedge x\ y.\ [\![\ x \in A;\ x \in DOMAIN\, f;\ y \in A;\ y \in DOMAIN\, f;\ f[x] = f[y]$
$]\!] \Longrightarrow x = y$
  shows $InjectiveOn(f,A)$
using *assms* **by** (*auto simp*: *InjectiveOn-def*)

lemma *injectiveOnD*:
  assumes $f[x] = f[y]$ **and** $InjectiveOn(f,A)$
  and $x \in A$ **and** $x \in DOMAIN\, f$ **and** $y \in A$ **and** $y \in DOMAIN\, f$
  shows $x = y$
using *assms* **by** (*auto simp*: *InjectiveOn-def*)

lemma *injectiveOnE*:
  assumes $InjectiveOn(f,A)$
  and $(\bigwedge x\ y.\ [\![\ x \in A;\ x \in DOMAIN\, f;\ y \in A;\ y \in DOMAIN\, f;\ f[x] = f[y]\ ]\!] \Longrightarrow$
$x{=}y) \Longrightarrow P$
  shows P
using *assms* **by** (*auto simp*: *InjectiveOn-def*)

lemma *injectiveOnIff*: — useful for simplification
  assumes $InjectiveOn(f,A)$ **and** $x \in A \cap DOMAIN\, f$ **and** $y \in A \cap DOMAIN\, f$
  shows $(f[x] = f[y]) = (x = y)$
using *assms injectiveOnD* **by** *auto*

lemma *injectiveOnSubset*:
  assumes *InjectiveOn*(f,A) **and** $B \subseteq A$
  shows *InjectiveOn*(f,B)
using *assms* **by** (*auto simp*: *InjectiveOn-def*)

lemma *injectiveOnDifference*:
  assumes *InjectiveOn*(f,A)
  shows *InjectiveOn*(f, $A \setminus B$)
using *assms* **by** (*auto simp*: *InjectiveOn-def*)

The existence of an inverse function implies injectivity.

lemma *inverseThenInjective*:
  assumes *inv*: $\bigwedge x.\ [\![\ x \in A;\ x \in DOMAIN\ f\ ]\!] \Longrightarrow g[f[x]] = x$
  shows *InjectiveOn*(f,A)
proof (*rule injectiveOnI*)
  fix x y
  assume x: $x \in A$ $x \in DOMAIN\ f$
    and y: $y \in A$ $y \in DOMAIN\ f$
    and *eq*: $f[x] = f[y]$
  from x **have** *x1*: $x = g[f[x]]$ **by** (*rule sym*[*OF inv*])
  from y **have** *y1*: $g[f[y]] = y$ **by** (*rule inv*)
  from *x1 y1 eq* **show** $x = y$ **by** *simp*
qed

Trivial cases.

lemma *injectiveOnEmpty* [*intro*!,*simp*]:
  InjectiveOn(f, {})
by (*blast intro*: *injectiveOnI*)

lemma *injectiveOnSingleton* [*intro*!,*simp*]:
  InjectiveOn(f, $\{x\}$)
by (*blast intro*: *injectiveOnI*)

Injectivity for function extensions.

lemma *injectiveOnExcept*:
  assumes *1*: *InjectiveOn*(f, $A \setminus \{v\}$) **and** *2*: *isAFcn*(f)
  and *3*: $v \in DOMAIN\ f \Rightarrow (\forall\, x \in (DOMAIN\ f \cap A) \setminus \{v\} : f[x] \neq e)$
  shows *InjectiveOn*([f *EXCEPT* ![v] = e], A) (**is** *InjectiveOn*(*?exc*, A))
proof (*rule injectiveOnI*)
  fix x y
  assume *x1*: $x \in A$ **and** *x2*: $x \in DOMAIN$ *?exc*
    and *y1*: $y \in A$ **and** *y2*: $y \in DOMAIN$ *?exc*
    and *eq*: *?exc*[x] = *?exc*[y]
  show $x = y$
  proof (*cases* $v \in DOMAIN\ f$)
    case *False*
    from *False 2* **have** *?exc* = f **by** (*rule exceptTrivial*)
    with *eq* **have** *eq′*: $f[x] = f[y]$ **by** *simp*

```
      from False x1 x2 have x3: x ∈ (A \ {v}) x ∈ DOMAIN f by auto
      from False y1 y2 have y3: y ∈ (A \ {v}) y ∈ DOMAIN f by auto
      from eq' 1 x3 y3 show x = y by (rule injectiveOnD)
    next
      case True
      with 3 have 4: ∀ x ∈ (DOMAIN f ∩ A) \ {v} : f[x] ≠ e by (rule mp)
      show x = y
      proof (rule classical)
        assume neq: x ≠ y
        have x ≠ v
        proof
  assume contr: x = v
  with True have fx: ?exc[x] = e by auto
  from y2 contr neq have ?exc[y] = f[y] by auto
  with contr neq y1 y2 4 have ?exc[y] ≠ e by auto
  with fx eq show FALSE by simp
        qed
        with x1 x2 have x3: x ∈ (A \ {v}) x ∈ DOMAIN f by auto
        have y ≠ v  — symmetrical reasoning
        proof
  assume contr: y = v
  with True have fy: ?exc[y] = e by auto
  from x2 contr neq have ?exc[x] = f[x] by auto
  with contr neq x1 x2 4 have ?exc[x] ≠ e by auto
  with fy eq show FALSE by simp
        qed
        with y1 y2 have y3: y ∈ (A \ {v}) y ∈ DOMAIN f by auto
        from eq x3 y3 have f[x] = f[y] by auto
        from this 1 x3 y3 show x = y by (rule injectiveOnD)
      qed
    qed
qed

lemma injectiveOnExtend:
  assumes f: InjectiveOn(f,A) and g: InjectiveOn(g,A \ DOMAIN f)
  and disj: Image(f, A) ∩ Image(g, A \ DOMAIN f) = {}
  shows InjectiveOn(f @@ g, A)
proof (rule injectiveOnI)
  fix x y
  assume 1: x ∈ A y ∈ A x ∈ DOMAIN (f @@ g) y ∈ DOMAIN (f @@ g)
     and 2: (f @@ g)[x] = (f @@ g)[y]
  show x = y
  proof (cases x ∈ DOMAIN f)
    case True
    have y ∈ DOMAIN f
    proof (rule contradiction)
      assume y: y ∉ DOMAIN f
      from 1 True have (f @@ g)[x] ∈ Image(f,A) by auto
      moreover
```

from *1 y* **have** $(f\ @@\ g)[y] \in Image(g,\ A \setminus DOMAIN\, f)$ **by** *auto*
moreover
note *2*
ultimately have $(f\ @@\ g)[y] \in Image(f,A) \cap Image(g,\ A \setminus DOMAIN\, f)$
by *auto*
with *disj* **show** *FALSE* **by** *blast*
qed
with *True f 1 2* **show** $x = y$ **by** (*auto elim*: *injectiveOnD*)
next
case *False*
have $y \notin DOMAIN\, f$
proof
assume *y*: $y \in DOMAIN\, f$
with *1* **have** $(f\ @@\ g)[y] \in Image(f,A)$ **by** *auto*
moreover
from *False 1* **have** $(f\ @@\ g)[x] \in Image(g,\ A \setminus DOMAIN\, f)$ **by** *auto*
moreover
note *2*
ultimately have $(f\ @@\ g)[y] \in Image(f,A) \cap Image(g,\ A \setminus DOMAIN\, f)$
by *auto*
with *disj* **show** *FALSE* **by** *blast*
qed
with *False g 1 2* **show** $x = y$ **by** (*auto elim*: *injectiveOnD*)
qed
qed

lemma *injectiveExtend*: — special case
assumes *1*: $Injective(f)$ **and** *2*: $InjectiveOn(g,\ DOMAIN\ g \setminus DOMAIN\, f)$
and *3*: $Range(f) \cap Image(g,\ DOMAIN\ g \setminus DOMAIN\, f) = \{\}$
shows $Injective(f\ @@\ g)$
proof (*rule injectiveOnExtend*)
from *1* **show** $InjectiveOn(f,\ DOMAIN\ (f\ @@\ g))$
by (*auto simp*: *InjectiveOn-def*)
next
from *2* **show** $InjectiveOn(g,\ DOMAIN\ (f\ @@\ g) \setminus DOMAIN\, f)$
by (*auto simp*: *InjectiveOn-def*)
next
show $Image(f,\ DOMAIN\ (f\ @@\ g)) \cap Image(g,\ DOMAIN\ (f\ @@\ g) \setminus DOMAIN$
$f) = \{\}$
proof (*clarify*)
fix *x*
assume *xf*: $x \in Image(f,\ DOMAIN\ (f\ @@\ g))$
and *xg*: $x \in Image(g,\ DOMAIN\ (f\ @@\ g) \setminus DOMAIN\, f)$
from *xf* **have** $x \in Range(f)$ **by** *auto*
moreover
from *xg* **have** $x \in Image(g,\ DOMAIN\ g \setminus DOMAIN\, f)$ **by** *auto*
moreover
note *3*
ultimately show *FALSE* **by** *blast*

qed
qed

lemma *injectiveOnImageInter*:
assumes *InjectiveOn*(f, A) **and** $B \subseteq A$ **and** $C \subseteq A$
shows *Image*$(f, B \cap C) =$ *Image*$(f,B) \cap$ *Image*(f,C)
using *assms* **by** (*auto simp*: *InjectiveOn-def Image-def*)

lemma *injectiveOnImageDifference*:
assumes *InjectiveOn*(f,A) **and** $B \subseteq A$ **and** $C \subseteq A$
shows *Image*$(f, B \setminus C) =$ *Image*$(f,B) \setminus$ *Image*(f,C)
using *assms* **by** (*auto simp*: *InjectiveOn-def Image-def*)

lemma *injectiveImageMember*:
assumes *Injective*(f) **and** $a \in$ *DOMAIN* f
shows $(f[a] \in$ *Image*$(f,A)) = (a \in A)$
using *assms* **by** (*auto simp*: *InjectiveOn-def Image-def*)

lemma *injectiveImageSubset*:
assumes f: *Injective*(f)
shows (*Image*$(f,A) \subseteq$ *Image*$(f,B)) = (A \cap$ *DOMAIN* $f \subseteq B \cap$ *DOMAIN* $f)$
proof (*auto*) — the inclusion "$\supseteq$" is solved automatically
fix x
assume *ab*: *Image*$(f,A) \subseteq$ *Image*(f,B) **and** x: $x \in A$ $x \in$ *DOMAIN* f
from x **have** $f[x] \in$ *Image*(f,A) **by** (*rule imageI*)
with *ab* **have** $f[x] \in$ *Image*(f,B) **..**
then obtain z **where** z: $z \in B \cap$ *DOMAIN* f $f[x] = f[z]$ **by** *auto*
from f z x **have** $x = z$ **by** (*auto elim*: *injectiveOnD*)
with z **show** $x \in B$ **by** *simp*
qed

lemma *injectiveImageEqual*:
assumes f: *Injective*(f)
shows (*Image*$(f,A) =$ *Image*$(f,B)) = (A \cap$ *DOMAIN* $f = B \cap$ *DOMAIN* $f)$
proof −
have (*Image*$(f,A) =$ *Image*$(f,B)) =$ (*Image*$(f,A) \subseteq$ *Image*$(f,B) \wedge$ *Image*(f,B) $\subseteq$ *Image*$(f,A))$
by *auto*
also from f **have** $\ldots = (A \cap$ *DOMAIN* $f \subseteq B \cap$ *DOMAIN* $f \wedge B \cap$ *DOMAIN* $f \subseteq A \cap$ *DOMAIN* $f)$
by (*simp add*: *injectiveImageSubset*)
also have $\ldots = (A \cap$ *DOMAIN* $f = B \cap$ *DOMAIN* $f)$
by *auto*
finally show *?thesis* **.**
qed

lemma *injectionsI*:
assumes $f \in [S \to T]$ **and** $\bigwedge x\ y.\ [\![\ x \in S;\ y \in S;\ f[x] = f[y]\ ]\!] \Longrightarrow x = y$
shows $f \in$ *Injections*(S,T)

using *assms funcSetDomain* **by** (*auto simp*: *Injections-def InjectiveOn-def*)

lemma *injectionsE*:
  assumes *1*: $f \in Injections(S,T)$
  and *2*: $[\![f \in [S \to T]; \bigwedge x\ y.\ [\![x \in S;\ y \in S;\ f[x] = f[y]]\!] \Longrightarrow x = y]\!] \Longrightarrow P$
  shows P
using *assms* **unfolding** *Injections-def InjectiveOn-def* **by** *blast*

4.6.3 Surjective functions

definition *Surjective*
where $Surjective(f,A) \equiv A \subseteq Range(f)$

definition *Surjections*
where $Surjections(S,T) \equiv \{ f \in [S \to T] : Surjective(f,T) \}$

lemmas
  setEqualI [**where** $A = Surjections(S,T)$, *standard*, *intro*!]
  setEqualI [**where** $B = Surjections(S,T)$, *standard*, *intro*!]

lemma *surjectiveIsBool* [*intro*!,*simp*]:
  $isBool(Surjective(f,A))$
by (*simp add*: *Surjective-def*)

lemma *boolifySurjective* [*simp*]:
  $boolify(Surjective(f,A)) = Surjective(f,A)$
by *auto*

lemma *surjectiveI*:
  assumes $\bigwedge y.\ y \in A \Longrightarrow \exists x \in DOMAIN\, f : y = f[x]$
  shows $Surjective(f,A)$
unfolding *Surjective-def* **by** (*blast intro*: *rangeI*[*OF assms*])

lemma *surjectiveD*:
  assumes $Surjective(f,A)$ **and** $y \in A$
  shows $\exists x \in DOMAIN\, f : y = f[x]$
using *assms* **by** (*auto simp*: *Surjective-def Image-def*)

$[\![Surjective(f,A);\ y \in A;\ \bigwedge x.\ [\![x \in DOMAIN\, f;\ y = f[x]]\!] \Longrightarrow P]\!] \Longrightarrow P$

lemmas *surjectiveE* = *surjectiveD*[*THEN bExE*, *standard*]

lemma *surjectiveRange*:
  shows $Surjective(f, Range(f))$
by (*simp add*: *Surjective-def*)

lemma *surjectiveSubset*:
  assumes $Surjective(f,A)$ **and** $B \subseteq A$
  shows $Surjective(f,B)$
using *assms* **by** (*auto simp*: *Surjective-def*)

lemma *surjectionsI*:
  assumes $f \in [S \to T]$ **and** $\bigwedge y.\ y \in T \Longrightarrow \exists\, x \in S : y = f[x]$
  shows $f \in Surjections(S,T)$
using *assms* **by** (*unfold Surjections-def*, *auto intro*!: *surjectiveI*)

lemma *surjectionsFuncSet*:
  assumes $f \in Surjections(S,T)$
  shows $f \in [S \to T]$
using *assms* **by** (*simp add*: *Surjections-def*)

lemma *surjectionsSurjective*:
  assumes *1*: $f \in Surjections(S,T)$ **and** *2*: $y \in T$
  shows $\exists\, x \in S : y = f[x]$
proof −
  from *1* **have** $Surjective(f,T)$ **by** (*simp add*: *Surjections-def*)
  from *this 2* **have** $\exists\, x \in DOMAIN\, f : y = f[x]$ **by** (*rule surjectiveD*)
  with *1 funcSetDomain* **show** *?thesis* **by** (*auto simp*: *Surjections-def*)
qed

lemma *surjectionsE*:
  assumes *1*: $f \in Surjections(S,T)$
  and *2*: $[\![\ f \in [S \to T];\ \forall\, y{\in}T : \exists\, x \in S : y = f[x]\ ]\!] \Longrightarrow P$
  shows P
using *1 surjectionsFuncSet surjectionsSurjective* **by** (*intro 2*, *auto*)

lemma *surjectionsRange*:
  assumes $f \in Surjections(S,T)$
  shows $Range(f) = T$
using *assms* **by** (*rule surjectionsE*, *auto*)

4.6.4 Bijective functions

Here we do not define a predicate *Bijective* because it would require a set parameter for the codomain and would therefore be curiously asymmetrical.

definition *Bijections*
where $Bijections(S,T) \equiv Injections(S,T) \cap Surjections(S,T)$

lemmas
  setEqualI [**where** $A = Bijections(S,T)$, *standard*, *intro*!]
  setEqualI [**where** $B = Bijections(S,T)$, *standard*, *intro*!]

lemma *bijectionsI* [*intro*!]:
  assumes $f \in [S \to T]$ **and** $Injective(f)$ **and** $Surjective(f,T)$
  shows $f \in Bijections(S,T)$
using *assms* **by** (*simp add*: *Bijections-def Injections-def Surjections-def*)

lemma *bijectionsInjections*:
  assumes $f \in Bijections(S,T)$

shows $f \in Injections(S,T)$
using *assms* **by** (*simp add*: *Bijections-def*)

lemma *bijectionsSurjections*:
assumes $f \in Bijections(S,T)$
shows $f \in Surjections(S,T)$
using *assms* **by** (*simp add*: *Bijections-def*)

lemma *bijectionsE*:
assumes *1*: $f \in Bijections(S,T)$
and *2*: $[\![f \in [S \to T]; Injective(f); Surjective(f,T)]\!] \Longrightarrow P$
shows P
using *1* **by** (*intro 2*, *auto simp*: *Bijections-def Injections-def Surjections-def*)

4.6.5 Inverse of a function

definition *Inverse*
where $Inverse(f) \equiv [\ y \in Range(f) \mapsto CHOOSE\ x \in DOMAIN\ f : f[x] = y\ ]$

lemma *inverseIsAFcn* [*simp*,*intro*!]:
$isAFcn(Inverse(f))$
by (*simp add*: *Inverse-def*)

lemma *inverseDomain* [*simp*]:
$DOMAIN\ Inverse(f) = Range(f)$
by (*simp add*: *Inverse-def*)

lemma *inverseFcnSet*:
$Inverse(f) \in [Range(f) \to DOMAIN\ f]$
proof
show $\forall x \in Range(f) : Inverse(f)[x] \in DOMAIN\ f$
proof
fix y
assume *y*: $y \in Range(f)$
then obtain x **where** $x \in DOMAIN\ f$ **and** $f[x] = y$ **by** *auto*
hence $(CHOOSE\ x \in DOMAIN\ f : f[x] = y) \in DOMAIN\ f$ **by** (*rule bChooseInSet*)
with y **show** $Inverse(f)[y] \in DOMAIN\ f$ **by** (*simp add*: *Inverse-def*)
qed
qed (*auto*)

lemma *inverseInDomain*:
assumes $y \in Range(f)$
shows $Inverse(f)[y] \in DOMAIN\ f$
using *inverseFcnSet assms* **by** (*rule funcSetValue*)

lemma *Inverse*:

assumes *y*: $y \in Range(f)$
shows $f[\ Inverse(f)[y]\ ] = y$
proof −
from *y* **obtain** *x* **where** $x \in DOMAIN\, f$ **and** $f[x] = y$ **by** *auto*
hence $f[CHOOSE\ x \in DOMAIN\, f\colon f[x]{=}y] = y$ **by** (*rule bChooseI*)
with *y* **show** *?thesis* **by** (*simp add*: *Inverse-def*)
qed

lemma *inverseInjective* [*simp*,*intro*]:
Injective(*Inverse*(*f*))
proof (*intro injectiveOnI*, *auto*)
fix *x x′*
assume *x*: $x \in DOMAIN\, f$ **and** *x′*: $x' \in DOMAIN\, f$
and *eq*: $Inverse(f)[\ f[x]\ ] = Inverse(f)[\ f[x']\ ]$ (**is** *?invx* = *?invx′*)
from *x* **have** $f[x] = f[?invx]$ **by** (*intro sym*[*OF Inverse*], *auto*)
moreover
from *x′* **have** $f[?invx'] = f[x']$ **by** (*intro Inverse*, *auto*)
moreover
note *eq*
ultimately show $f[x] = f[x']$ **by** *simp*
qed

For injective functions, *Inverse* really inverts the function.

lemma *injectiveInverse*:
assumes *f*: *Injective*(*f*) **and** *x*: $x \in DOMAIN\, f$
shows $Inverse(f)[\ f[x]\ ] = x$
proof −
from *x* **have** $f[x] \in Range(f)$ **by** *auto*
hence $Inverse(f)[f[x]] = (CHOOSE\ z \in DOMAIN\ f\ :\ f[z] = f[x])$ **by** (*simp add*: *Inverse-def*)
moreover
have $\ldots = x$
proof (*rule bChooseI2*)
from *x* **show** $\exists\, z \in DOMAIN\, f\ :\ f[z] = f[x]$ **by** *blast*
next
fix *z*
assume $z \in DOMAIN\, f$ **and** $f[z] = f[x]$
with *f x* **show** $z = x$ **by** (*auto elim*: *injectiveOnD*)
qed
ultimately
show *?thesis* **by** *simp*
qed

lemma *injectiveIffExistsInverse*:
$Injective(f) = (\exists\, g\ :\ \forall\, x \in DOMAIN\, f\ :\ g[f[x]] = x)$
by (*auto intro*: *inverseThenInjective dest*: *injectiveInverse*)

lemma *inverseOfInjectiveSurjective*:
assumes *f*: *Injective*(*f*)

shows *Surjective*(*Inverse*(*f*), *DOMAIN f*)
proof (*rule surjectiveI*)
fix *x*
assume *x*: $x \in$ *DOMAIN f*
with *f* **have** $x =$ *Inverse*(*f*)[*f*[*x*]] **by** (*rule sym*[*OF injectiveInverse*])
with *x* **show** $\exists\, y \in$ *DOMAIN Inverse*(*f*) : $x =$ *Inverse*(*f*)[*y*] **by** *auto*
qed

The inverse of a bijection is a bijection.

lemma *inverseBijections*:
assumes *f*: $f \in$ *Bijections*(*S*,*T*)
shows *Inverse*(*f*) $\in$ *Bijections*(*T*,*S*)
proof
from *f* **have** $f \in$ *Surjections*(*S*,*T*) **by** (*simp add*: *Bijections-def*)
hence *rng*: *Range*(*f*) = *T* **by** (*rule surjectionsRange*)
from *f* **have** *dom*: *DOMAIN f* = *S* **by** (*auto elim*: *bijectionsE*)
from *dom rng inverseFcnSet* **show** *Inverse*(*f*) $\in [T \to S]$ **by** *auto*
next
from *f* **have** *Injective*(*f*) **by** (*auto elim*: *bijectionsE*)
hence *Surjective*(*Inverse*(*f*), *DOMAIN f*) **by** (*rule inverseOfInjectiveSurjective*)
with *f* **show** *Surjective*(*Inverse*(*f*),*S*) **by** (*auto elim*: *bijectionsE*)
qed (*rule inverseInjective*)

end

5 Peano's axioms and natural numbers

theory *Peano*
imports *FixedPoints Functions*
begin

As a preparation for the definition of numbers and arithmetic in TLA$^+$, we state Peano's axioms for natural numbers and prove the existence of a structure satisfying them. The presentation of the axioms is somewhat simplified compared to the TLA$^+$ book. (Moreover, the existence of such a structure is assumed, but not proven in the book.)

5.1 The Peano Axioms

definition *PeanoAxioms* :: [*c*,*c*,*c*] $\Rightarrow$ *c* **where**
— parameters: the set of natural numbers, zero, and succ function
PeanoAxioms(*N*,*Z*,*Sc*) $\equiv$
$Z \in N$
$\wedge$ $Sc \in [N \to N]$
$\wedge$ $(\forall\, n \in N : Sc[n] \neq Z)$
$\wedge$ $(\forall\, m,n \in N$: $Sc[m] = Sc[n] \Rightarrow m = n)$

$\wedge\ (\forall\, S \in \mathit{SUBSET}\ N : Z \in S \wedge (\forall\, n{\in}S : Sc[n] \in S) \Rightarrow N \subseteq S)$

The existence of a structure satisfying Peano's axioms is proven following the standard ZF construction where $\{\}$ is zero, $i \cup \{i\}$ is taken as the successor of any natural number i, and the set of natural numbers is defined as the least set that contains zero and is closed under successor (this is a subset of the infinity set asserted to exist in ZF set theory). In TLA$^+$, natural numbers are defined by a sequence of *CHOOSE*'s below, so there is no commitment to that particular structure.

theorem *peanoExists*: $\exists\, N,Z,Sc : \mathit{PeanoAxioms}(N,Z,Sc)$
proof –
  let $?sc = \lambda n.\ \mathit{addElt}(n,n)$ — successor *operator*
  def $\mathit{expand} \equiv \lambda S.\ \{\{\}\} \cup \{\ ?sc(n) : n \in S\}$
  def $N \equiv \mathit{lfp}(\mathit{infinity},\ \mathit{expand})$
  def $Z \equiv \{\}$
  def $Sc \equiv [n \in N \mapsto\ ?sc(n)]$ — successor *function*
  have *mono*: $\mathit{Monotonic}(\mathit{infinity},\ \mathit{expand})$
    using *infinity* **by** (*auto simp*: *Monotonic-def expand-def*)
  hence *expandN*: $\mathit{expand}(N) \subseteq N$
    by (*unfold N-def*, *rule lfpPreFP*)
  from *expandN* **have** *1*: $Z \in N$
    by (*auto simp*: *expand-def Z-def*)
  have *2*: $Sc \in [N \to N]$
  proof (*unfold Sc-def*, *rule functionInFuncSet*)
    show $\forall\, n \in N :\ ?sc(n) \in N$ **using** *expandN* **by** (*auto simp*: *expand-def*)
  qed
  have *3*: $\forall\, m{\in}N : Sc[m] \neq Z$
    unfolding *Z-def Sc-def* **by** *auto*
  have *4*: $\forall\, m,n \in N : Sc[m] = Sc[n] \Rightarrow m = n$
  proof (*clarify*)
    fix $m\ n$
    assume $m \in N$ **and** $n \in N$ **and** $Sc[m] = Sc[n]$
    hence *eq*: $?sc(m) =\ ?sc(n)$ **by** (*simp add*: *Sc-def*)
    show $m = n$
    proof (*rule setEqual*)
      show $m \subseteq n$
      proof (*rule subsetI*)
 fix x
 assume x: $x \in m$ **show** $x \in n$
 proof (*rule contradiction*)
   assume $x \notin n$
   with *x eq* **have** $n \in m$ **by** *auto*
       moreover
   from *eq* **have** $m \in\ ?sc(n)$ **by** *auto*
       ultimately
   show *FALSE* **by** (*blast elim*: *inAsym*)
 qed
     qed

```
    next
      show n ⊆ m
      proof (rule subsetI)
 fix x
 assume x: x ∈ n show x ∈ m
 proof (rule contradiction)
   assume x ∉ m
   with x eq have m ∈ n by auto
          moreover
          from eq have n ∈ ?sc(m) by auto
          ultimately
   show FALSE by (blast elim: inAsym)
 qed
      qed
    qed
  qed
  have 5: ∀ S ∈ SUBSET N : Z ∈ S ∧ (∀ n∈S : Sc[n] ∈ S) ⇒ N ⊆ S
  proof (clarify del: subsetI)
    fix S
    assume sub: S ⊆ N and Z: Z ∈ S and Sc: ∀ n∈S : Sc[n] ∈ S
    show N ⊆ S
    proof (unfold N-def, rule lfpLB)
      show expand(S) ⊆ S
      proof (auto simp: expand-def)
 from Z show {} ∈ S by (simp add: Z-def)
      next
 fix n
 assume n: n ∈ S
 with Sc have Sc[n] ∈ S ..
 moreover
 from n sub have n ∈ N by auto
 hence Sc[n] = ?sc(n) by (simp add: Sc-def)
 ultimately show ?sc(n) ∈ S by simp
      qed
    next
      have N ⊆ infinity
 by (unfold N-def, rule lfpSubsetDomain)
      with sub show S ⊆ infinity by auto
    qed
  qed
  from 1 2 3 4 5 have PeanoAxioms(N,Z,Sc)
    unfolding PeanoAxioms-def by blast
  thus ?thesis by blast
qed

lemma peanoInduct:
  assumes pa: PeanoAxioms(N,Z,Sc)
  and S ⊆ N and Z ∈ S and ⋀n. n ∈ S ⟹ Sc[n] ∈ S
  shows N ⊆ S
```

proof −
  from *pa* **have** $\forall S \in SUBSET\ N : Z \in S \land (\forall n{\in}S : Sc[n] \in S) \Rightarrow N \subseteq S$
    unfolding *PeanoAxioms-def* **by** *blast*
  with *assms* **show** *?thesis* **by** *blast*
qed

5.2 Natural numbers: definition and elementary theorems

The structure of natural numbers is now defined to be some set, zero, and successor satisfying Peano's axioms.

definition *Succ* :: *c*
where $Succ \equiv CHOOSE\ Sc : \exists\, N,Z : PeanoAxioms(N,Z,Sc)$

definition *Nat* :: *c*
where $Nat \equiv DOMAIN\ Succ$

definition *zero* :: *c* (*0*)
where $0 \equiv CHOOSE\ Z : PeanoAxioms(Nat, Z, Succ)$

abbreviation $one \equiv Succ[0]$
notation *one* (*1*)
abbreviation $two \equiv Succ[1]$
notation *two* (*2*)
abbreviation $three \equiv Succ[2]$
notation *three* (*3*)
abbreviation $four \equiv Succ[3]$
notation *four* (*4*)
abbreviation $five \equiv Succ[4]$
notation *five* (*5*)
abbreviation $six \equiv Succ[5]$
notation *six* (*6*)
abbreviation $seven \equiv Succ[6]$
notation *seven* (*7*)
abbreviation $eight \equiv Succ[7]$
notation *eight* (*8*)
abbreviation $nine \equiv Succ[8]$
notation *nine* (*9*)
abbreviation $ten \equiv Succ[9]$
notation *ten* (*10*)
abbreviation $eleven \equiv Succ[10]$
notation *eleven* (*11*)
abbreviation $twelve \equiv Succ[11]$
notation *twelve* (*12*)
abbreviation $thirteen \equiv Succ[12]$
notation *thirteen* (*13*)
abbreviation $fourteen \equiv Succ[13]$
notation *fourteen* (*14*)
abbreviation $fifteen \equiv Succ[14]$
notation *fifteen* (*15*)

```
lemma peanoNatZeroSucc: PeanoAxioms(Nat, 0, Succ)
proof −
  have ∃ N,Z : PeanoAxioms(N,Z,Succ)
  proof (unfold Succ-def, rule chooseI-ex)
    from peanoExists show ∃ Sc,N,Z : PeanoAxioms(N,Z,Sc) by blast
  qed
  then obtain N Z where PNZ: PeanoAxioms(N,Z,Succ) by blast
  hence Succ ∈ [N → N]
    by (simp add: PeanoAxioms-def)
  hence N = Nat
    by (simp add: Nat-def)
  with PNZ have PeanoAxioms(Nat, Z, Succ) by simp
  thus ?thesis by (unfold zero-def, rule chooseI)
qed

lemmas
  setEqualI [where A = Nat, standard, intro!]
  setEqualI [where B = Nat, standard, intro!]

lemma zeroIsNat [intro!,simp]: 0 ∈ Nat
using peanoNatZeroSucc by (simp add: PeanoAxioms-def)

lemma succInNatNat [intro!,simp]: Succ ∈ [Nat → Nat]
using peanoNatZeroSucc by (simp add: PeanoAxioms-def)

lemma succIsAFcn [intro!,simp]: isAFcn(Succ)
using succInNatNat by blast

— DOMAIN Succ = Nat
lemmas domainSucc [intro!,simp] = funcSetDomain[OF succInNatNat]
— n ∈ Nat =⇒ Succ[n] ∈ Nat
lemmas succIsNat [intro!,simp] = funcSetValue[OF succInNatNat]

lemma oneIsNat [intro!,simp]: 1 ∈ Nat
by simp

lemma twoIsNat [intro!,simp]: 2 ∈ Nat
by simp

lemma [simp]:
  assumes n ∈ Nat
  shows (Succ[n] = 0) = FALSE
using assms peanoNatZeroSucc by (auto simp: PeanoAxioms-def)

lemma [simp]:
  assumes n: n ∈ Nat
  shows (0 = Succ[n]) = FALSE
```

using *assms* **by** (*auto dest*: *sym*)

lemma *succNotZero* :
  $[\![Succ[n] = 0;\ n \in Nat]\!] \Longrightarrow P$
  $[\![0 = Succ[n];\ n \in Nat]\!] \Longrightarrow P$
by (*simp+*)

lemma *succInj* [*dest*]:
  assumes $Succ[m] = Succ[n]$ **and** $m \in Nat$ **and** $n \in Nat$
  shows $m{=}n$
using *peanoNatZeroSucc assms* **by** (*auto simp*: *PeanoAxioms-def*)

lemma *succInjIff* [*simp*]:
  assumes $m \in Nat$ **and** $n \in Nat$
  shows $(Succ[m] = Succ[n]) = (m = n)$
using *assms* **by** *auto*

lemma *natInduct*:
  assumes *z*: $P(0)$
  and *sc*: $\bigwedge n.\ [\![n \in Nat;\ P(n)]\!] \Longrightarrow P(Succ[n])$
  shows $\forall\, n{\in}Nat : P(n)$
proof –
  let $?P = \{n \in Nat : P(n)\}$
  from *peanoNatZeroSucc* **have** $Nat \subseteq\ ?P$
    by (*rule peanoInduct*, *auto simp*: *z sc*)
  thus *?thesis* **by** *auto*
qed

— version of above suitable for the inductive reasoning package
lemma *natInductE* [*case-names 0 Succ*, *induct set*: *Nat*]:
  assumes $n \in Nat$ **and** $P(0)$ **and** $\bigwedge n.\ [\![n \in Nat;\ P(n)]\!] \Longrightarrow P(Succ[n])$
  shows $P(n)$
using *bspec*[*OF natInduct*, **where** $P{=}P$] *assms* **by** *blast*

lemma *natCases* [*case-names 0 Succ*, *cases set*: *Nat*]:
  assumes *n*: $n \in Nat$
  and *z*: $n{=}0 \Longrightarrow P$ **and** *sc*: $\bigwedge m.\ [\![m \in Nat;\ n = Succ[m]]\!] \Longrightarrow P$
  shows P
proof –
  from *n* **have** $n{=}0 \lor (\exists\, m \in Nat : n = Succ[m])$
    by (*induct*, *auto*)
  thus *?thesis*
  proof
    assume $n{=}0$ **thus** P **by** (*rule z*)
  next
    assume $\exists\, m{\in}Nat : n = Succ[m]$
    then obtain m **where** $m \in Nat$ **and** $n = Succ[m]$ **..**

thus P **by** (*rule sc*)
qed
qed

lemma *succIrrefl*:
assumes n: $n \in Nat$
shows $Succ[n] \neq n$
using n **by** (*induct*, *auto*)

lemma *succIrreflE* :
$[\![Succ[n] = n; n \in Nat]\!] \Longrightarrow P$
$[\![n = Succ[n]; n \in Nat]\!] \Longrightarrow P$
by (*auto dest*: *succIrrefl*)

lemma *succIrrefl-iff* [*simp*]:
$n \in Nat \Longrightarrow (Succ[n] = n) = FALSE$
$n \in Nat \Longrightarrow (n = Succ[n]) = FALSE$
by (*auto dest*: *succIrrefl*)

— Induction over two parameters along the "diagonal".
lemma *diffInduction*:
assumes $b1$: $\forall m \in Nat : P(m,0)$ **and** $b2$: $\forall n \in Nat : P(0, Succ[n])$
and *step*: $\forall m,n \in Nat : P(m,n) \Rightarrow P(Succ[m], Succ[n])$
shows $\forall m,n \in Nat : P(m,n)$
proof (*rule natInduct*)
show $\forall n \in Nat : P(0,n)$
using *b1 b2* **by** (*intro natInduct*, *auto*)
next
fix m
assume m: $m \in Nat$ **and** *ih*: $\forall n \in Nat : P(m,n)$
show $\forall n \in Nat : P(Succ[m],n)$
proof (*rule bAllI*)
fix n
assume $n \in Nat$ **thus** $P(Succ[m],n)$
proof (*cases*)
case *0* **with** *b1 m* **show** *?thesis* **by** *auto*
next
case *Succ* **with** *step ih m* **show** *?thesis* **by** *auto*
qed
qed
qed

lemma *diffInduct*:
assumes n: $n \in Nat$ **and** m: $m \in Nat$
and $b1$: $\bigwedge m.\ m \in Nat \Longrightarrow P(m,0)$ **and** $b2$: $\bigwedge n.\ n \in Nat \Longrightarrow P(0, Succ[n])$
and *step*: $\bigwedge m\ n.\ [\![m \in Nat; n \in Nat; P(m,n)\ ]\!] \Longrightarrow P(Succ[m], Succ[n])$
shows $P(m,n)$
proof −

have $\forall\, m,n{\in}Nat : P(m,n)$
  by (*rule diffInduction*, *auto intro*: *b1 b2 step*)
  with *n m* **show** *?thesis* **by** *blast*
qed

lemma *not0-implies-Suc*:
  $[\![n \in Nat;\ n \neq 0]\!] \Longrightarrow \exists\, m \in Nat\colon n = Succ[m]$
by(*rule natCases*, *auto*)

5.3 Initial intervals of natural numbers and "less than"

The set of natural numbers up to (and including) a given n is inductively defined as the smallest set of natural numbers that contains n and that is closed under predecessor.

NB: "less than" is not first-order definable from the Peano axioms, a set-theoretic definition such as the following seems to be unavoidable.

definition *upto* :: $c \Rightarrow c$
where $upto(n) \equiv lfp(Nat, \lambda S.\ \{n\} \cup \{\ k \in Nat : Succ[k] \in S\ \})$

lemmas
  setEqualI [**where** $A = upto(n)$, *standard*, *intro*!]
  setEqualI [**where** $B = upto(n)$, *standard*, *intro*!]

lemma *uptoNat*: $upto(n) \subseteq Nat$
  unfolding *upto-def* **by** (*rule lfpSubsetDomain*)

lemma *uptoPred*:
  assumes *Suc*: $Succ[m] \in upto(n)$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $m \in upto(n)$
proof −
  let $?f = \lambda S.\ \{n\} \cup \{k{\in}Nat : Succ[k] \in S\}$
  from *n* **have** *mono*: $Monotonic(Nat,\ ?f)$
    unfolding *Monotonic-def* **by** *blast*
  from *m Suc* **have** *1*: $m \in\ ?f(upto(n))$ **by** *auto*
  from *mono* **have** *2*: $?f(upto(n)) \subseteq upto(n)$
    unfolding *upto-def* **by** (*rule lfpPreFP*)
  from *1 2* **show** *?thesis* **by** *blast*
qed

lemma *uptoZero*: $upto(0) = \{0\}$
proof (*rule setEqual*)
  have $\{0\} \cup \{\ k \in Nat : Succ[k] \in \{0\}\ \} \subseteq \{0\}$ **by** *auto*
  thus $upto(0) \subseteq \{0\}$
    unfolding *upto-def* **by** (*rule lfpLB*, *auto*)
next
  show $\{0\} \subseteq upto(0)$
    unfolding *upto-def* **by** (*rule lfpGLB*, *auto*)
qed

lemma *uptoSucc*:
assumes *n*: $n \in Nat$
shows $upto(Succ[n]) = upto(n) \cup \{Succ[n]\}$ (**is** *?lhs = ?rhs*)
proof −
let $?preds(S) = \{k \in Nat : Succ[k] \in S\}$
let $?f(S,k) = \{k\} \cup ?preds(S)$
have *mono*: $\bigwedge k.\ k \in Nat \Longrightarrow Monotonic(Nat, \lambda S.\ ?f(S,k))$
by (*auto simp*: *Monotonic-def*)
— "⊆"
from *n* **have** $?preds(?rhs) \subseteq ?f(upto(n), n)$ **by** *auto*
also have $\ldots \subseteq upto(n)$
by (*unfold upto-def*, *rule lfpPreFP*, *rule mono*, *rule n*)
finally have $?f(?rhs, Succ[n]) \subseteq ?rhs$ **by** *auto*
moreover from *n* **have** $?rhs \subseteq Nat$
by (*intro cupLUB*, *auto elim*: *uptoNat*[*THEN subsetD*])
ultimately have *1*: $?lhs \subseteq ?rhs$
by (*unfold upto-def*[**where** *n=Succ*[*n*]], *rule lfpLB*)
— "⊇"
from *n mono* **have** *2*: $?f(?lhs, Succ[n]) \subseteq ?lhs$
unfolding *upto-def* **by** (*intro lfpPreFP*, *blast*)
with *n* **have** $?f(?lhs, n) \subseteq ?lhs$ **by** *auto*
moreover have $?lhs \subseteq Nat$ **by** (*rule uptoNat*)
ultimately have *3*: $upto(n) \subseteq ?lhs$
unfolding *upto-def*[**where** *n=n*] **by** (*rule lfpLB*)
from *2* **have** *4*: $Succ[n] \in ?lhs$ **by** *auto*
from *3 4* **have** $?rhs \subseteq ?lhs$ **by** *auto*
with *1* **show** *?thesis* **by** (*rule setEqual*)
qed

lemma *uptoRefl*:
assumes *n*: $n \in Nat$
shows $n \in upto(n)$
using *n* **proof** (*cases*)
case *0* **thus** *?thesis* **by** (*simp add*: *uptoZero*)
next
case *Succ* **thus** *?thesis* **by** (*auto simp*: *uptoSucc*)
qed

lemma *zeroInUpto*:
assumes *n*: $n \in Nat$
shows $0 \in upto(n)$
using *n* **by** (*induct*, *auto simp*: *uptoZero uptoSucc*)

lemma *SuccNotUptoZero*:
assumes $n \in Nat$ **and** $Succ[n] \in upto(0)$
shows *P*
using *assms* **by** (*auto simp*: *uptoZero*)

lemma *uptoTrans*:
  assumes $k \in upto(m)$ **and** $m \in upto(n)$ **and** $n \in Nat$
  shows $k \in upto(n)$
proof –
  have $\forall n \in Nat : m \in upto(n) \Rightarrow upto(m) \subseteq upto(n)$
    by (*rule natInduct*, *auto simp*: *uptoZero uptoSucc*)
  with *assms* **show** *?thesis* **by** *blast*
qed

lemma *succNotinUpto*:
  assumes n: $n \in Nat$
  shows $Succ[n] \notin upto(n)$
using *n* **proof** (*induct*)
  show $1 \notin upto(0)$ **by** (*auto simp*: *uptoZero*)
next
  fix *n*
  assume n: $n \in Nat$ **and** *ih*: $Succ[n] \notin upto(n)$
  show $Succ[Succ[n]] \notin upto(Succ[n])$
  proof (*auto simp*: *uptoSucc n*)
    assume $Succ[Succ[n]] \in upto(n)$
    with *n* **have** $Succ[n] \in upto(n)$
      by (*auto elim*: *uptoPred*)
    with *ih* **show** *FALSE* **..**
  qed
qed

lemma *uptoLimit*:
  assumes m: $m \in upto(n)$ **and** *suc*: $Succ[m] \notin upto(n)$ **and** n: $n \in Nat$
  shows $m=n$
proof –
  from *m uptoNat* **have** *mNat*: $m \in Nat$ **by** *blast*
  from *n* **have** $\forall m \in Nat$: $m \in upto(n) \wedge Succ[m] \notin upto(n) \Rightarrow m=n$ (**is** $?P(n)$)
    by (*induct*, *auto simp*: *uptoZero uptoSucc*)
  with *mNat m suc* **show** *?thesis* **by** *blast*
qed

lemma *uptoAntisym*:
  assumes *mn*: $m \in upto(n)$ **and** *nm*: $n \in upto(m)$
  shows $m=n$
proof –
  from *mn uptoNat* **have** m: $m \in Nat$ **by** *blast*
  from *nm uptoNat* **have** n: $n \in Nat$ **by** *blast*
  have $\forall m,n \in Nat : m \in upto(n) \wedge n \in upto(m) \Rightarrow m=n$ (**is** $\forall m,n \in Nat : ?P(m,n)$)
  proof (*rule natInduct*)
    show $\forall n \in Nat : ?P(0,n)$ **by** (*auto simp*: *uptoZero*)
  next
    fix *m*
    assume m: $m \in Nat$ **and** *ih*: $\forall n \in Nat : ?P(m,n)$

```
    show ∀ n∈Nat : ?P(Succ[m],n)
    proof (auto simp: uptoSucc m)
      fix n
      assume Succ[m] ∈ upto(n) and n ∈ upto(m)
      from this m have Succ[m] ∈ upto(m) by (rule uptoTrans)
      with m show Succ[m] = n — contradiction
 by (blast dest: succNotinUpto)
    qed
  qed
  with m n mn nm show ?thesis by blast
qed

lemma uptoInj [simp]:
  assumes n: n ∈ Nat and m: m ∈ Nat
  shows (upto(n) = upto(m)) = (n = m)
proof (auto)
  assume 1: upto(n) = upto(m)
  from n have n ∈ upto(n) by (rule uptoRefl)
  with 1 have n ∈ upto(m) by auto
  moreover
  from m have m ∈ upto(m) by (rule uptoRefl)
  with 1 have m ∈ upto(n) by auto
  ultimately
  show n = m by (rule uptoAntisym)
qed

lemma uptoLinear:
  assumes m: m ∈ Nat and n: n ∈ Nat
  shows m ∈ upto(n) ∨ n ∈ upto(m) (is ?P(m,n))
using m proof induct
  from n show ?P(0,n) by (auto simp: zeroInUpto)
next
  fix k
  assume k: k ∈ Nat and ih: ?P(k,n)
  from k show ?P(Succ[k],n)
  proof (auto simp: uptoSucc)
    assume kn: (Succ[k] ∈ upto(n)) = FALSE
    show n ∈ upto(k)
    proof (rule contradiction)
      assume c: n ∉ upto(k)
      with ih have k ∈ upto(n) by simp
      from this kn n have k = n by (rule uptoLimit[simplified])
      with n have n ∈ upto(k) by (simp add: uptoRefl)
      with c show FALSE ..
    qed
  qed
qed
```

5.4 Primitive Recursive Functions

We axiomatize a primitive recursive scheme for functions with one argument and domain on natural numbers. Later, we use it to define addition, multiplication and difference.

axiomatization where
  primrec-nat: $\exists f : isAFcn(f) \wedge DOMAIN\, f = Nat$
    $\wedge\ f[0] = e \wedge (\forall\, n \in Nat : f[Succ[n]] = h(n,f[n]))$

lemma *bprimrec-nat*:
  assumes *e*: $e \in S$ **and** *suc*: $\forall\, n \in Nat : \forall\, x \in S : h(n,x) \in S$
  shows $\exists f \in [Nat \to S] : f[0] = e \wedge (\forall\, n \in Nat\colon f[Succ[n]] = h(n,f[n]))$
proof –
  from *primrec-nat*[*of e h*] **obtain** *f* **where**
    1: *isAFcn*(*f*) **and** *2*: $DOMAIN\, f = Nat$
    and *3*: $f[0] = e$ **and** *4*: $\forall\, n{\in}Nat : f[Succ[n]] = h(n,f[n])$
    by *blast*
  have $\forall\, n{\in}Nat : f[n] \in S$
  proof (*rule natInduct*)
    from *3 e* **show** $f[0] \in S$ **by** *simp*
  next
    fix *n*
    assume $n \in Nat$ **and** $f[n] \in S$
    with *suc 4* **show** $f[Succ[n]] \in S$ **by** *force*
  qed
  with *1 2 3 4* **show** *?thesis*
    by *blast*
qed

lemma *primrec-natE*:
  assumes *e*: $e \in S$ **and** *suc*: $\forall\, n \in Nat : \forall\, x \in S : h(n,x) \in S$
  and *f*: $f = (CHOOSE\ g \in [Nat \to S] : g[0] = e \wedge (\forall\, n \in Nat\colon g[Succ[n]] = h(n,g[n])))$
    (**is** $f = ?g$)
  and *maj*: $[\![\ f \in [Nat \to S];\ f[0] = e;\ \forall\, n \in Nat : f[Succ[n]] = h(n,\ f[n])\ ]\!] \Longrightarrow P$
  shows *P*
proof –
  from *e suc* **have** $\exists\ g \in [Nat \to S] : g[0] = e \wedge (\forall\, n \in Nat\colon g[Succ[n]] = h(n,g[n]))$
    by (*rule bprimrec-nat*)
  hence $?g \in [Nat \to S] \wedge ?g[0] = e \wedge (\forall\, n \in Nat\colon ?g[Succ[n]] = h(n,?g[n]))$
    by (*rule bChooseI2*, *auto*)
  with *f maj* **show** *?thesis* **by** *blast*
qed

lemma *bprimrecType-nat*:
  assumes $e \in S$ **and** $\forall\, n \in Nat : \forall\, x \in S : h(n,x) \in S$
  shows $(CHOOSE\, f \in [Nat \to S] : f[0] = e\ \wedge$

$(\forall\, n \in \mathit{Nat}\colon f[\mathit{Succ}[n]] = h(n,f[n])))$
$\in [\mathit{Nat} \to S]$
by (*rule primrec-natE*[*OF assms*], *auto*)

end

6 Orders on natural numbers

theory *NatOrderings*
imports *Peano*
begin

Using the sets *upto* we can now define the standard ordering on natural numbers. The constant ≤ is defined over the naturals by the axiom (conditional definition) *nat-leq-def* below; it should be defined over other domains as appropriate later on.

We generally define the constant < such that $a{<}b$ iff $a{\leq}b \wedge a{\neq}b$, over any domain.

definition *leq* :: [*c*,*c*] ⇒ *c* (**infixl** <= *50*)

where *nat-leq-def*: (*m* <= *n*) ≡ (*m* ∈ *upto*(*n*))

abbreviation (*input*)
geq :: [*c*,*c*] ⇒ *c* (**infixl** >= *50*)
where *x* >= *y* ≡ *y* <= *x*

notation (*xsymbols*)
leq (**infixl** ≤ *50*) **and**
geq (**infixl** ≥ *50*)

notation (*HTML* **output**)
leq (**infixl** ≤ *50*) **and**
geq (**infixl** ≥ *50*)

6.1 Operator definitions and generic facts about <

definition *less* :: [*c*,*c*] ⇒ *c* (**infixl** < *50*)
where *a* < *b* ≡ *a* ≤ *b* ∧ *a* ≠ *b*

abbreviation (*input*)
greater :: [*c*,*c*] ⇒ *c* (**infixl** > *50*)
where *x* > *y* ≡ *y* < *x*

lemma *boolify-less* [*simp*]: *boolify*(*a* < *b*) = (*a* < *b*)
by (*simp add*: *less-def*)

lemma *less-isBool* [*intro*!,*simp*]: *isBool*(*a*<*b*)

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

lemma *less-imp-leq* [*elim*!]: $a < b \Longrightarrow a \leq b$
unfolding *less-def* **by** *simp*

lemma *less-irrefl* [*simp*]: $(a < a) = FALSE$
unfolding *less-def* **by** *simp*

lemma *less-irreflE* [*elim*!]: $a < a \Longrightarrow R$
by *simp*

lemma *less-not-refl*: $a < b \Longrightarrow a \neq b$
by *auto*

lemma *neq-leq-trans* [*trans*]: $a \neq b \Longrightarrow a \leq b \Longrightarrow a < b$
by (*simp add*: *less-def*)

declare *neq-leq-trans*[*simplified*,*trans*]

lemma *leq-neq-trans* [*trans*,*elim*!]: $a \leq b \Longrightarrow a \neq b \Longrightarrow a < b$
by (*simp add*: *less-def*)

declare *leq-neq-trans*[*simplified*,*trans*]

lemma *leq-neq-iff-less*: $a \leq b \Longrightarrow (a \neq b) = (a < b)$
by *auto*

6.2 Facts about $\leq$ over *Nat*

lemma *nat-boolify-leq* [*simp*]: $boolify(m \leq n) = (m \leq n)$
by (*simp add*: *nat-leq-def*)

lemma *nat-leq-isBool* [*intro*,*simp*]: $isBool(m \leq n)$
by (*simp add*: *nat-leq-def*)

lemma *nat-leq-refl* [*intro*,*simp*]: $n \in Nat \Longrightarrow n \leq n$
unfolding *nat-leq-def* **by** (*rule uptoRefl*)

lemma *eq-leq-bothE*: — reduce equality over integers to double inequality
  assumes $m \in Nat$ **and** $n \in Nat$ **and** $m = n$ **and** $[\![m \leq n; n \leq m]\!] \Longrightarrow P$
  shows P
using *assms* **by** *simp*

lemma *nat-zero-leq* [*simp*]: $n \in Nat \Longrightarrow 0 \leq n$
unfolding *nat-leq-def* **by** (*rule zeroInUpto*)

lemma *nat-leq-zero* [*simp*]: $n \in Nat \Longrightarrow (n \leq 0) = (n = 0)$
by (*simp add*: *nat-leq-def uptoZero*)

lemma *nat-leq-SuccI* [*elim*!,*simp*]:
  assumes $m \leq n$ **and** $m \in Nat$ **and** $n \in Nat$
  shows $m \leq Succ[n]$
using *assms* **by** (*auto simp*: *nat-leq-def uptoSucc*)

lemma *nat-leq-Succ*:
  assumes $m \in Nat$ **and** $n \in Nat$
  shows $(m \leq Succ[n]) = (m \leq n \lor m = Succ[n])$
using *assms* **by** (*auto simp*: *nat-leq-def uptoSucc*)

lemma *nat-leq-SuccE* [*elim*]:
  assumes $m \leq Succ[n]$ **and** $m \in Nat$ **and** $n \in Nat$
    and $m \leq n \Longrightarrow P$ **and** $m = Succ[n] \Longrightarrow P$
  shows P
using *assms* **by** (*auto simp*: *nat-leq-Succ*)

lemma *nat-leq-limit*:
  assumes $m \leq n$ **and** $\neg(Succ[m] \leq n)$ **and** $m \in Nat$ **and** $n \in Nat$
  shows $m=n$
using *assms* **by** (*auto simp*: *nat-leq-def intro*: *uptoLimit*)

lemma *nat-leq-trans* [*trans*]:
  assumes $k \leq m$ **and** $m \leq n$ **and** $k \in Nat$ **and** $m \in Nat$ **and** $n \in Nat$
  shows $k \leq n$
using *assms* **by** (*auto simp*: *nat-leq-def elim*: *uptoTrans*)

lemma *nat-leq-antisym*:
  assumes $m \leq n$ **and** $n \leq m$ **and** $m \in Nat$ **and** $n \in Nat$
  shows $m = n$
using *assms* **by** (*auto simp add*: *nat-leq-def elim*: *uptoAntisym*)

lemma *nat-Succ-not-leq-self* [*simp*]:
  assumes *n*: $n \in Nat$
  shows $(Succ[n] \leq n) = FALSE$
using *n* **by** (*auto dest*: *nat-leq-antisym*)

lemma *nat-Succ-leqD*:
  assumes *leq*: $Succ[m] \leq n$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $m \leq n$
proof −
  from *m* **have** $m \leq Succ[m]$ **by** *simp*
  with *leq m n* **show** *?thesis* **by** (*elim nat-leq-trans*, *auto*)
qed

lemma *nat-Succ-leq-Succ*:

  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(Succ[m] \leq Succ[n]) = (m \leq n)$

using *m n* **by** (*auto simp*: *nat-leq-Succ intro*: *nat-leq-limit elim*: *nat-Succ-leqD*)

lemma *nat-leq-linear*: $[\![m \in Nat; n \in Nat]\!] \Longrightarrow m \leq n \lor n \leq m$
unfolding *nat-leq-def* **using** *uptoLinear* **.**

lemma *nat-leq-cases*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  and *leq*: $m \leq n \Longrightarrow P$ **and** *geq*: $[\![n \leq m; n \neq m]\!] \Longrightarrow P$
  shows *P*
proof (*cases* $m \leq n$)
  case *True* **thus** *P* **by** (*rule leq*)
next
  case *False*
  with *m n* **have** *nm*: $n \leq m$ **by** (*blast dest*: *nat-leq-linear*)
  thus *P*
  proof (*cases* $n=m$)
    case *True*
    with *m* **have** $m \leq n$ **by** *simp*
    thus *P* **by** (*rule leq*)
  next
    case *False*
    with *nm* **show** *P* **by** (*rule geq*)
  qed
qed

lemma *nat-leq-induct*: — sometimes called "complete induction"
  assumes $P(0)$
  and $\forall n \in Nat : (\forall m \in Nat : m \leq n \Rightarrow P(m)) \Rightarrow P(Succ[n])$
  shows $\forall n \in Nat : P(n)$
proof −
  from *assms* **have** $\forall n \in Nat : \forall m \in Nat : m \leq n \Rightarrow P(m)$
    by (*intro natInduct*, *auto simp*: *nat-leq-Succ*)
  thus *?thesis* **by** (*blast dest*: *nat-leq-refl*)
qed

lemma *nat-leq-inductE*:
  assumes $n \in Nat$
  and $P(0)$ **and** $\bigwedge n.\ [\![n \in Nat; \forall m \in Nat : m \leq n \Rightarrow P(m)]\!] \Longrightarrow P(Succ[n])$
  shows $P(n)$
using *assms* **by** (*blast dest*: *nat-leq-induct*)

6.3 Facts about < over *Nat*

lemma *nat-Succ-leq-iff-less* [*simp*]:
  assumes $m \in Nat$ **and** $n \in Nat$
  shows $(Succ[m] \leq n) = (m < n)$
using *assms* **by** (*auto simp*: *less-def dest*: *nat-Succ-leqD nat-leq-limit*)

— alternative definition of < over *Nat*

lemmas *nat-less-iff-Succ-leq* = *sym*[*OF nat-Succ-leq-iff-less*, *standard*]

Reduce $\leq$ to $<$.

lemma *nat-leq-less*: — premises needed for *isBool*($m \leq n$) and reflexivity
  assumes $m \in Nat$ **and** $n \in Nat$
  shows $m \leq n = (m < n \vee m = n)$
using *assms* **by** (*auto simp*: *less-def*)

lemma *nat-less-Succ-iff-leq* [*simp*]:
  assumes $m \in Nat$ **and** $n \in Nat$
  shows $(m < Succ[n]) = (m \leq n)$
using *assms*
by (*simp del*: *nat-Succ-leq-iff-less add*: *nat-less-iff-Succ-leq nat-Succ-leq-Succ*)

lemmas *nat-leq-iff-less-Succ* = *sym*[*OF nat-less-Succ-iff-leq*, *standard*]

lemma *nat-not-leq-one*:
  assumes $n \in Nat$
  shows $(\neg(1 \leq n)) = (n = 0)$
using *assms* **by** (*cases*, *auto*)

declare *nat-not-leq-one*[*simplified*,*simp*]

$<$ and *Succ*.

lemma *nat-Succ-less-mono*:
  assumes $m \in Nat$ **and** $n \in Nat$
  shows $(Succ[m] < Succ[n]) = (m < n)$
using *assms* **by** *simp*

lemma *nat-Succ-less-SuccE*:
  assumes $Succ[m] < Succ[n]$ **and** $m \in Nat$ **and** $n \in Nat$ **and** $m < n \Longrightarrow P$
  shows P
using *assms* **by** *simp*

lemma *nat-not-less0* [*simp*]:
  assumes $n \in Nat$
  shows $(n < 0) = FALSE$
using *assms* **by** (*auto simp*: *less-def*)

lemma *nat-less0E* :
  assumes $n < 0$ **and** $n \in Nat$
  shows P
using *assms* **by** *simp*

lemma *nat-less-SuccI*:
  assumes $m < n$ **and** $m \in Nat$ **and** $n \in Nat$
  shows $m < Succ[n]$
using *assms* **by** *auto*

lemma *nat-Succ-lessD*:
  assumes *1*: $Succ[m] < n$ **and** *2*: $m \in Nat$ **and** *3*: $n \in Nat$
  shows $m < n$
using *1*[*unfolded less-def*] *2 3* **by** *simp*

lemma *nat-less-leq-not-leq*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(m < n) = (m \leq n \wedge \neg\, n \leq m)$
using *assms* **by** (*auto simp*: *less-def dest*: *nat-leq-antisym*)

Transitivity.

lemma *nat-less-trans* :
  assumes $k < m$ **and** $m < n$ **and** $k \in Nat$ **and** $m \in Nat$ **and** $n \in Nat$
  shows $k < n$
using *assms* **by** (*auto simp*: *less-def dest*: *nat-leq-trans nat-leq-antisym*)

lemma *nat-less-trans-Succ* [*trans*]:
  assumes *lt1*: $i < j$ **and** *lt2*: $j < k$
    and *i*: $i \in Nat$ **and** *j*: $j \in Nat$ **and** *k*: $k \in Nat$
  shows $Succ[i] < k$
proof −
  from *i j lt1* **have** $Succ[Succ[i]] \leq Succ[j]$ **by** *simp*
  also from *j k lt2* **have** $Succ[j] \leq k$ **by** *simp*
  finally show *?thesis* **using** *i j k* **by** *simp*
qed

lemma *nat-leq-less-trans* [*trans*]:
  assumes $k \leq m$ **and** $m < n$ **and** $k \in Nat$ **and** $m \in Nat$ **and** $n \in Nat$
  shows $k < n$
using *assms* **by** (*auto simp*: *less-def dest*: *nat-leq-trans nat-leq-antisym*)

lemma *nat-less-leq-trans* [*trans*]:
  assumes $k < m$ **and** $m \leq n$ **and** $k \in Nat$ **and** $m \in Nat$ **and** $n \in Nat$
  shows $k < n$
using *assms* **by** (*auto simp*: *less-def dest*: *nat-leq-trans nat-leq-antisym*)

Asymmetry.

lemma *nat-less-not-sym*:
  assumes $m < n$ **and** $m \in Nat$ **and** $n \in Nat$
  shows $(n < m) = FALSE$
using *assms* **by** (*simp add*: *nat-less-leq-not-leq*)

lemma *nat-less-asym*:
  assumes $m < n$ **and** $m \in Nat$ **and** $n \in Nat$ **and** $\neg P \Longrightarrow n < m$
  shows P
proof (*rule contradiction*)
  assume $\neg P$ **with** *assms* **show** *FALSE* **by** (*auto dest*: *nat-less-not-sym*)
qed

Linearity (totality).

lemma *nat-less-linear*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $m < n \vee m = n \vee n < m$
unfolding *less-def* **using** *nat-leq-linear*[*OF m n*] **by** *blast*

lemma *nat-leq-less-linear*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $m \leq n \vee n < m$
using *assms nat-less-linear*[*OF m n*] **by** (*auto simp*: *less-def*)

lemma *nat-less-cases* [*case-names less equal greater*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(m < n \Longrightarrow P) \Longrightarrow (m = n \Longrightarrow P) \Longrightarrow (n < m \Longrightarrow P) \Longrightarrow P$
using *nat-less-linear*[*OF m n*] **by** *blast*

lemma *nat-not-less*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(\neg\ m < n) = (n \leq m)$
using *assms nat-leq-linear*[*OF m n*] **by** (*auto simp*: *less-def dest*: *nat-leq-antisym*)

lemma *nat-not-less-iff-gr-or-eq*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(\neg\ m < n) = (m > n \vee m = n)$
unfolding *nat-not-less*[*OF m n*] **using** *assms* **by** (*auto simp*: *less-def*)

lemma *nat-not-less-eq*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(\neg\ m < n) = (n < Succ[m])$
unfolding *nat-not-less*[*OF m n*] **using** *assms* **by** *simp*

lemma *nat-not-leq*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(\neg\ m \leq n) = (n < m)$
using *assms* **by** (*simp add*: *sym*[*OF nat-not-less*])

— often useful, but not active by default
lemmas *nat-not-order-simps*[*simplified*] = *nat-not-less nat-not-leq*

lemma *nat-not-leq-eq*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(\neg\ m \leq n) = (Succ[n] \leq m)$
unfolding *nat-not-leq*[*OF m n*] **using** *assms* **by** *simp*

lemma *nat-neq-iff*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $m \neq n = (m < n \vee n < m)$
using *assms nat-less-linear*[*OF m n*] **by** *auto*

lemma *nat-neq-lessE*:

assumes $m \neq n$ **and** $m \in Nat$ **and** $n \in Nat$
shows $(m < n \Longrightarrow R) \Longrightarrow (n < m \Longrightarrow R) \Longrightarrow R$
using *assms* **by** (*auto simp*: *nat-neq-iff* [*simplified*])

lemma *nat-antisym-conv1*:
assumes $\neg(m < n)$ **and** $m \in Nat$ **and** $n \in Nat$
shows $(m \leq n) = (m = n)$
using *assms* **by** (*auto simp*: *nat-leq-less*)

lemma *nat-antisym-conv2*:
assumes $m \leq n$ **and** $m \in Nat$ **and** $n \in Nat$
shows $(\neg\ m < n) = (m = n)$
using *assms* **by** (*auto simp*: *nat-antisym-conv1*)

lemma *nat-antisym-conv3*:
assumes $\neg\ n < m$ **and** $m \in Nat$ **and** $n \in Nat$
shows $(\neg\ m < n) = (m = n)$
using *assms* **by** (*auto simp*: *nat-not-order-simps elim*: *nat-leq-antisym*)

lemma *nat-not-lessD*:
assumes $\neg(m < n)$ **and** $m \in Nat$ **and** $n \in Nat$
shows $n \leq m$
using *assms* **by** (*simp add*: *nat-not-order-simps*)

lemma *nat-not-lessI*:
assumes $n \leq m$ **and** $m \in Nat$ **and** $n \in Nat$
shows $\neg(m < n)$
using *assms* **by** (*simp add*: *nat-not-order-simps*)

lemma *nat-gt0-not0* :
assumes $n \in Nat$
shows $(0 < n) = (n \neq 0)$
using *assms* **by** (*auto simp*: *nat-neq-iff* [*simplified*])

lemmas *nat-neq0-conv* = *sym*[*OF nat-gt0-not0*, *standard*]

Introduction properties

lemma *nat-less-Succ-self* :
assumes $n \in Nat$
shows $n < Succ[n]$
using *assms* **by** *simp*

lemma *nat-zero-less-Succ* :
assumes $n \in Nat$
shows $0 < Succ[n]$
using *assms* **by** *simp*

Elimination properties.

lemma *nat-less-Succ*:

assumes $m \in Nat$ **and** $n \in Nat$
shows $(m < Succ[n]) = (m < n \vee m = n)$
using *assms* **by** (*simp add*: *nat-leq-less*)

lemma *nat-less-SuccE*:
assumes $m < Succ[n]$ **and** $m \in Nat$ **and** $n \in Nat$
and $m < n \Longrightarrow P$ **and** $m = n \Longrightarrow P$
shows P
using *assms* **by** (*auto simp*: *nat-leq-less*)

lemma *nat-less-one* :
assumes $n \in Nat$
shows $(n < 1) = (n = 0)$
using *assms* **by** *simp*

"Less than" is antisymmetric, sort of.

lemma *nat-less-antisym*:
assumes m: $m \in Nat$ **and** n: $n \in Nat$
shows $[\![\neg(n < m); n < Succ[m]]\!] \Longrightarrow m = n$
using *assms* **by** (*auto simp*: *nat-not-order-simps elim*: *nat-leq-antisym*)

Lifting $<$ monotonicity to $\leq$ monotonicity.

lemma *less-mono-imp-leq-mono*:
assumes i: $i \in Nat$ **and** j: $j \in Nat$ **and** f: $\forall n \in Nat : f(n) \in Nat$
and ij: $i \leq j$ **and** *mono*: $\bigwedge i\ j.\ [\![i \in Nat; j \in Nat; i<j]\!] \Longrightarrow f(i) < f(j)$
shows $f(i) \leq f(j)$
using *ij* **proof** (*auto simp*: *nat-leq-less*[*OF i j*])
assume $i < j$
with $i\ j$ **have** $f(i) < f(j)$ **by** (*rule mono*)
thus $f(i) \leq f(j)$ **by** (*simp add*: *less-imp-leq*)
next
from $j\ f$ **show** $f(j) \leq f(j)$ **by** *auto*
qed

Inductive (?) properties.

lemma *nat-Succ-lessI*:
assumes $m \in Nat$ **and** $n \in Nat$ **and** $m < n$ **and** $Succ[m] \neq n$
shows $Succ[m] < n$
using *assms* **by** (*simp add*: *leq-neq-iff-less*[*simplified*])

lemma *nat-lessE*:
assumes *major*: $i < k$ **and** i: $i \in Nat$ **and** k: $k \in Nat$
obtains j **where** $j \in Nat$ **and** $i \leq j$ **and** $k = Succ[j]$
proof −
from *k major* **have** $\exists j{\in}Nat : i \leq j \wedge k = Succ[j]$
proof (*induct k*)
case *0* **with** i **show** *?case* **by** *simp*
next
fix n

assume *n*: $n \in Nat$ **and** *1*: $i < Succ[n]$
and *ih*: $i < n \Longrightarrow \exists j \in Nat : i \leq j \land n = Succ[j]$
from *i n 1* **have** $i < n \lor i = n$ **by** (*simp add*: *nat-leq-less*)
thus $\exists j \in Nat : i \leq j \land Succ[n] = Succ[j]$
proof
assume $i < n$
then obtain *j* **where** $j \in Nat$ **and** $i \leq j$ **and** $n = Succ[j]$
by (*blast dest*: *ih*)
with *i* **have** $Succ[j] \in Nat$ **and** $i \leq Succ[j]$ **and** $Succ[n] = Succ[Succ[j]]$
by *auto*
thus *?thesis* **by** *blast*
next
assume $i = n$
with *i* **show** *?thesis* **by** *blast*
qed
qed
with *that* **show** *?thesis* **by** *blast*
qed

lemma *nat-Succ-lessE*:
assumes *major*: $Succ[i] < k$ **and** *i*: $i \in Nat$ **and** *k*: $k \in Nat$
obtains *j* **where** $j \in Nat$ **and** $i < j$ **and** $k = Succ[j]$
using *assms* **by** (*auto elim*: *nat-lessE*)

lemma *nat-gt0-implies-Succ*:
assumes *1*: $0 < n$ **and** *2*: $n \in Nat$
shows $\exists m : m \in Nat \land n = Succ[m]$
using *2 1* **by** (*cases*, *auto*)

lemma *nat-gt0-iff-Succ*:
assumes *n*: $n \in Nat$
shows $(0 < n) = (\exists m \in Nat: n = Succ[m])$
using *n* **by** (*auto dest*: *nat-gt0-implies-Succ*)

lemma *nat-less-Succ-eq-0-disj*:
assumes $m \in Nat$ **and** $n \in Nat$
shows $(m < Succ[n]) = (m = 0 \lor (\exists j \in Nat: m = Succ[j] \land j < n))$
using *assms* **by** (*induct m*, *auto*)

lemma *nat-less-antisym-false*: $[\![m < n; m \in Nat; n \in Nat]\!] \Longrightarrow n < m = FALSE$
unfolding *less-def* **using** *nat-leq-antisym* **by** *auto*

lemma *nat-less-antisym-leq-false*: $[\![m < n; m \in Nat; n \in Nat]\!] \Longrightarrow n \leq m = FALSE$
unfolding *less-def* **using** *nat-leq-antisym*[*of m n*] **by** *auto*

6.4 Intervals of natural numbers

definition *natInterval* :: $[c,c] \Rightarrow c$ ((- .. -) [*90*,*90*] *70*)

where $m\ ..\ n \equiv \{\ k \in Nat : m \leq k \wedge k \leq n\ \}$

lemma *inNatIntervalI* [*intro*!,*simp*]:
  assumes $k \in Nat$ **and** $m \leq k$ **and** $k \leq n$
  shows $k \in m\ ..\ n$
using *assms* **by** (*simp add*: *natInterval-def*)

lemma *inNatIntervalE* [*elim*]:
  assumes *1*: $k \in m\ ..\ n$ **and** *2*: $[\![k \in Nat;\ m \leq k;\ k \leq n]\!] \Longrightarrow P$
  shows P
using *1* **by** (*intro 2*, *auto simp add*: *natInterval-def*)

lemma *inNatInterval-iff*: $(k \in m\ ..\ n) = (k \in Nat \wedge m \leq k \wedge k \leq n)$
using *assms* **by** *auto*

lemmas
  setEqualI [**where** $A = m\ ..\ n$, *standard*, *intro*]
  setEqualI [**where** $B = m\ ..\ n$, *standard*, *intro*]

lemma *lowerInNatInterval* [*iff*]:
  assumes $m \leq n$ **and** $m \in Nat$
  shows $m \in m\ ..\ n$
using *assms* **by** (*simp add*: *natInterval-def*)

lemma *upperInNatInterval* [*iff*]:
  assumes $m \leq n$ **and** $n \in Nat$
  shows $n \in m\ ..\ n$
using *assms* **by** (*simp add*: *natInterval-def*)

lemma *gtNotinNatInterval*:
  assumes *gt*: $m > n$ **and** *k*: $k \in m\ ..\ n$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows P
proof −
  from *k* **have** *1*: $m \leq k$ **and** *2*: $k \leq n$ **and** *3*: $k \in Nat$ **by** *auto*
  from *1 2 m 3 n* **have** $m \leq n$ **by** (*rule nat-leq-trans*)
  with *m n* **have** $\neg(n < m)$ **by** (*simp add*: *nat-not-order-simps*)
  from *this gt* **show** *?thesis* **..**
qed

lemma *natIntervalIsEmpty*:
  assumes $m \in Nat$ **and** $n \in Nat$ **and** $m > n$
  shows $m\ ..\ n = \{\}$
using *assms* **by** (*blast dest*: *gtNotinNatInterval*)

lemma *natIntervalEmpty-iff*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(m\ ..\ n = \{\}) = (m > n)$
proof (*auto dest*: *natIntervalIsEmpty*[*OF m n*])
  assume *mt*: $m\ ..\ n = \{\}$

show $n < m$
proof (*rule contradiction*)
assume $\neg(n < m)$
with *m n* **have** $m \leq n$ **by** (*simp add*: *nat-not-order-simps*)
from *this m* **have** $m \in m .. n$ **by** (*rule lowerInNatInterval*)
with *mt* **show** *FALSE* **by** *blast*
qed
qed

lemma *natIntervalSingleton* [*simp*]:
assumes $n \in Nat$
shows $n .. n = \{n\}$
using *assms* **by** (*auto dest*: *nat-leq-antisym*)

lemma *natIntervalSucc* [*simp*]:
assumes $m \in Nat$ **and** $n \in Nat$ **and** $m \leq Succ[n]$
shows $m .. Succ[n] = addElt(Succ[n], m .. n)$
using *assms* **by** (*auto simp*: *natInterval-def*)

lemma *succNatInterval*:
assumes $m \in Nat$ **and** $n \in Nat$
shows $Succ[m] .. n = (m .. n \setminus \{m\})$
using *assms* **by** (*auto simp*: *natInterval-def*)

lemma *natIntervalEqual-iff* :
assumes *k*: $k \in Nat$ **and** *l*: $l \in Nat$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $(k .. l = m .. n) = ((k > l \land m > n) \lor (k = m \land l = n))$ (**is** *?lhs* = *?rhs*)
proof −
have *1*: *?lhs* $\Rightarrow$ *?rhs*
proof
assume *eq*: *?lhs*
show *?rhs*
proof (*cases* $k .. l = \{\}$)
case *True*
with *k l* **have** $k > l$ **by** (*simp only*: *natIntervalEmpty-iff*)
moreover
from *True eq m n* **have** $m > n$ **by** (*simp only*: *natIntervalEmpty-iff*)
ultimately
show *?rhs* **by** *blast*
next
case *False*
with *k l* **have** *11*: $k \leq l$ **by** (*simp only*: *natIntervalEmpty-iff nat-not-less*)
from *False eq m n* **have** *12*: $m \leq n$ **by** (*simp only*: *natIntervalEmpty-iff nat-not-less*)
from *11 k eq* **have** *13*: $m \leq k$ **by** *auto*
from *12 m eq* **have** *14*: $k \leq m$ **by** *auto*
from *14 13 k m* **have** *15*: $k = m$ **by** (*rule nat-leq-antisym*)
from *11 l eq* **have** *16*: $l \leq n$ **by** *auto*

```
      from 12 n eq have 17: n ≤ l by auto
      from 16 17 l n have l = n by (rule nat-leq-antisym)
      with 15 show ?rhs by blast
    qed
  qed
  have 2: ?rhs ⇒ ?lhs
  proof auto
    assume lk: l < k and nm: n < m
    from k l lk have k .. l = {} by (rule natIntervalIsEmpty)
    moreover
    from m n nm have m .. n = {} by (rule natIntervalIsEmpty)
    ultimately
    show ?lhs by auto
  qed
  from 1 2 show ?thesis by blast
qed

lemma zerotoInj [simp]:
  assumes l ∈ Nat and m ∈ Nat and n ∈ Nat
  shows (0 .. l = m .. n) = (m=0 ∧ l=n)
using assms by (auto simp: natIntervalEqual-iff)

lemma zerotoInj' [simp]:
  assumes k ∈ Nat and l ∈ Nat and n ∈ Nat
  shows (k .. l = 0 .. n) = (k=0 ∧ l=n)
using assms by (auto simp: natIntervalEqual-iff)

lemma zerotoEmpty [simp]:
  assumes m ∈ Nat
  shows Succ[m] .. 0 = {}
using assms by auto

lemma onetoInj [simp]:
  assumes l ∈ Nat and m ∈ Nat and n ∈ Nat and l≠0 ∨ m=1
  shows (1 .. l = m .. n) = (m=1 ∧ l=n)
using assms by (auto simp: natIntervalEqual-iff)

lemma onetoInj' [simp]:
  assumes k ∈ Nat and l ∈ Nat and n ∈ Nat and n≠0 ∨ k=1
  shows (k .. l = 1 .. n) = (k=1 ∧ l=n)
using assms by (auto simp: natIntervalEqual-iff)

lemma SuccInNatIntervalSucc:
  assumes m ≤ k and k ∈ Nat and m ∈ Nat and n ∈ Nat
  shows (Succ[k] ∈ m .. Succ[n]) = (k ∈ m .. n)
using assms by auto

lemma SuccInNatIntervalSuccSucc:
  assumes k ∈ Nat and m ∈ Nat and n ∈ Nat
```

shows ($Succ[k] \in Succ[m]$ .. $Succ[n]$) = ($k \in m$ .. n)
using *assms* **by** *auto*

end

7 Arithmetic (except division) over natural numbers

theory *NatArith*
imports *NatOrderings*
begin

ML⟪
structure AlgebraSimps =
Named-Thms(val name = algebra-simps
val description = algebra simplification rules);
⟫

setup *AlgebraSimps.setup*

The rewrites accumulated in *algebra-simps* deal with the classical algebraic structures of groups, rings and family. They simplify terms by multiplying everything out (in case of a ring) and bringing sums and products into a canonical form (by ordered rewriting). As a result these rewrites decide group and ring equalities but also help with inequalities.

Of course it also works for fields, but it knows nothing about multiplicative inverses or division. This should be catered for by *field-simps*.

7.1 Addition of natural numbers

definition *addnat*
where $addnat(m) \equiv CHOOSE\ g \in [Nat \to Nat] : g[0] = m \wedge (\forall x \in Nat : g[Succ[x]] = Succ[g[x]])$

definition *arith-add* :: $[c,c] \Rightarrow c$ (**infixl** + *65*)
where *nat-add-def*: $[\![m \in Nat;\ n \in Nat]\!] \Longrightarrow (m + n) \equiv addnat(m)[n]$

Closure

lemma *addnatType*:
assumes $m \in Nat$ **shows** $addnat(m) \in [Nat \to Nat]$
using *assms* **unfolding** *addnat-def* **by** (*rule bprimrecType-nat*, *auto*)

lemma *addIsNat* [*intro*!,*simp*]:
assumes $m \in Nat$ **and** $n \in Nat$ **shows** $m + n \in Nat$
unfolding *nat-add-def*[*OF assms*] **using** *assms addnatType* **by** *blast*

Base case and Inductive case

lemma *addnat-0*:
  assumes $m \in Nat$ **shows** $addnat(m)[0] = m$
using *assms* **unfolding** *addnat-def* **by** (*rule primrec-natE*, *auto*)

lemma *add-0-nat* [*simp*]:
  assumes $m \in Nat$ **shows** $m + 0 = m$
by (*simp add*: *nat-add-def*[*OF assms*] *addnat-0*[*OF assms*])

lemma *addnat-Succ*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $addnat(m)[Succ[n]] = Succ[addnat(m)[n]]$
proof (*rule primrec-natE*[*OF m*])
  show $addnat(m) = (CHOOSE\ g \in [Nat \to Nat] : g[0] = m \wedge (\forall x \in Nat : g[Succ[x]] = Succ[g[x]]))$
    unfolding *addnat-def* **..**
next
  assume $\forall n \in Nat : addnat(m)[Succ[n]] = Succ[addnat(m)[n]]$
  with *n* **show** $addnat(m)[Succ[n]] = Succ[addnat(m)[n]]$ **by** *blast*
qed (*auto*)

lemma *add-Succ-nat* [*simp*]:
  assumes $m \in Nat$ **and** $n \in Nat$
  shows $m + Succ[n] = Succ[m + n]$
using *assms* **by** (*simp add*: *nat-add-def addnat-Succ*[*OF assms*])

lemma *add-0-left-nat* [*simp*]:
  assumes *n*: $n \in Nat$
  shows $0 + n = n$
using *n* **by** (*induct*, *auto*)

lemma *add-Succ-left-nat* [*simp*]:
  assumes *n*: $n \in Nat$ **and** *m*: $m \in Nat$
  shows $Succ[m] + n = Succ[m + n]$
using *n* **apply** *induct*
using *m* **by** *auto*

lemma *add-Succ-shift-nat*:
  assumes *n*: $n \in Nat$ **and** *m*: $m \in Nat$
  shows $Succ[m] + n = m + Succ[n]$
using *assms* **by** *simp*

Commutativity

lemma *add-commute-nat* [*algebra-simps*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $m + n = n + m$
using *n* **apply** *induct*
using *assms* **by** *auto*

Associativity

lemma *add-assoc-nat* [*algebra-simps*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
  shows $m + (n + p) = (m + n) + p$
using *assms* **by** (*induct*, *simp-all*)

Cancellation rules

lemma *add-left-cancel-nat* [*simp*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
  shows $(m + n = m + p) = (n = p)$
using *assms* **by** (*induct*, *simp-all*)

lemma *add-right-cancel-nat* [*simp*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
  shows $(n + m = p + m) = (n = p)$
using *assms* **by** (*induct*, *simp-all*)

lemma *add-left-commute-nat* [*algebra-simps*]:
  assumes *a*: $a \in Nat$ **and** *b*: $b \in Nat$ **and** *c*: $c \in Nat$
  shows $a + (b + c) = b + (a + c)$
using *assms* **by**(*simp only*: *add-assoc-nat add-commute-nat*)

theorems *add-ac-nat* = *add-assoc-nat add-commute-nat add-left-commute-nat*

Reasoning about $m + n = 0$, etc.

lemma *add-is-0-nat* [*iff*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(m + n = 0) = (m = 0 \wedge n = 0)$
using *m* **apply** (*rule natCases*)
using *n* **by** (*induct*, *auto*)

lemma *add-is-1-nat*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(m + n = 1) = (m = 1 \wedge n = 0 \vee m = 0 \wedge n = 1)$
using *m* **apply** (*rule natCases*)
using *n* **by** (*induct*, *auto*)

lemma *one-is-add-nat*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(1 = m + n) = (m = 1 \wedge n = 0 \vee m = 0 \wedge n = 1)$
using *m* **apply** (*rule natCases*)
using *n* **by** (*induct*, *auto*)+

lemma *add-eq-self-zero-nat* [*simp*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(m + n = m) = (n = 0)$
using *n* **apply** (*rule natCases*)
  using *m* **apply** *simp*

using *m* **apply** *induct* **apply** *auto*
done

7.2 Multiplication of natural numbers

definition *multnat*
where $multnat(m) \equiv CHOOSE\ g \in [Nat \to Nat] : g[0] = 0 \wedge (\forall x \in Nat : g[Succ[x]] = g[x] + m)$

definition *mult* :: $[c,c] \Rightarrow c$ (**infixl** $*$ *70*)
where *nat-mult-def*: $[\![m \in Nat; n \in Nat]\!] \Longrightarrow m * n \equiv multnat(m)[n]$

Closure

lemma *multnatType*:
assumes $m \in Nat$ **shows** $multnat(m) \in [Nat \to Nat]$
unfolding *multnat-def* **by** (*rule bprimrecType-nat*, *auto simp*: *assms*)

lemma *multIsNat* [*intro*!, *simp*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $m * n \in Nat$
unfolding *nat-mult-def*[*OF assms*] **using** *assms multnatType* **by** *blast*

Base case and Inductive step

lemma *multnat-0*:
assumes $m \in Nat$ **shows** $multnat(m)[0] = 0$
unfolding *multnat-def* **by** (*rule primrec-natE*, *auto simp*: *assms*)

lemma *mult-0-nat* [*simp*]: — neutral element
assumes *n*: $n \in Nat$ **shows** $n * 0 = 0$
by (*simp add*: *nat-mult-def*[*OF assms*] *multnat-0*[*OF assms*])

lemma *multnat-Succ*:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $multnat(m)[Succ[n]] = multnat(m)[n] + m$
proof (*rule primrec-natE*)
show $multnat(m) = (CHOOSE\ g \in [Nat \to Nat] : g[0] = 0 \wedge (\forall x \in Nat : g[Succ[x]] = g[x] + m))$
unfolding *multnat-def* **..**
next
assume $\forall n \in Nat : multnat(m)[Succ[n]] = multnat(m)[n] + m$
with *n* **show** $multnat(m)[Succ[n]] = multnat(m)[n] + m$ **by** *blast*
qed (*auto simp*: *m*)

lemma *mult-Succ-nat* [*simp*]:
assumes $m \in Nat$ **and** $n \in Nat$
shows $m * Succ[n] = m * n + m$
using *assms* **by** (*simp add*: *nat-mult-def multnat-Succ*[*OF assms*])

lemma *mult-0-left-nat* [*simp*]:

assumes *n*: $n \in Nat$
shows $0 * n = 0$
using *n* **by** (*induct*, *simp-all*)

lemma *mult-Succ-left-nat* [*simp*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $Succ[m] * n = m * n + n$
using *n* **apply** *induct*
using *m* **by** (*simp-all add*: *add-ac-nat*)

Commutativity

lemma *mult-commute-nat* [*algebra-simps*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $m * n = n * m$
using *assms* **by** (*induct*, *simp-all*)

Distributivity

lemma *add-mult-distrib-left-nat* [*algebra-simps*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $m * (n + p) = m * n + m * p$
using *assms* **apply** *induct*
proof *auto*
fix *m*
assume $m \in Nat$ $m * (n + p) = m * n + m * p$
with *n p*
add-assoc-nat[*of* $m * n + m * p$ *n p*]
add-assoc-nat[*of* $m * n$ $m * p$ *n*]
add-commute-nat[*of* $m * p$ *n*]
add-assoc-nat[*of* $m * n$ *n* $m * p$]
add-assoc-nat[*of* $m * n + n$ $m * p$ *p*]
show $m * n + m * p + (n + p) = m * n + n + (m * p + p)$
by *simp*
qed

lemma *add-mult-distrib-right-nat* [*algebra-simps*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $(n + p) * m = n * m + p * m$
using *m* **apply** *induct*
using *n p* **apply** *auto*
proof −
fix *m*
assume $m \in Nat$ $(n + p) * m = n * m + p * m$
with *n p*
add-assoc-nat[*of* $n * m + p * m$ *n p*]
add-assoc-nat[*of* $n * m$ $p * m$ *n*]
add-commute-nat[*of* $p * m$ *n*]
add-assoc-nat[*of* $n * m$ *n* $p * m$]
add-assoc-nat[*of* $n * m + n$ $p * m$ *p*]
show $n * m + p * m + (n + p) = n * m + n + (p * m + p)$

by *simp*
qed

Identity element

lemma *mult-1-right-nat*: $a \in Nat \Longrightarrow a * 1 = a$ **by** *simp*
lemma *mult-1-left-nat*: $a \in Nat \Longrightarrow 1 * a = a$ **by** *simp*

Associativity

lemma *mult-assoc-nat*[*algebra-simps*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $m * (n * p) = (m * n) * p$
using *m* **apply** *induct*
using *assms* **by** (*auto simp add*: *add-mult-distrib-right-nat*)

Reasoning about $m * n = 0$, etc.

lemma *mult-is-0-nat* [*simp*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $(m * n = 0) = (m = 0 \lor n = 0)$
using *m* **apply** *induct*
using *n* **by** *auto*

lemma *mult-eq-1-iff-nat* [*simp*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $(m * n = 1) = (m = 1 \land n = 1)$
using *m* **apply** *induct*
using *n* **by** (*induct*, *auto*)+

lemma *one-eq-mult-iff-nat* [*simp*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $(1 = m * n) = (m = 1 \land n = 1)$
proof −
have $(1 = m * n) = (m * n = 1)$ **by** *auto*
also from *assms* **have** ... $= (m = 1 \land n = 1)$ **by** *simp*
finally show *?thesis* **.**
qed

Cancellation rules

lemma *mult-cancel1-nat* [*simp*]:
assumes *k*: $k \in Nat$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $(k * m = k * n) = (m = n \lor (k = 0))$
proof −
have $k \neq 0 \Longrightarrow k * m = k * n \Longrightarrow m = n$
using *n m* **proof** (*induct arbitrary*: *m*)
fix *m*
assume $k \neq 0$ $k * m = k * 0$ $m \in Nat$
with *k* **show** $m = 0$ **by** *simp*
next
fix *n m*

assume
n': $n \in Nat$ **and** $h1$:$\bigwedge m.$ $[\![k \neq 0;\ k * m = k * n;\ m \in Nat]\!] \Longrightarrow m = n$
and $k0$: $k \neq 0$ **and** $h2$: $k * m = k * Succ[n]$ **and** m':$m \in Nat$
from m' **show** $m = Succ[n]$
proof (*rule natCases*)
assume $m = 0$
with k **have** $k * m = 0$ **by** *simp*
moreover
from k $k0$ n' **have** $k * Succ[n] \neq 0$ **by** *simp*
moreover
note $h2$
ultimately show *?thesis* **by** *simp*
next
fix w
assume w:$w \in Nat$ **and** $h3$:$m = Succ[w]$
with k n' $h2$ **have** $k * w = k * n$ **by** *simp*
with k $k0$ w $h1[of\ w]$ $h3$ **show** *?thesis* **by** *simp*
qed
qed
with k m n **show** *?thesis* **by** *auto*
qed

lemma *mult-cancel2-nat* [*simp*]:
assumes m: $m \in Nat$ **and** n: $n \in Nat$ **and** k: $k \in Nat$
shows $(m * k = n * k) = (m = n \lor k = 0)$
using *assms* **by** (*simp add*: *mult-commute-nat*)

lemma *Suc-mult-cancel1-nat*:
assumes k: $k \in Nat$ **and** m: $m \in Nat$ **and** n: $n \in Nat$
shows $(Succ[k] * m = Succ[k] * n) = (m = n)$
using k m n *mult-cancel1-nat*[*of Succ*[k] m n] **by** *simp*

lemma *mult-left-commute-nat*[*algebra-simps*]:
assumes a: $a \in Nat$ **and** b: $b \in Nat$ **and** c: $c \in Nat$
shows $a * (b * c) = b * (a * c)$
using *assms* **by**(*simp only*: *mult-commute-nat mult-assoc-nat*)

theorems *mult-ac-nat* = *mult-assoc-nat mult-commute-nat mult-left-commute-nat*

7.3 Predecessor

definition *Pred*
where $Pred \equiv [n \in Nat \mapsto IF\ n{=}0\ THEN\ 0\ ELSE\ CHOOSE\ x \in Nat : n{=}Succ[x]]$

lemma *Pred-0-nat* [*simp*]: $Pred[0] = 0$
by (*simp add*: *Pred-def*)

lemma *Pred-Succ-nat* [*simp*]: $n \in Nat \Longrightarrow Pred[Succ[n]] = n$
unfolding *Pred-def* **by** (*auto intro*: *bChooseI2*)

lemma *Succ-Pred-nat* [*simp*]:
  assumes $n \in Nat$ **and** $n \neq 0$
  shows $Succ[Pred[n]] = n$
using *assms* **unfolding** *Pred-def* **by** (*cases n*, *auto intro*: *bChooseI2*)

lemma *Pred-in-nat* [*intro*!, *simp*]:
  assumes $n \in Nat$ **shows** $Pred[n] \in Nat$
using *assms* **by** (*cases n*, *auto*)

7.4 Difference of natural numbers

We define a form of difference $--$ of natural numbers that cuts off at 0, that is $m -- n = 0$ if $m < n$. This is sometimes called "arithmetic difference".

definition *adiffnat*
where $adiffnat(m) \equiv CHOOSE\ g \in [Nat \to Nat] : g[0] = m \wedge (\forall x \in Nat : g[Succ[x]] = Pred[g[x]])$

definition *adiff*   (**infixl** $--$ *65*)
where *nat-adiff-def*: $[\![m \in Nat; n \in Nat]\!] \Longrightarrow (m -- n) \equiv adiffnat(m)[n]$

Closure

lemma *adiffnatType*:
  assumes $m \in Nat$ **shows** $adiffnat(m) \in [Nat \to Nat]$
using *assms* **unfolding** *adiffnat-def* **by** (*rule bprimrecType-nat*, *auto*)

lemma *adiffIsNat* [*intro*!, *simp*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **shows** $m -- n \in Nat$
unfolding *nat-adiff-def*[*OF assms*] **using** *assms adiffnatType* **by** *blast*

Neutral element and Inductive step

lemma *adiffnat-0*:
  assumes $m \in Nat$ **shows** $adiffnat(m)[0] = m$
using *assms* **unfolding** *adiffnat-def* **by** (*rule primrec-natE*, *auto*)

lemma *adiff-0-nat* [*simp*]:
  assumes $m \in Nat$ **shows** $m -- 0 = m$
by (*simp add*: *nat-adiff-def*[*OF assms*] *adiffnat-0*[*OF assms*])

lemma *adiffnat-Succ*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $adiffnat(m)[Succ[n]] = Pred[adiffnat(m)[n]]$
proof (*rule primrec-natE*[*OF m*])
  show $adiffnat(m) = (CHOOSE\ g \in [Nat \to Nat] : g[0] = m \wedge (\forall x \in Nat : g[Succ[x]] = Pred[g[x]]))$
    unfolding *adiffnat-def* **..**

next
  assume $\forall\, n \in Nat : adiffnat(m)[Succ[n]] = Pred[adiffnat(m)[n]]$
  with n **show** $adiffnat(m)[Succ[n]] = Pred[adiffnat(m)[n]]$ **by** *blast*
qed (*auto*)

lemma *adiff-Succ-nat*:
  assumes $m \in Nat$ **and** $n \in Nat$
  shows $m -- Succ[n] = Pred[m -- n]$
using *assms* **by** (*simp add*: *nat-adiff-def adiffnat-Succ*[*OF assms*])

lemma *adiff-0-eq-0-nat* [*simp*]:
  assumes n: $n \in Nat$
  shows $0 -- n = 0$
using n **apply** *induct* **by** (*simp-all add*: *adiff-Succ-nat*)

Reasoning about $m -- m = 0$, etc.

lemma *adiff-Succ-Succ-nat* [*simp*]:
  assumes m: $m \in Nat$ **and** n: $n \in Nat$
  shows $Succ[m] -- Succ[n] = m -- n$
using n **apply** *induct*
using *assms* **by** (*auto simp add*: *adiff-Succ-nat*)

lemma *adiff-self-eq-0-nat* [*simp*]:
  assumes m: $m \in Nat$
  shows $m -- m = 0$
using m **apply** *induct* **by** *auto*

Associativity

lemma *adiff-adiff-left-nat*:
  assumes i: $i \in Nat$ **and** j: $j \in Nat$ **and** k: $k \in Nat$
  shows $(i -- j) -- k = i -- (j + k)$
using i j **apply** (*rule diffInduct*)
using k **by** *auto*

lemma *Succ-adiff-adiff-nat* [*simp*]:
  assumes m: $m \in Nat$ **and** n: $n \in Nat$ **and** k: $k \in Nat$
  shows $(Succ[m] -- n) -- Succ[k] = (m -- n) -- k$
using *assms* **by** (*simp add*: *adiff-adiff-left-nat*)

Commutativity

lemma *adiff-commute-nat*:
  assumes i: $i \in Nat$ **and** j: $j \in Nat$ **and** k: $k \in Nat$
  shows $i -- j -- k = i -- k -- j$
using *assms* **by** (*simp add*: *adiff-adiff-left-nat add-commute-nat*)

Cancellation rules

lemma *adiff-add-inverse-nat* [*simp*]:
  assumes m: $m \in Nat$ **and** n: $n \in Nat$

shows $(n + m) -- n = m$
using *n* **apply** *induct*
using *assms* **by** *auto*

lemma *adiff-add-inverse2-nat* [*simp*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $(m + n) -- n = m$
using *assms* **by** (*simp add*: *add-commute-nat* [*of m n*])

lemma *adiff-cancel-nat* [*simp*]:
assumes *k*: $k \in Nat$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $(k + m) -- (k + n) = m -- n$
using *k* **apply** *induct*
using *assms* **by** *simp-all*

lemma *adiff-cancel2-nat* [*simp*]:
assumes *k*: $k \in Nat$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $(m + k) -- (n + k) = m -- n$
using *assms* **by** (*simp add*: *add-commute-nat*)

lemma *adiff-add-0-nat* [*simp*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $n -- (n + m) = 0$
using *n* **apply** *induct*
using *assms* **by** *simp-all*

Difference distributes over multiplication

lemma *adiff-mult-distrib-nat* [*algebra-simps*]:
assumes *k*: $k \in Nat$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $(m -- n) * k = (m * k) -- (n * k)$
using *m n* **apply**(*rule diffInduct*)
using *k* **by** *simp-all*

lemma *adiff-mult-distrib2-nat* [*algebra-simps*]:
assumes *k*: $k \in Nat$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $k * (m -- n) = (k * m) -- (k * n)$
using *assms* **by** (*simp add*: *adiff-mult-distrib-nat mult-commute-nat* [*of k*])

— NOT added as rewrites, since sometimes they are used from right-to-left
lemmas *nat-distrib* =
add-mult-distrib-right-nat add-mult-distrib-left-nat
adiff-mult-distrib-nat adiff-mult-distrib2-nat

7.5 Additional arithmetic theorems

7.5.1 Monotonicity of Addition

lemma *Succ-pred* [*simp*]:
assumes *n*: $n \in Nat$
shows $n > 0 \Longrightarrow Succ[n -- 1] = n$

using *assms* **by** (*simp add*: *adiff-Succ-nat*[*OF n zeroIsNat*] *nat-gt0-not0*[*OF n*])

lemma *nat-add-left-cancel-leq* [*simp*]:
  assumes *k*: $k \in Nat$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(k + m \leq k + n) = (m \leq n)$
using *assms* **by** (*induct k*) *simp-all*

lemma *nat-add-left-cancel-less* [*simp*]:
  assumes *k*: $k \in Nat$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(k + m < k + n) = (m < n)$
using *k* **apply** *induct*
using *assms* **by** *simp-all*

lemma *nat-add-right-cancel-less* [*simp*]:
  assumes *k*: $k \in Nat$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(m + k < n + k) = (m < n)$
using *k* **apply** *induct*
using *assms* **by** *simp-all*

lemma *nat-add-right-cancel-leq* [*simp*]:
  assumes *k*: $k \in Nat$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(m + k \leq n + k) = (m \leq n)$
using *k* **apply** *induct*
using *assms* **by** *simp-all*

lemma *add-gr-0* [*simp*]:
  assumes $n \in Nat$ **and** $m \in Nat$
  shows $(m + n > 0) = (m > 0 \lor n > 0)$
using *assms* **by** (*auto dest*: *nat-gt0-implies-Succ nat-not-lessD*)

lemma *less-imp-Succ-add*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $m < n \Longrightarrow (\exists\, k \in Nat: n = Succ[m + k])$ (**is** - $\Longrightarrow$ *?P*(*n*))
using *n* **proof** (*induct*)
  case *0* **with** *m* **show** *?case* **by** *simp*
next
  fix *n*
  assume *n*: $n \in Nat$ **and** *ih*: $m<n \Longrightarrow$ *?P*(*n*) **and** *suc*: $m < Succ[n]$
  from *suc m n* **show** *?P*(*Succ*[*n*])
  proof (*rule nat-less-SuccE*)
    assume $m<n$
    then obtain *k* **where** $k \in Nat$ **and** $n = Succ[m + k]$ **by** (*blast dest*: *ih*)
    with *m n* **have** $Succ[k] \in Nat$ **and** $Succ[n] = Succ[m + Succ[k]]$ **by** *auto*
    thus *?thesis* **..**
  next
    assume $m=n$
    with *n* **have** $Succ[n] = Succ[m + 0]$ **by** *simp*
    thus *?thesis* **by** *blast*
  qed

qed

lemma *nat-leq-trans-add-left-false* [*simp*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $[\![m + n \leq p;\ p \leq n]\!] \Longrightarrow (m + n < p) = FALSE$
apply (*induct n p rule*: *diffInduct*)
using *assms* **by** *simp-all*

7.5.2 (Partially) Ordered Groups

— The two following lemmas are just "one half" of *nat-add-left-cancel-leq* and *nat-add-right-cancel-leq* proved above.
lemma *add-leq-left-mono*:
assumes *a*: $a \in Nat$ **and** *b*: $b \in Nat$ **and** *c*: $c \in Nat$
shows $a \leq b \Longrightarrow c + a \leq c + b$
using *assms* **by** *simp*

lemma *add-leq-right-mono*:
assumes *a*: $a \in Nat$ **and** *b*: $b \in Nat$ **and** *c*: $c \in Nat$
shows $a \leq b \Longrightarrow a + c \leq b + c$
using *assms* **by** *simp*

non-strict, in both arguments

lemma *add-leq-mono*:
assumes *a*: $a \in Nat$ **and** *b*: $b \in Nat$ **and** *c*: $c \in Nat$ **and** *d*: $d \in Nat$
shows $a \leq b \Longrightarrow c \leq d \Longrightarrow a + c \leq b + d$
using *assms*
add-leq-right-mono[*OF a b c*]
add-leq-left-mono[*OF c d b*]
nat-leq-trans[*of* $a + c$ $b + c$ $b + d$]
by *simp*

— Similar for strict less than.
lemma *add-less-left-mono*:
assumes *a*: $a \in Nat$ **and** *b*: $b \in Nat$ **and** *c*: $c \in Nat$
shows $a < b \Longrightarrow c + a < c + b$
using *assms* **by** *simp*

lemma *add-less-right-mono*:
assumes *a*: $a \in Nat$ **and** *b*: $b \in Nat$ **and** *c*: $c \in Nat$
shows $a < b \Longrightarrow a + c < b + c$
using *assms* **by** *simp*

Strict monotonicity in both arguments

lemma *add-less-mono*:
assumes *a*: $a \in Nat$ **and** *b*: $b \in Nat$ **and** *c*: $c \in Nat$ **and** *d*: $d \in Nat$
shows $a < b \Longrightarrow c < d \Longrightarrow a + c < b + d$
using *assms*
add-less-right-mono[*OF a b c*]

add-less-left-mono[*OF c d b*]
nat-less-trans[*of a* + *c b* + *c b* + *d*]
by *simp*

lemma *add-less-leq-mono*:
assumes *a*: $a \in Nat$ **and** *b*: $b \in Nat$ **and** *c*: $c \in Nat$ **and** *d*: $d \in Nat$
shows $a < b \Longrightarrow c \leq d \Longrightarrow a + c < b + d$
using *assms*
add-less-right-mono[*OF a b c*]
add-leq-left-mono[*OF c d b*]
nat-less-leq-trans[*of a* + *c b* + *c b* + *d*]
by *blast*

lemma *add-leq-less-mono*:
assumes *a*: $a \in Nat$ **and** *b*: $b \in Nat$ **and** *c*: $c \in Nat$ **and** *d*: $d \in Nat$
shows $a \leq b \Longrightarrow c < d \Longrightarrow a + c < b + d$
using *assms*
add-leq-right-mono[*OF a b c*]
add-less-left-mono[*OF c d b*]
nat-leq-less-trans[*of a* + *c b* + *c b* + *d*]
by *blast*

lemma *leq-add1* [*simp*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $n \leq n + m$
using *assms add-leq-left-mono*[*of 0 m n*] **by** *simp*

lemma *leq-add2* [*simp*]:
assumes $m \in Nat$ **and** $n \in Nat$
shows $n \leq m + n$
using *assms add-leq-right-mono* [*of 0 m n*] **by** *simp*

lemma *less-add-Succ1*:
assumes $i \in Nat$ **and** $m \in Nat$
shows $i < Succ[i + m]$
using *assms* **by** *simp*

lemma *less-add-Succ2*:
assumes $i \in Nat$ **and** $m \in Nat$
shows $i < Succ[m + i]$
using *assms* **by** *simp*

lemma *less-iff-Succ-add*:
assumes $m \in Nat$ **and** $n \in Nat$
shows $(m < n) = (\exists\, k \in Nat\colon n = Succ[m + k])$
using *assms* **by** (*auto intro*!: *less-imp-Succ-add*)

lemma *trans-leq-add1*:
assumes $i \leq j$ **and** $i \in Nat$ **and** $j \in Nat$ **and** $m \in Nat$

shows $i \leq j + m$
using *assms* **by** (*auto elim*: *nat-leq-trans*)

lemma *trans-leq-add2*:
assumes $i \leq j$ **and** $i \in Nat$ **and** $j \in Nat$ **and** $m \in Nat$
shows $i \leq m + j$
using *assms* **by** (*auto elim*: *nat-leq-trans*)

lemma *trans-less-add1*:
assumes $i < j$ **and** $i \in Nat$ **and** $j \in Nat$ **and** $m \in Nat$
shows $i < j + m$
using *assms* **by** (*auto elim*: *nat-less-leq-trans*)

lemma *trans-less-add2*:
assumes $i < j$ **and** $i \in Nat$ **and** $j \in Nat$ **and** $m \in Nat$
shows $i < m + j$
using *assms* **by** (*auto elim*: *nat-less-leq-trans*)

lemma *add-lessD1*:
assumes $i+j < k$ **and** $i \in Nat$ **and** $j \in Nat$ **and** $k \in Nat$
shows $i < k$
using *assms* **by** (*intro nat-leq-less-trans*[*of i i+j k*], *simp+*)

lemma *not-add-less1* [*simp*]:
assumes *i*: $i \in Nat$ **and** *j*: $j \in Nat$
shows $(i + j < i) = FALSE$
by (*auto dest*: *add-lessD1*[*OF - i j i*])

lemma *not-add-less2* [*simp*]:
assumes $i \in Nat$ **and** $j \in Nat$
shows $(j + i < i) = FALSE$
using *assms* **by** (*simp add*: *add-commute-nat*)

lemma *add-leqD1*:
assumes $m + k \leq n$ **and** $k \in Nat$ **and** $m \in Nat$ **and** $n \in Nat$
shows $m \leq n$
using *assms* **by** (*intro nat-leq-trans*[*of m m+k n*], *simp+*)

lemma *add-leqD2*:
assumes $m+k \leq n$ **and** $k \in Nat$ **and** $m \in Nat$ **and** $n \in Nat$
shows $k \leq n$
using *assms* **by** (*intro nat-leq-trans*[*of k m+k n*], *simp+*)

lemma *add-leqE*:
assumes $m+k \leq n$ **and** $k \in Nat$ **and** $m \in Nat$ **and** $n \in Nat$
shows $(m \leq n \Longrightarrow k \leq n \Longrightarrow R) \Longrightarrow R$
using *assms* **by** (*blast dest*: *add-leqD1 add-leqD2*)

lemma *leq-add-left-false* [*simp*]:

assumes m: $m \in Nat$ **and** n: $n \in Nat$ **and** $n \neq 0$
shows $m + n \leq m = FALSE$
using *assms nat-leq-less*[*of* $m + n$ m] *add-eq-self-zero-nat*[*OF* m n] **by** *auto*

7.5.3 More results about arithmetic difference

Addition is the inverse of subtraction: if $n \leq m$ then $n + (m -- n) = m$.

lemma *add-adiff-inverse*:
assumes $m \in Nat$ **and** $n \in Nat$
shows $\neg(m < n) \Longrightarrow n + (m -- n) = m$
apply (*induct m n rule*: *diffInduct*)
using *assms* **by** *simp-all*

lemma *le-add-adiff-inverse* [*simp*]:
assumes $m \in Nat$ **and** $n \in Nat$
shows $n \leq m \Longrightarrow n + (m -- n) = m$
using *assms* **by** (*simp add*: *add-adiff-inverse nat-not-order-simps*)

lemma *le-add-adiff-inverse2* [*simp*]:
assumes $m \in Nat$ **and** $n \in Nat$
shows $n \leq m \Longrightarrow (m -- n) + n = m$
using *assms* **by** (*simp add*: *add-commute-nat*)

lemma *Succ-adiff-leq*:
assumes $m \in Nat$ **and** $n \in Nat$
shows $n \leq m \Longrightarrow Succ[m] -- n = Succ[m -- n]$
apply (*induct m n rule*: *diffInduct*)
using *assms* **by** *simp-all*

lemma *adiff-less-Succ*:
assumes m: $m \in Nat$ **and** n: $n \in Nat$
shows $m -- n < Succ[m]$
apply (*induct m n rule*: *diffInduct*)
using *assms* **by** (*auto simp*: *nat-less-Succ*)

lemma *adiff-leq-self* [*simp*]:
assumes m: $m \in Nat$ **and** n: $n \in Nat$
shows $m -- n \leq m$
apply (*induct m n rule*: *diffInduct*)
using *assms* **by** *simp-all*

lemma *leq-iff-add*:
assumes m: $m \in Nat$ **and** n: $n \in Nat$
shows $m \leq n = (\exists\, k \in Nat: n = m + k)$ (**is** *?lhs* = *?rhs*)
proof −
have *1*: *?lhs* ⇒ *?rhs*
proof
assume *mn*: $m \leq n$
with m n **have** $n = m + (n -- m)$ **by** *simp*

with *m n* **show** *?rhs* **by** *blast*
qed
from *assms* **have** *2*: *?rhs* ⇒ *?lhs* **by** *auto*
from *1 2 assms* **show** *?thesis* **by** *blast*
qed

lemma *less-imp-adiff-less*:
assumes *jk*: $j < k$ **and** *j*: $j \in Nat$ **and** *k*: $k \in Nat$ **and** *n*: $n \in Nat$
shows $j -- n < k$
proof −
from *j n* **have** $j -- n \in Nat$ $j -- n \leq j$ **by** *simp+*
with *j k jk* **show** *?thesis* **by** (*auto elim*: *nat-leq-less-trans*)
qed

lemma *adiff-Succ-less* :
assumes *i*: $i \in Nat$ **and** *n*: $n \in Nat$
shows $0 < n \Longrightarrow n -- Succ[i] < n$
using *n* **apply** *cases*
using *i* **by** *auto*

lemma *adiff-add-assoc*:
assumes $k \leq j$ **and** $i \in Nat$ **and** $j \in Nat$ **and** $k \in Nat$
shows $(i + j) -- k = i + (j -- k)$
using *assms* **by** (*induct j k rule*: *diffInduct*, *simp+*)

lemma *adiff-add-assoc2*:
assumes $k \leq j$ **and** $i \in Nat$ **and** $j \in Nat$ **and** $k \in Nat$
shows $(j + i) -- k = (j -- k) + i$
using *assms* **by** (*simp add*: *add-commute-nat adiff-add-assoc*)

lemma *adiff-add-assoc3*:
assumes $n \leq p$ **and** $p \leq m+n$ **and** $m \in Nat$ **and** $n \in Nat$ **and** $p \in Nat$
shows $m -- (p -- n) = m + n -- p$
using *assms* **by** (*induct p n rule*: *diffInduct*, *simp+*)

lemma *adiff-add-assoc4*:
assumes *1*: $n \leq m$ **and** *2*: $m -- n \leq p$ **and** *3*: $m \leq p$
and *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $p -- (m -- n) = (p -- m) + n$
using *assms*
adiff-add-assoc2[*OF - n p m*, *symmetric*]
adiff-add-assoc3[*OF - - p n m*] **apply** *simp*
using *trans-leq-add1*[*OF - m p n*] **by** *simp*

lemma *le-imp-adiff-is-add*:
assumes $i \leq j$ **and** $i \in Nat$ **and** $j \in Nat$ **and** $k \in Nat$
shows $(j -- i = k) = (j = k + i)$
using *assms* **by** *auto*

lemma *adiff-is-0-eq* [*simp*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $(m -- n = 0) = (m \leq n)$
by (*induct m n rule*: *diffInduct*, *simp-all add*: *assms*)

lemma *adiff-is-0-eq′* :
assumes $m \leq n$ **and** $m \in Nat$ **and** $n \in Nat$
shows $m -- n = 0$
using *assms* **by** *simp*

lemma *zero-less-adiff* [*simp*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $(0 < n -- m) = (m < n)$
by (*induct m n rule*: *diffInduct*, *simp-all add*: *assms*)

lemma *less-imp-add-positive*:
assumes $i < j$ **and** *i*: $i \in Nat$ **and** *j*: $j \in Nat$
shows $\exists k \in Nat$: $0 < k \wedge i + k = j$
proof −
from *assms* **have** $i \leq j$ **by** *auto*
with *i j* **have** $i + (j -- i) = j$ **by** *simp*
moreover
from *assms* **have** $j -- i \in Nat$ $0 < j -- i$ **by** *simp+*
ultimately
show *?thesis* **by** *blast*
qed

lemma *leq-adiff-right-add-left*:
assumes $k \leq n$ **and** $m \in Nat$ **and** $n \in Nat$ **and** $k \in Nat$
shows $m \leq n -- k = (m + k \leq n)$
using *assms* **by** (*induct n k rule*: *diffInduct*, *simp+*)

lemma *leq-adiff-left-add-right*:
assumes *1*: $n -- p \leq m$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $n \leq m + p$
using *assms* **by** (*induct n p rule*: *diffInduct*, *simp+*)

lemma *leq-adiff-trans*:
assumes $p \leq m$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $p -- n \leq m$
apply(*rule nat-leq-trans*[*of p -- n p m*])
using *assms adiff-leq-self*[*OF p n*] **by** *simp-all*

lemma *leq-adiff-right-false* [*simp*]:
assumes $n \neq 0$ $n \leq m$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$
shows $m \leq m -- n = FALSE$
using *assms* **by** (*simp add*: *leq-adiff-right-add-left*[*OF - m m n*])

lemma *leq-adiff-right-imp-0*:

assumes *h*:$n \leq n$ −− *p* $p \leq n$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $p = 0$
using *p h* **apply** (*induct*)
using *n* **by** *auto*

7.5.4 Monotonicity of Multiplication

lemma *mult-leq-left-mono*:
assumes *1*:$a \leq b$ **and** *a*: $a \in Nat$ **and** *b*: $b \in Nat$ **and** *c*: $c \in Nat$
shows $c * a \leq c * b$
using *c* **apply** *induct*
using *1 a b* **by** (*simp-all add*: *add-leq-mono*)

lemma *mult-leq-right-mono*:
assumes *1*:$a \leq b$ **and** *a*: $a \in Nat$ **and** *b*: $b \in Nat$ **and** *c*: $c \in Nat$
shows $a * c \leq b * c$
using *c* **apply** *induct*
using *1 a b* **by** (*simp-all add*: *add-leq-mono add-commute-nat*)

$\leq$ monotonicity, BOTH arguments

lemma *mult-leq-mono*:
assumes *1*: $i \leq j$ $k \leq l$
and *i*: $i \in Nat$ **and** *j*: $j \in Nat$ **and** *k*: $k \in Nat$ **and** *l*: $l \in Nat$
shows $i * k \leq j * l$
using *assms*
mult-leq-right-mono[*OF - i j k*]
mult-leq-left-mono[*OF - k l j*]
nat-leq-trans[*of* $i * k$ $j * k$ $j * l$]
by *simp*

strict, in 1st argument

lemma *mult-less-left-mono*:
assumes *1*: $i < j$ $0 < k$ **and** *i*: $i \in Nat$ **and** *j*: $j \in Nat$ **and** *k*: $k \in Nat$
shows $k * i < k * j$
using *1*
proof (*auto simp*: *nat-gt0-iff-Succ*[*OF k*])
fix *m*
assume *m*: $m \in Nat$ **and** $i < j$
with *m i j* **show** $Succ[m] * i < Succ[m] * j$
by (*induct m*, *simp-all add*: *add-less-mono*)
qed

lemma *mult-less-right-mono*:
assumes *1*: $i < j$ $0 < k$ **and** *i*: $i \in Nat$ **and** *j*: $j \in Nat$ **and** *k*: $k \in Nat$
shows $i * k < j * k$
using *1*
proof (*auto simp*: *nat-gt0-iff-Succ*[*OF k*])
fix *m*
assume *m*: $m \in Nat$ **and** $i < j$

with *m i j* **show** $i * Succ[m] < j * Succ[m]$
  by (*induct m*, *simp-all add*: *add-less-mono*)
qed

lemma *nat-0-less-mult-iff* [*simp*]:
  assumes *i*: $i \in Nat$ **and** *j*: $j \in Nat$
  shows $(0 < i * j) = (0 < i \wedge 0 < j)$
using *i* **apply** *induct*
  using *j* **apply** *simp*
  using *j* **apply**(*induct*, *simp-all*)
done

lemma *one-leq-mult-iff* :
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(1 \leq m * n) = (1 \leq m \wedge 1 \leq n)$
using *assms* **by** *simp*

lemma *mult-less-cancel-left* [*simp*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *k*: $k \in Nat$
  shows $(k * m < k * n) = (0 < k \wedge m < n)$
proof (*auto intro*!: *mult-less-left-mono*[*OF* - - *m n k*])
  assume $k*m < k*n$
  from *k m n this* **show** $0 < k$ **by** (*cases k*, *simp-all*)
next
  assume *1*: $k*m < k*n$
  show $m < n$
  proof (*rule contradiction*)
    assume $\neg(m < n)$
    with *m n k* **have** $k*n \leq k*m$ **by** (*simp add*: *nat-not-order-simps mult-leq-left-mono*)
    with *m n k* **have** $\neg(k*m < k*n)$ **by** (*simp add*: *nat-not-order-simps*)
    with *1* **show** *FALSE* **by** *simp*
  qed
qed

lemma *mult-less-cancel-right* [*simp*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *k*: $k \in Nat$
  shows $(m * k < n * k) = (0 < k \wedge m < n)$
proof (*auto intro*!: *mult-less-right-mono*[*OF* - - *m n k*])
  assume $m*k < n*k$
  from *k m n this* **show** $0 < k$ **by** (*cases k*, *simp-all*)
next
  assume *1*: $m*k < n*k$
  show $m < n$
  proof (*rule contradiction*)
    assume $\neg(m < n)$
    with *m n k* **have** $n*k \leq m*k$ **by** (*simp add*: *nat-not-order-simps mult-leq-right-mono*)
    with *m n k* **have** $\neg(m*k < n*k)$ **by** (*simp add*: *nat-not-order-simps*)
    with *1* **show** *FALSE* **by** *simp*
  qed

qed

lemma *mult-less-self-left* [*dest*]:
  assumes *less*: $n*k < n$ **and** *n*: $n \in Nat$ **and** *k*: $k \in Nat$
  shows $k=0$
using *k assms* **by** (*cases*, *auto*)

lemma *mult-less-self-right* [*dest*]:
  assumes *less*: $k*n < n$ **and** *n*: $n \in Nat$ **and** *k*: $k \in Nat$
  shows $k=0$
using *k assms* **by** (*cases*, *auto*)

lemma *mult-leq-cancel-left* [*simp*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *k*: $k \in Nat$
  shows $(k * m \leq k * n) = (k = 0 \vee m \leq n)$
using *assms* **proof** (*auto simp*: *mult-leq-left-mono nat-neq0-conv*[*simplified*])
  assume *1*: $k*m \leq k*n$ **and** *2*: $0 < k$
  show $m \leq n$
  proof (*rule contradiction*)
    assume $\neg(m \leq n)$
   with *2 m n k* **have** $k*n < k*m$ **by** (*simp add*: *nat-not-order-simps mult-less-left-mono*)
    with *m n k* **have** $\neg(k*m \leq k*n)$ **by** (*simp add*: *nat-not-order-simps*)
    with *1* **show** *FALSE* **by** *simp*
  qed
qed

lemma *mult-leq-cancel-right* [*simp*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *k*: $k \in Nat$
  shows $(m * k \leq n * k) = (k = 0 \vee m \leq n)$
using *assms* **proof** (*auto simp*: *mult-leq-right-mono nat-neq0-conv*[*simplified*])
  assume *1*: $m*k \leq n*k$ **and** *2*: $0 < k$
  show $m \leq n$
  proof (*rule contradiction*)
    assume $\neg(m \leq n)$
   with *2 m n k* **have** $n*k < m*k$ **by** (*simp add*: *nat-not-order-simps mult-less-right-mono*)
    with *m n k* **have** $\neg(m*k \leq n*k)$ **by** (*simp add*: *nat-not-order-simps*)
    with *1* **show** *FALSE* **by** *simp*
  qed
qed

lemma *Suc-mult-less-cancel1*:
  assumes $m \in Nat$ **and** $n \in Nat$ **and** $k \in Nat$
  shows $(Succ[k] * m < Succ[k] * n) = (m < n)$
using *assms* **by** (*simp del*: *mult-Succ-left-nat*)

lemma *Suc-mult-leq-cancel1*:
  assumes $m \in Nat$ **and** $n \in Nat$ **and** $k \in Nat$
  shows $(Succ[k] * m \leq Succ[k] * n) = (m \leq n)$
using *assms* **by** (*simp del*: *mult-Succ-left-nat*)

lemma *nat-leq-square*:
  assumes *m*: $m \in Nat$
  shows $m \leq m * m$
using *m* **by** (*cases*, *auto*)

lemma *nat-leq-cube*:
  assumes *m*: $m \in Nat$
  shows $m \leq m * (m * m)$
using *m* **by** (*cases*, *auto*)

Lemma for *gcd*

lemma *mult-eq-self-implies-10*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(m * n = m) = (n = 1 \vee m = 0)$ (**is** *?lhs* = *?rhs*)
proof –
  from *assms* **have** $(m*n = m) = (m*n = m*1)$ **by** *simp*
  also have ... = *?rhs* **by** (*rule mult-cancel1-nat*[*OF m n oneIsNat*])
  finally show *?thesis* **.**
qed

end

8 Tuples and Relations in TLA$^+$

theory *Tuples*
imports *NatOrderings*
begin

We develop a theory of tuples and relations in TLA$^+$. Tuples are functions whose domains are intervals of the form $1..n$, for some natural number n, and relations are sets of tuples. In particular, TLA$^+$ distinguishes between a function and its graph, and we have functions to convert between the two. (This is useful, for example, when defining functions recursively, as we have a fixed point theorem on sets but not on functions.) We also introduce standard notions for binary relations, such as orderings, equivalence relations and so on.

8.1 Sequences and Tuples

Tuples and sequences are the same mathematical objects in TLA$^+$, so we give elementary definitions for sequences here. Further operations on sequences require arithmetic and will be introduced in a separate theory.

definition *Seq* — set of finite sequences with elements from *S*

where *Seq*(*S*) ≡ *UNION* { [*1* .. *n* → *S*] : *n* ∈ *Nat* }

definition *isASeq* — characteristic predicate for sequences or tuples
where *isASeq*(*s*) ≡ *isAFcn*(*s*) ∧ (∃ *n* ∈ *Nat* : *DOMAIN s* = *1* .. *n*)

definition *Len* — length of a sequence
where *Len*(*s*) ≡ *CHOOSE n* ∈ *Nat* : *DOMAIN s* = *1* .. *n*

lemma *isASeqIsBool* [*intro*!,*simp*]:
  isBool(*isASeq*(*s*))
by (*simp add*: *isASeq-def*)

lemma *boolifyIsASeq* [*simp*]:
  boolify(*isASeq*(*s*)) = *isASeq*(*s*)
by *auto*

lemma *isASeqI* [*intro*]:
  assumes *isAFcn*(*s*) **and** *n* ∈ *Nat* **and** *DOMAIN s* = *1* .. *n*
  shows *isASeq*(*s*)
using *assms* **by** (*auto simp*: *isASeq-def*)

lemma *SeqIsASeq* [*elim*!]:
  assumes *s* ∈ *Seq*(*S*)
  shows *isASeq*(*s*)
using *assms* **by** (*auto simp*: *Seq-def*)

lemma *LenI* [*intro*]:
  assumes *DOMAIN s* = *1* .. *n* **and** *n* ∈ *Nat*
  shows *Len*(*s*) = *n*
proof (*unfold Len-def*, *rule bChooseI2*)
  from *assms* **show** ∃ *x* ∈ *Nat* : *DOMAIN s* = *1* .. *x* **by** *blast*
next
  fix *m*
  assume *m* ∈ *Nat* **and** *DOMAIN s* = *1* .. *m*
  with *assms* **show** *m* = *n* **by** *auto*
qed

lemma *isASeqE* [*elim*]:
  assumes *isASeq*(*s*)
  and ⟦*isAFcn*(*s*); *DOMAIN s* = *1* .. *Len*(*s*); *Len*(*s*) ∈ *Nat*⟧ ⟹ *P*
  shows *P*
using *assms* **by** (*auto simp*: *isASeq-def dest*: *LenI*)

lemma *SeqIsAFcn* :
  assumes *isASeq*(*s*)
  shows *isAFcn*(*s*)
using *assms* **by** *auto*

— *s* ∈ *Seq*(*S*) ⟹ *isAFcn*(*s*)

lemmas $SeqIsAFcn'$ = $SeqIsASeq[THEN\ SeqIsAFcn,\ standard]$

lemma $LenInNat$ $[simp]$:
  assumes $isASeq(s)$
  shows $Len(s) \in Nat$
using $assms$ **by** $auto$

— $s \in Seq(S) \Longrightarrow Len(s) \in Nat$
lemmas $LenInNat'$ $[simp]$ = $SeqIsASeq[THEN\ LenInNat,\ standard]$

lemma $DomainSeqLen$ $[simp]$:
  assumes $isASeq(s)$
  shows $DOMAIN\ s = 1\ ..\ Len(s)$
using $assms$ **by** $auto$

— $s \in Seq(S) \Longrightarrow DOMAIN\ s = 1\ ..\ Len(s)$
lemmas $DomainSeqLen'$ = $SeqIsASeq[THEN\ DomainSeqLen,\ standard]$

lemma $seqEqualI$:
  assumes $isASeq(s)$ **and** $isASeq(t)$
    and $Len(s) = Len(t)$ **and** $\forall\, k \in 1\ ..\ Len(t) : s[k] = t[k]$
  shows $s = t$
using $assms$ **by** $(intro\ fcnEqual[of\ s\ t],\ auto)$

lemma $seqEqualE$:
  assumes $isASeq(s)$ **and** $isASeq(t)$ **and** $s{=}t$
    and $\llbracket\ Len(s) = Len(t);\ \forall\, k \in 1\ ..\ Len(t) : s[k] = t[k]\ \rrbracket \Longrightarrow P$
  shows P
using $assms$ **by** $auto$

lemma $seqEqualIff$:
  assumes $isASeq(s)$ **and** $isASeq(t)$
  shows $(s = t) = (Len(s) = Len(t) \wedge (\forall\, k \in 1\ ..\ Len(t) : s[k] = t[k]))$
by $(auto\ elim:\ seqEqualI[OF\ assms]\ seqEqualE[OF\ assms])$

lemma $SeqI$ $[intro!]$:
  assumes $isASeq(s)$ **and** $\bigwedge k.\ k \in 1\ ..\ Len(s) \Longrightarrow s[k] \in S$
  shows $s \in Seq(S)$
using $assms$ **by** $(auto\ simp:\ Seq\text{-}def)$

lemma $SeqI'$: — closer to the definition but probably less useful
  assumes $s \in [1\ ..\ n \to S]$ **and** $n \in Nat$
  shows $s \in Seq(S)$
using $assms$ **by** $(auto\ simp:\ Seq\text{-}def)$

lemma $SeqE$ $[elim]$:
  assumes s: $s \in Seq(S)$
  and p: $\llbracket s \in [1\ ..\ Len(s) \to S];\ Len(s) \in Nat \rrbracket \Longrightarrow P$
  shows P

proof (*rule p*)
  from *s* **show** *Len*(*s*) ∈ *Nat* **by** (*rule LenInNat'*)
next
  from *s* **obtain** *n* **where** *n* ∈ *Nat* **and** *s* ∈ [*1* .. *n* → *S*]
    by (*auto simp*: *Seq-def*)
  with *DomainSeqLen'*[*OF s*] **show** *s* ∈ [*1* .. *Len*(*s*) → *S*] **by** *auto*
qed

lemma *seqFuncSet*:
  assumes *s* ∈ *Seq*(*S*)
  shows *s* ∈ [*1* .. *Len*(*s*) → *S*]
using *assms* **by** *auto*

lemma *seqElt* [*elim*!]:
  assumes *s* ∈ *Seq*(*S*) **and** *n* ∈ *Nat* **and** *1* ≤ *n* **and** *n* ≤ *Len*(*s*)
  shows *s*[*n*] ∈ *S*
using *assms* **by** *auto*

lemma *seqInSeqRange*:
  assumes *isASeq*(*s*)
  shows *s* ∈ *Seq*(*Range*(*s*))
using *assms* **by** *auto*

lemma *isASeqInSeq*: *isASeq*(*s*) = (∃ *S*: *s* ∈ *Seq*(*S*))
by (*auto elim*: *seqInSeqRange*)

8.2 Sequences via *emptySeq* and *Append*

Sequences can be built from the constructors *emptySeq* (written ⟨⟩) and Append.

definition *emptySeq* ((<< >>))
where << >> ≡ [*x* ∈ *1* .. *0* ↦ {}]

notation (*xsymbols*)
  emptySeq   ((⟨⟩))

notation (*HTML* **output**)
  emptySeq   ((⟨⟩))

definition *Append* :: [*c*,*c*] ⇒ *c*
where *Append*(*s*,*e*) ≡ [*k* ∈ *1* .. *Succ*[*Len*(*s*)] ↦ *IF k* = *Succ*[*Len*(*s*)] *THEN e ELSE s*[*k*]]

lemma *emptySeqIsASeq* [*simp*,*intro*!]: *isASeq*(⟨⟩)
by (*auto simp*: *emptySeq-def isASeq-def*)

— *isAFcn*(⟨⟩)
lemmas *emptySeqIsAFcn* [*simp*,*intro*!] = *emptySeqIsASeq*[*THEN SeqIsAFcn*]

lemma *lenEmptySeq* [*simp*]: $Len(\langle\rangle) = 0$
by (*auto simp*: *emptySeq-def*)

lemma *emptySeqInSeq* : $\langle\rangle \in Seq(S)$
by *auto*

lemma *SeqNotEmpty* [*simp*]:
$(Seq(S) = \{\}) = FALSE$
$(\{\} = Seq(S)) = FALSE$
by *auto*

lemma *appendIsASeq* [*simp*,*intro*!]:
assumes *s*: $isASeq(s)$
shows $isASeq(Append(s,e))$
using *s* **unfolding** *Append-def*
by (*rule isASeqE*, *intro isASeqI*, *auto simp del*: *natIntervalSucc*)

— $isASeq(s) \Longrightarrow isAFcn(Append(s,e))$
lemmas *appendIsAFcn* [*simp*,*intro*!] = *appendIsASeq*[*THEN SeqIsAFcn*, *standard*]

lemma *domainEmptySeq* [*simp*]: $DOMAIN\ \langle\rangle = \{\}$
by (*simp add*: *emptySeq-def*)

lemma *domainAppend* [*simp*]: $DOMAIN\ Append(s,e) = 1\ ..\ Succ[Len(s)]$
by (*simp add*: *Append-def*)

lemma *isEmptySeq* [*intro*!]:
$[\![isAFcn(f);\ DOMAIN\ f = \{\}]\!] \Longrightarrow f = \langle\rangle$
$[\![isAFcn(f);\ DOMAIN\ f = \{\}]\!] \Longrightarrow \langle\rangle = f$
by *auto*

— immediate consequence of *isEmptySeq*
lemma *emptySeqEmptyFcn*: $\langle\rangle = [x \in \{\} \mapsto y]$
by *auto*

— Symmetric equation could be a useful rewrite rule (it is applied by TLC)
lemmas *emptyFcnEmptySeq* = *sym*[*OF emptySeqEmptyFcn*, *standard*]

lemma *emptyDomainIsEmptySeq* [*simp*]: $(f \in [\{\} \to S]) = (f = \langle\rangle)$
by *auto*

lemma *seqLenZeroIsEmpty* :
assumes $isASeq(s)$
shows $(Len(s) = 0) = (s = \langle\rangle)$
using *assms* **by** *auto*

lemma *emptySeqIff* [*simp*]:
assumes $isAFcn(s)$
shows $(s = \langle\rangle) = (DOMAIN\ s = \{\} \wedge Len(s) = 0)$

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

lemma *emptySeqIff′* [*simp*]:
  assumes *isAFcn*(*s*)
  shows (⟨⟩ = *s*) = (*DOMAIN s* = {} ∧ *Len*(*s*) = *0*)
using *assms* **by** *auto*

lemma *lenAppend* [*simp*]:
  assumes *isASeq*(*s*)
  shows *Len*(*Append*(*s*,*e*)) = *Succ*[*Len*(*s*)]
using *assms* **by** (*intro LenI*, *auto simp*: *Append-def*)

— *s* ∈ *Seq*(*S*) ⟹ *Len*(*Append*(*s*,*e*)) = *Succ*[*Len*(*s*)]
lemmas *lenAppend′* [*simp*] = *SeqIsASeq*[*THEN lenAppend*, *standard*]

lemma *appendElt* [*simp*]:
  assumes *isASeq*(*s*) **and** *k* ∈ *Nat* **and** *0* < *k* **and** *k* ≤ *Succ*[*Len*(*s*)]
  shows *Append*(*s*,*e*)[*k*] = (*IF k* = *Succ*[*Len*(*s*)] *THEN e ELSE s*[*k*])
using *assms* **by** (*auto simp*: *Append-def*)

lemmas *appendElt′* [*simp*] = *SeqIsASeq*[*THEN appendElt*, *standard*]

lemma *appendElt1* :
  assumes *isASeq*(*s*) **and** *k* ∈ *Nat* **and** *0* < *k* **and** *k* ≤ *Len*(*s*)
  shows *Append*(*s*,*e*)[*k*] = *s*[*k*]
using *assms* **by** (*auto simp*: *Append-def*)

lemmas *appendElt1′* = *SeqIsASeq*[*THEN appendElt1*, *standard*]

lemma *appendElt2* :
  assumes *isASeq*(*s*)
  shows *Append*(*s*,*e*)[*Succ*[*Len*(*s*)]] = *e*
using *assms* **by** (*auto simp*: *Append-def*)

lemmas *appendElt2′* = *SeqIsASeq*[*THEN appendElt2*, *standard*]

lemma *isAppend* [*intro*!]:
  assumes *f*: *isAFcn*(*f*) **and** *dom*: *DOMAIN f* = *1* .. *Succ*[*Len*(*s*)] **and** *s*: *isASeq*(*s*)
  and *elt1*: ∀ *n* ∈ *1* .. *Len*(*s*) : *f*[*n*] = *s*[*n*] **and** *elt2*: *f*[*Succ*[*Len*(*s*)]] = *e*
  shows *f* = *Append*(*s*,*e*)
proof (*rule fcnEqual*[*OF f*])
  from *s* **show** *isAFcn*(*Append*(*s*,*e*)) **by** *simp*
next
  from *dom* **show** *DOMAIN f* = *DOMAIN Append*(*s*,*e*) **by** *simp*
next
  from *s elt1 elt2* **show** ∀ *x* ∈ *DOMAIN Append*(*s*, *e*) : *f*[*x*] = *Append*(*s*, *e*)[*x*]
    by (*auto simp*: *Append-def*)
qed

lemmas *isAppend'* [*intro*!] = *isAppend*[*symmetric*, *standard*]

lemma *appendInSeq* [*simp*]:
  assumes *s*: $s \in Seq(S)$ **and** *e*: $e \in S$
  shows $Append(s,e) \in Seq(S)$
using *assms* **by** (*force simp*: *nat-leq-Succ*)

lemma *appendD1*:
  assumes *s*: $isASeq(s)$ **and** *t*: $isASeq(t)$ **and** *app*: $Append(s,e) = Append(t,f)$
  shows $s = t$
proof −
  let $?s1 = Append(s,e)$
  let $?t1 = Append(t,f)$
  from *s* **have** *1*: $isASeq(?s1)$ **by** *simp*
  from *t* **have** *2*: $isASeq(?t1)$ **by** *simp*
  from *1 2 app* **have** *len*: $Len(?s1) = Len(?t1)$ **and** *elt*: $\forall k \in 1 .. Len(?t1) : ?s1[k] = ?t1[k]$
    by (*blast elim*: *seqEqualE*)+
  from *s t len* **have** *ls*: $Len(s) = Len(t)$ **by** *simp*
  thus *?thesis*
  proof (*rule seqEqualI*[*OF s t*], *auto*)
    fix *k*
    assume *k*: $k \in 1 .. Len(t)$
    with *s ls* **have** $s[k] = ?s1[k]$ **by** (*intro sym*[*OF appendElt1*], *auto*)
    also from *k elt t* **have** ... $= ?t1[k]$ **by** *auto*
    also from *t k* **have** ... $= t[k]$ **by** (*intro appendElt1*, *auto*)
    finally show $s[k] = t[k]$ **.**
  qed
qed

lemma *appendD2*:
  assumes *s*: $isASeq(s)$ **and** *t*: $isASeq(t)$ **and** *app*: $Append(s,e) = Append(t,f)$
  shows $e = f$
proof −
  let $?s1 = Append(s,e)$
  let $?t1 = Append(t,f)$
  from *s* **have** *1*: $isASeq(?s1)$ **by** *simp*
  from *t* **have** *2*: $isASeq(?t1)$ **by** *simp*
  from *1 2 app* **have** $Len(?s1) = Len(?t1)$ **and** $\forall k \in 1 .. Len(?t1) : ?s1[k] = ?t1[k]$
    by (*blast elim*: *seqEqualE*)+
  with *s t* **have** $?s1[Len(?s1)] = ?t1[Len(?t1)]$ **by** *auto*
  with *s t* **show** *?thesis* **by** *simp*
qed

lemma *appendEqualIff* [*simp*]:
  assumes *s*: $isASeq(s)$ **and** *t*: $isASeq(t)$
  shows $(Append(s,e) = Append(t,f)) = (s = t \wedge e = f)$

```
using appendD1[OF s t] appendD2[OF s t] by auto
```

The following lemma gives a possible alternative definition of *Append*.

```
lemma appendExtend:
  assumes isASeq(s)
  shows Append(s,e) = s @@ (Succ[Len(s)] :> e)
using assms by force

lemma imageEmptySeq [simp]: Image(⟨⟩, A) = {}
by (simp add: emptySeq-def)

lemma imageAppend [simp]:
  assumes s: isASeq(s)
  shows Image(Append(s,e), A) =
        (IF Succ[Len(s)] ∈ A THEN addElt(e, Image(s,A)) ELSE Image(s,A))
unfolding appendExtend[OF s]
using assms by (auto elim!: inNatIntervalE, force+)
```

Inductive reasoning about sequences, based on ⟨⟩ and *Append*.

```
lemma seqInduct [case-names empty append, induct set: Seq]:
  assumes s: s ∈ Seq(S)
  and base: P(⟨⟩)
  and step: ⋀s e. ⟦s ∈ Seq(S); e ∈ S; P(s)⟧ ⟹ P(Append(s,e))
  shows P(s)
proof −
  have ∀ n ∈ Nat : ∀ s ∈ [1 .. n → S] : P(s) (is ∀ n ∈ Nat : ?A(n))
  proof (rule natInduct)
    from base show ?A(0) by (auto del: funcSetE')
  next
    fix n
    assume n: n ∈ Nat and ih: ?A(n)
    show ?A(Succ[n])
    proof
      fix sn
      assume sn: sn ∈ [1 .. Succ[n] → S]
      def so ≡ [k ∈ 1 .. n ↦ sn[k]]
      def lst ≡ sn[Succ[n]]
      have 1: sn = Append(so, lst)
      proof
 from sn show isAFcn(sn) by simp
      next
 from sn n show DOMAIN sn = 1 .. Succ[Len(so)]
   by (simp add: so-def LenI)
      next
 from n show isASeq(so) by (force simp: so-def)
      next
        from n show ∀ k ∈ 1 .. Len(so) : sn[k] = so[k]
          by (auto simp: so-def LenI)
      next
```

from *n* **show** $sn[Succ[Len(so)]] = lst$
  by (*simp add*: *so-def lst-def LenI*)
    qed
    from *sn n* **have** *2*: $so \in [1\ ..\ n \to S]$
by (*force simp*: *so-def*)
    with *ih* **have** *3*: $P(so)$ **..**
    from *2 n* **have** *4*: $so \in Seq(S)$
      unfolding *Seq-def* **by** *auto*
    from *sn n* **have** $lst \in S$ **by** (*auto simp*: *lst-def*)
    with *1 3 4* **show** $P(sn)$ **by** (*auto intro*: *step*)
   qed
  qed
  with *s* **show** *?thesis* **unfolding** *Seq-def* **by** *auto*
qed

— example of an inductive proof about sequences
lemma *seqEmptyOrAppend*:
  assumes $s \in Seq(S)$
  shows $s = \langle\rangle \lor (\exists\, s' \in Seq(S)\text{: } \exists\, e \in S : s = Append(s',e))$
using *assms* **by** (*induct s*, *auto*)

lemma *seqCases* [*case-names Empty Append*, *cases set*: *Seq*]:
  assumes $s \in Seq(S)$
  and $s = \langle\rangle \Longrightarrow P$ **and** $\bigwedge t\ e.\ [\![t \in Seq(S);\ e \in S;\ s = Append(t,e)]\!] \Longrightarrow P$
  shows P
using *assms* **by** (*auto dest*: *seqEmptyOrAppend*)

8.3 Enumerated sequences

We introduce the conventional syntax $\langle a,b,c\rangle$ for tuples and enumerated sequences, based on the above constructors.

nonterminal *tpl*

syntax
  :: $c \Rightarrow tpl$ (-)
  @*app* :: $[tpl,\ c] \Rightarrow tpl$ (-,/ -)
  @*tuple* :: $tpl \Rightarrow c$ (<<(-)>>)

syntax (*xsymbols*)
  @*tuple* :: $tpl \Rightarrow c$ (⟨(-)⟩)

syntax (*HTML* **output**)
  @*tuple* :: $tpl \Rightarrow c$ (⟨(-)⟩)

translations
  $\langle\ tp,\ x\ \rangle \rightleftharpoons$ *CONST Append*$(\langle tp\rangle,\ x)$
  $\langle\ x\ \rangle \rightleftharpoons$ *CONST Append*$(\langle\rangle,\ x)$

TLA$^+$ has a form of quantification over tuples written $\exists\langle x, y, z\rangle \in S$:

$P(x, y, z)$. We cannot give a generic definition of this form for arbitrary tuples, but introduce input syntax for tuples of length up to 5.

syntax
```
  @bEx2 :: [idt,idt,c,c] ⇒ c ((3EX <<-,->> in - :/ -) [100,100,0,0] 10)
  @bEx3 :: [idt,idt,idt,c,c] ⇒ c ((3EX <<-,-,->> in - :/ -) [100,100,100,0,0] 10)
  @bEx4 :: [idt,idt,idt,idt,c,c] ⇒ c ((3EX <<-,-,-,->> in - :/ -) [100,100,100,100,0,0] 10)
  @bEx5 :: [idt,idt,idt,idt,idt,c,c] ⇒ c ((3EX <<-,-,-,-,->> in - :/ -) [100,100,100,100,100,0,0] 10)
  @bAll2 :: [idt,idt,c,c] ⇒ c ((3ALL <<-,->> in - :/ -) [100,100,0,0] 10)
  @bAll3 :: [idt,idt,idt,c,c] ⇒ c ((3ALL <<-,-,->> in - :/ -) [100,100,100,0,0] 10)
  @bAll4 :: [idt,idt,idt,idt,c,c] ⇒ c ((3ALL <<-,-,-,->> in - :/ -) [100,100,100,100,0,0] 10)
  @bAll5 :: [idt,idt,idt,idt,idt,c,c] ⇒ c ((3ALL <<-,-,-,-,->> in - :/ -) [100,100,100,100,100,0,0] 10)
```

syntax (*xsymbols*)
```
  @bEx2 :: [idt,idt,c,c] ⇒ c ((3∃ ⟨-,-⟩ ∈ - :/ -) [100,100,0,0] 10)
  @bEx3 :: [idt,idt,idt,c,c] ⇒ c ((3∃ ⟨-,-,-⟩ ∈ - :/ -) [100,100,100,0,0] 10)
  @bEx4 :: [idt,idt,idt,idt,c,c] ⇒ c ((3∃ ⟨-,-,-,-⟩ ∈ - :/ -) [100,100,100,100,0,0] 10)
  @bEx5 :: [idt,idt,idt,idt,idt,c,c] ⇒ c ((3∃ ⟨-,-,-,-,-⟩ ∈ - :/ -) [100,100,100,100,100,0,0] 10)
  @bAll2 :: [idt,idt,c,c] ⇒ c ((3∀ ⟨-,-⟩ ∈ - :/ -) [100,100,0,0] 10)
  @bAll3 :: [idt,idt,idt,c,c] ⇒ c ((3∀ ⟨-,-,-⟩ ∈ - :/ -) [100,100,100,0,0] 10)
  @bAll4 :: [idt,idt,idt,idt,c,c] ⇒ c ((3∀ ⟨-,-,-,-⟩ ∈ - :/ -) [100,100,100,100,0,0] 10)
  @bAll5 :: [idt,idt,idt,idt,idt,c,c] ⇒ c ((3∀ ⟨-,-,-,-,-⟩ ∈ - :/ -) [100,100,100,100,100,0,0] 10)
```

translations
```
  ∃ ⟨x,y⟩ ∈ S : P          ⇀ ∃ x,y : ⟨x,y⟩ ∈ S ∧ P
  ∃ ⟨x,y,z⟩ ∈ S : P        ⇀ ∃ x,y,z : ⟨x,y,z⟩ ∈ S ∧ P
  ∃ ⟨x,y,z,u⟩ ∈ S : P      ⇀ ∃ x,y,z,u : ⟨x,y,z,u⟩ ∈ S ∧ P
  ∃ ⟨x,y,z,u,v⟩ ∈ S : P    ⇀ ∃ x,y,z,u,v : ⟨x,y,z,u,v⟩ ∈ S ∧ P
  ∀ ⟨x,y⟩ ∈ S : P          ⇀ ∀ x,y : ⟨x,y⟩ ∈ S ⇒ P
  ∀ ⟨x,y,z⟩ ∈ S : P        ⇀ ∀ x,y,z : ⟨x,y,z⟩ ∈ S ⇒ P
  ∀ ⟨x,y,z,u⟩ ∈ S : P      ⇀ ∀ x,y,z,u : ⟨x,y,z,u⟩ ∈ S ⇒ P
  ∀ ⟨x,y,z,u,v⟩ ∈ S : P    ⇀ ∀ x,y,z,u,v : ⟨x,y,z,u,v⟩ ∈ S ⇒ P
```

8.4 Sets of finite functions

We introduce notation such as $[x\colon S,\ y\colon T]$ to designate the set of finite functions f with domain $\{x,y\}$ (for constants x, y) such that $f[x] \in S$ and $f[y] \in T$. Typically, elements of such a function set will be constructed as $(x :> s)@@(y :> t)$. This notation for sets of finite functions generalizes similar TLA$^+$ notation for records.

Internally, the set is represented as $EnumFuncSet(\langle x,y\rangle,\ \langle S,T\rangle)$, using appropriate translation functions between the internal and external representations.

definition *EnumFuncSet* :: $c \Rightarrow c \Rightarrow c$
where $EnumFuncSet(doms,\ rngs) \equiv \{\ f \in [Range(doms) \to UNION\ Range(rngs)]$
:
$\forall\, i \in DOMAIN\ doms : f[doms[i]] \in rngs[i]\ \}$

lemmas — establish set equality for sets of enumerated functions
setEqualI [**where** $A = EnumFuncSet(doms,\ rngs)$, *standard*, *intro*!]
setEqualI [**where** $B = EnumFuncSet(doms,\ rngs)$, *standard*, *intro*!]

lemma *EnumFuncSetI* [*intro*!,*simp*]:
assumes *1*: $isAFcn(f)$ **and** *2*: $DOMAIN\, f = Range(doms)$
and *3*: $DOMAIN\ rngs = DOMAIN\ doms$
and *4*: $\forall\, i \in DOMAIN\ doms$: $f[doms[i]] \in rngs[i]$
shows $f \in EnumFuncSet(doms,\ rngs)$

proof −
from *1 2* **have** $f \in [Range(doms) \to UNION\ Range(rngs)]$
proof
from *3 4* **show** $\forall\, x \in Range(doms) : f[x] \in UNION\ Range(rngs)$ **by** *force*
qed
with *4* **show** *?thesis* **by** (*simp add*: *EnumFuncSet-def*)
qed

lemma *EnumFuncSetE* [*elim*!]:
assumes $f \in EnumFuncSet(doms,\ rngs)$
and $[\![f \in [Range(doms) \to UNION\ Range(rngs)];$
$\forall\, i \in DOMAIN\ doms : f[doms[i]] \in rngs[i]\ ]\!] \Longrightarrow P$
shows P
using *assms* **by** (*auto simp*: *EnumFuncSet-def*)

nonterminal *domrng* **and** *domrngs*

syntax
@*domrng* :: [*c*, *c*] ⇒ *domrng* ((*2*- :/ -) *10*)
:: *domrng* ⇒ *domrngs* (-)
@*domrngs* :: [*domrng*, *domrngs*] ⇒ *domrngs* (-,/ -)
@*EnumFuncSet*:: *domrngs* ⇒ *c* ([-])

parse-ast-translation ⟪
let
(∗ *make-tuple converts a list of ASTs to a tuple formed from these ASTs.*
The order of elements is reversed. ∗)
fun make-tuple [] = *Ast.Constant emptySeq*
| *make-tuple* (*t* :: *ts*) = *Ast.Appl*[*Ast.Constant Append*, *make-tuple ts*, *t*]

```
    (* get-doms-ranges extracts the lists of arguments and ranges
       from the arms of a domrngs expression.
       The order of the ASTs is reversed. *)
    fun get-doms-ranges (Ast.Appl[Ast.Constant @domrng, d, r]) =
          (* base case: one domain, one range *)
          ([d], [r])
      | get-doms-ranges (Ast.Appl[Ast.Constant @domrngs,
                                  Ast.Appl[Ast.Constant @domrng, d, r],
                                  pairs]) =
          (* one domrng, followed by remaining doms and ranges *)
          let val (ds, rs) = get-doms-ranges pairs
          in  (ds @ [d], rs @ [r])
          end
    fun enum-funcset-tr [pairs] =
        let val (doms, rngs) = get-doms-ranges pairs
            val dTuple = make-tuple doms
            val rTuple = make-tuple rngs
        in
          Ast.Appl[Ast.Constant EnumFuncSet, dTuple, rTuple]
        end
      | enum-funcset-tr - = raise Match;
  in
    [(@EnumFuncSet, enum-funcset-tr)]
  end
»
```

print-ast-translation «

```
  let
    fun list-from-tuple (Ast.Constant @{const-syntax emptySeq}) = []
      | list-from-tuple (Ast.Appl[Ast.Constant @tuple, tp]) =
        let fun list-from-tp (Ast.Appl[Ast.Constant @app, tp, t]) =
                  (list-from-tp tp) @ [t]
              | list-from-tp t = [t]
        in  list-from-tp tp
        end
    (* make-domrngs constructs an AST representing the domain/range pairs.
       The result is an AST of type domrngs.
       The lists of domains and ranges must be of equal length and non−empty. *)
   fun make-domrngs [d] [r] = Ast.Appl[Ast.Constant @{syntax-const @domrng},
d, r]
      | make-domrngs (d::ds) (r::rs) =
          Ast.Appl[Ast.Constant @{syntax-const @domrngs},
                Ast.Appl[Ast.Constant @{syntax-const @domrng}, d, r],
                make-domrngs ds rs]
    fun enum-funcset-tr' [dTuple, rTuple] =
        let val doms = list-from-tuple dTuple
            val rngs = list-from-tuple rTuple
        in  (* make sure that lists are of equal length, otherwise give up *)
            if length doms = length rngs
```

```
        then Ast.Appl[Ast.Constant @{syntax-const @EnumFuncSet}, make-domrngs
doms rngs]
          else raise Match
        end
      | enum-funcset-tr' - = raise Match
  in
    [(@{const-syntax EnumFuncSet}, enum-funcset-tr')]
  end
⟫
```

8.5 Set product

The cartesian product of two sets A and B is the set of pairs whose first component is in A and whose second component is in B. We generalize the definition of products to an arbitrary number of sets: $Product(\langle A_1, \ldots, A_n \rangle) = A_1 \times \cdots \times A_n$.

definition *Product*
where $Product(s) \equiv \{ f \in [1 \,..\, Len(s) \to UNION\ Range(s)] :$
$\forall\, i \in 1 \,..\, Len(s) : f[i] \in s[i] \}$

lemma *inProductI* [*intro*!]:
  assumes $isASeq(p)$ **and** $isASeq(s)$ **and** $Len(p) = Len(s)$
  and $\forall\, k \in 1 \,..\, Len(s) : p[k] \in s[k]$
  shows $p \in Product(s)$
using *assms* **by** (*auto simp add*: *Product-def*)

lemma *inProductIsASeq*:
  assumes $p \in Product(s)$ **and** $isASeq(s)$
  shows $isASeq(p)$
using *assms* **by** (*auto simp add*: *Product-def*)

lemma *inProductLen*:
  assumes $p \in Product(s)$ **and** $isASeq(s)$
  shows $Len(p) = Len(s)$
using *assms* **by** (*auto simp add*: *Product-def*)

lemma *inProductE* [*elim*!]:
  assumes $p \in Product(s)$ **and** $isASeq(s)$
  and $[\![isASeq(p);\ Len(p) = Len(s);\ p \in [1 \,..\, Len(s) \to UNION\ Range(s)];$
  $\forall\, k \in 1 \,..\, Len(s) : p[k] \in s[k]\ ]\!] \Longrightarrow P$
  shows P
using *assms* **by** (*auto simp add*: *Product-def*)

Special case: binary product

definition
  $prod :: c \Rightarrow c \Rightarrow c$ (**infixr** \X *100*) **where**
  A \X $B \equiv Product(\langle A,B \rangle)$
notation (*xsymbols*)

prod (**infixr** × *100*)
notation (*HTML* **output**)
prod (**infixr** × *100*)

lemma *inProd* [*simp*]:
(⟨*a*,*b*⟩ ∈ *A* × *B*) = (*a* ∈ *A* ∧ *b* ∈ *B*)
by (*auto simp add*: *prod-def*)

lemma *prodProj*:
assumes *p*: *p* ∈ *A* × *B*
shows *p* = ⟨*p*[*1*], *p*[*2*]⟩
using *assms* **by** (*auto simp add*: *prod-def*)

lemma *inProd'*:
(*p* ∈ *A* × *B*) = (∃ *a* ∈ *A* : ∃ *b* ∈ *B* : *p* = ⟨*a*,*b*⟩)
proof (*auto*)
assume *p*: *p* ∈ *A* × *B*
hence *1*: *p* = ⟨*p*[*1*], *p*[*2*]⟩ **by** (*rule prodProj*)
with *p* **have** ⟨*p*[*1*], *p*[*2*]⟩ ∈ *A* × *B* **by** *simp*
hence *p*[*1*] ∈ *A* *p*[*2*] ∈ *B* **by** *auto*
with *1* **show** ∃ *a* ∈ *A* : ∃ *b* ∈ *B* : *p* = ⟨*a*, *b*⟩ **by** *auto*
qed

lemma *inProdI* [*intro*]:
assumes *a*: *a* ∈ *A* **and** *b*: *b* ∈ *B* **and** *p*: *P*(⟨*a*,*b*⟩)
shows ∃ *p* ∈ *A* × *B* : *P*(*p*)
using *assms* **by** (*intro bExI*[*of* ⟨*a*,*b*⟩], *simp*+)

lemma *inProdI'*:
assumes *a*: *a* ∈ *A* **and** *b*: *b* ∈ *B*
obtains *p* **where** *p* ∈ *A* × *B* **and** *p*[*1*] = *a* **and** *p*[*2*] = *b*
proof
from *a b* **show** ⟨*a*,*b*⟩ ∈ *A* × *B* **by** *simp*
next
show ⟨*a*,*b*⟩[*1*] = *a* **by** *simp*
next
show ⟨*a*,*b*⟩[*2*] = *b* **by** *simp*
qed

lemma *inProdE* [*elim*]:
assumes *p* ∈ *A* × *B*
and ⋀*a b*. ⟦*a* ∈ *A*; *b* ∈ *B*; *p* = ⟨*a*,*b*⟩⟧ ⟹ *P*
shows *P*
using *assms* **by** (*auto simp add*: *inProd'*)

lemma *prodEmptyIff* [*simp*]:
(*A* × *B* = {}) = ((*A* = {}) ∨ (*B* = {}))

proof *auto*
  fix *a b*
  assume *a*: $a \in A$ **and** *b*: $b \in B$ **and** *prod*: $A \times B = \{\}$
  from *a b* **have** $\langle a,b \rangle \in A \times B$ **by** *simp*
  with *prod* **show** *FALSE* **by** *blast*
qed

lemma *prodEmptyIff'* [*simp*]:
  $(\{\} = A \times B) = ((A = \{\}) \vee (B = \{\}))$
proof *auto*
  fix *a b*
  assume *a*: $a \in A$ **and** *b*: $b \in B$ **and** *prod*: $\{\} = A \times B$
  from *a b* **have** $\langle a,b \rangle \in A \times B$ **by** *simp*
  with *prod* **show** *FALSE* **by** *blast*
qed

lemma *pairProj-in-rel* :
  assumes *r*: $r \subseteq A \times B$ **and** *p*: $p \in r$
  shows $\langle p[1],p[2] \rangle \in r$
using *p prodProj*[*OF rev-subsetD*[*OF p r*], *symmetric*] **by** *simp*

lemma *pairProj-in-prod* :
  assumes *r*: $r \subseteq A \times B$ **and** *p*: $p \in r$
  shows $\langle p[1],p[2] \rangle \in A \times B$
using *subsetD*[*OF r p*] *prodProj*[*OF rev-subsetD*[*OF p r*], *symmetric*] **by** *simp*

lemma *relProj1* [*elim*]:
  assumes $\langle a,b \rangle \in r$ **and** $r \subseteq A \times B$
  shows $a \in A$
using *assms* **by** (*auto dest*: *pairProj-in-prod*)

lemma *relProj2* [*elim*]:
  assumes $\langle a,b \rangle \in r$ **and** $r \subseteq A \times B$
  shows $b \in B$
using *assms* **by** (*auto dest*: *pairProj-in-prod*)

lemma *setOfAllPairs-eq-r* :
  assumes *r*: $r \subseteq A \times B$
  shows $\{\langle p[1], p[2] \rangle : p \in r\} = r$
apply *auto*
using *subsetD*[*OF r*, *THEN prodProj*[*of - A B*]] **by** *simp-all*

lemma *subsetsInProd*:
  assumes $A' \subseteq A$ **and** $B' \subseteq B$
  shows $A' \times B' \subseteq A \times B$
unfolding *prod-def Product-def*
using *assms* **by** *auto*

8.6 Syntax for setOfPairs: $\{e : \langle x,y \rangle \in R\}$

definition *setOfPairs* :: [*c*, [*c*,*c*]⇒*c*] ⇒ *c*
where *setOfPairs*(*R*, *f*) ≡ { *f*(*p*[*1*], *p*[*2*]) : *p* ∈ *R* }

syntax
  @*setOfPairs* :: [*c*, *idt*, *idt*, *c*] ⇒ *c*   ((*1*{- : <<-,->> \in -}))
syntax (*xsymbols*)
  @*setOfPairs* :: [*c*, *idt*, *idt*, *c*] ⇒ *c*   ((*1*{- : ⟨-,-⟩ ∈ -}))
translations
  {*e* : ⟨*x*,*y*⟩ ∈ *R*} ⇌ *CONST setOfPairs*(*R*, λ*x y*. *e*)

lemma *inSetOfPairsI-ex*:
  assumes ∃ ⟨*x*,*y*⟩ ∈ *R* : *a* = *e*(*x*,*y*)
  shows *a* ∈ { *e*(*x*,*y*) : ⟨*x*,*y*⟩ ∈ *R* }
using *assms* **by** (*auto simp*: *setOfPairs-def*)

lemma *inSetOfPairsI* [*intro*]:
  assumes *a* = *e*(*x*,*y*) **and** ⟨*x*,*y*⟩ ∈ *R*
  shows *a* ∈ *setOfPairs*(*R*, *e*)
using *assms* **by** (*auto simp*: *setOfPairs-def*)

lemma *inSetOfPairsE* [*elim*!]: — converse true only if *R* is a relation
  assumes *1*: *z* ∈ *setOfPairs*(*R*, *e*)
    and *2*: *R* ⊆ *A* × *B* **and** *3*: ⋀*x y*. ⟦ ⟨*x*,*y*⟩ ∈ *R*; *z* = *e*(*x*,*y*) ⟧ ⟹ *P*
  shows *P*
proof −
  from *1* **obtain** *p* **where** *pR*: *p* ∈ *R* **and** *pz*: *z* = *e*(*p*[*1*],*p*[*2*])
   by (*auto simp*: *setOfPairs-def*)
  from *pR 2* **have** *p* = ⟨*p*[*1*], *p*[*2*]⟩ **by** (*intro prodProj*, *auto*)
  with *pR pz* **show** *P* **by** (*intro 3*, *auto*)
qed

lemmas *setOfPairsEqualI* =
  setEqualI [**where** *A* = *setOfPairs*(*R*,*f*), *standard*,*intro*!]
  setEqualI [**where** *B* = *setOfPairs*(*R*,*f*), *standard*,*intro*!]

lemma *setOfPairs-triv* [*simp*]:
  assumes *s*: *R* ⊆ *A* × *B*
  shows { ⟨*x*,*y*⟩ : ⟨*x*,*y*⟩ ∈ *R* } = *R*
using *assms* **by** *auto*

lemma *setOfPairs-cong* :
  assumes *1*: *R* = *S* **and** *2*: *S* ⊆ *A* × *B* **and** *3*: ⋀*x y*. ⟨*x*,*y*⟩ ∈ *S* ⟹ *e*(*x*,*y*) = *f*(*x*,*y*)
  shows { *e*(*x*,*y*) : ⟨*x*,*y*⟩ ∈ *R* } = { *f*(*u*,*v*) : ⟨*u*,*v*⟩ ∈ *S* }
using *assms* **proof** (*auto*)
  fix *u v*
  let *?p* = ⟨*u*,*v*⟩
  assume *uv*: *?p* ∈ *S*

with *3* **have** $f(u,v) = e(?p[1], ?p[2])$ **by** *simp*
with *uv* **show** $f(u,v) \in setOfPairs(S, e)$ **by** *auto*
qed

lemma *setOfPairsEqual*:
assumes *1*: $\bigwedge x\ y.\ \langle x,y\rangle \in S \Longrightarrow \exists \langle x',y'\rangle \in T : e(x,y) = f(x',y')$
and *2*: $\bigwedge x'\ y'.\ \langle x',y'\rangle \in T \Longrightarrow \exists \langle x,y\rangle \in S : f(x',y') = e(x,y)$
and *3*: $S \subseteq A \times B$ **and** *4*: $T \subseteq C \times D$
shows $\{ e(x,y) : \langle x,y\rangle \in S \} = \{ f(x,y) : \langle x,y\rangle \in T \}$
using *assms* **by** (*auto*, *blast*+)

8.7 Basic notions about binary relations

definition *rel-domain* :: *c* => *c*
where $rel\text{-}domain(r) \equiv \{ p[1] : p \in r \}$

definition *rel-range* :: *c* => *c*
where $rel\text{-}range(r) \equiv \{ p[2] : p \in r \}$

definition *converse* :: *c* => *c* ((-ˆ−1) [*1000*] *999*)
where $r\hat{\ }{-1} \equiv \{ \langle p[2],p[1]\rangle : p \in r\}$

definition *rel-comp* :: [*c*,*c*] => *c* (**infixr** ∘ *75*) — binary relation composition
where $r \circ s \equiv \{ p \in rel\text{-}domain(s) \times rel\text{-}range(r) :$
$\exists x,z : p = \langle x,z\rangle \wedge (\exists y: \langle x,y\rangle \in s \wedge \langle y,z\rangle \in r) \}$

definition *rel-image* :: [*c*,*c*] => *c* (**infixl** `` *90*)
where $r \text{ `` } A \equiv \{y \in rel\text{-}range(r) : \exists x \in A: \langle x,y\rangle \in r\}$

definition *Id* :: $c \Rightarrow c$ — diagonal: identity over a set
where $Id(A) \equiv \{ \langle x,x\rangle : x \in A \}$

Properties of relations

definition *reflexive* — reflexivity over a set
where $reflexive(A,r) \equiv \forall x \in A: \langle x,x\rangle \in r$

lemma *boolifyReflexive* [*simp*]: $boolify(reflexive(A,r)) = reflexive(A,r)$
unfolding *reflexive-def* **by** *simp*

lemma *reflexiveIsBool*[*intro*!,*simp*]: $isBool(reflexive(A,r))$
unfolding *isBool-def* **by** (*rule boolifyReflexive*)

definition *symmetric* — symmetric relation
where $symmetric(r) \equiv \forall x,y: \langle x,y\rangle \in r \Rightarrow \langle y,x\rangle \in r$

lemma *boolifySymmetric* [*simp*]: $boolify(symmetric(r)) = symmetric(r)$
unfolding *symmetric-def* **by** *simp*

lemma *symmetricIsBool*[*intro*!,*simp*]: $isBool(symmetric(r))$

unfolding *isBool-def* **by** (*rule boolifySymmetric*)

definition *antisymmetric* — antisymmetric relation
where $antisymmetric(r) \equiv \forall\, x,y\colon \langle x,y\rangle \in r \wedge \langle y,x\rangle \in r \Rightarrow x = y$

lemma *boolifyAntisymmetric* [*simp*]: $boolify(antisymmetric(r)) = antisymmetric(r)$
unfolding *antisymmetric-def* **by** *simp*

lemma *antisymmetricIsBool*[*intro*!,*simp*]: $isBool(antisymmetric(r))$
unfolding *isBool-def* **by** (*rule boolifyAntisymmetric*)

definition *transitive* — transitivity predicate
where $transitive(r) \equiv \forall\, x,y,z\colon \langle x,y\rangle \in r \wedge \langle y,z\rangle \in r \Rightarrow \langle x,z\rangle \in r$

lemma *boolifyTransitive* [*simp*]: $boolify(transitive(r)) = transitive(r)$
unfolding *transitive-def* **by** *simp*

lemma *transitiveIsBool*[*intro*!,*simp*]: $isBool(transitive(r))$
unfolding *isBool-def* **by** (*rule boolifyTransitive*)

definition *irreflexive* — irreflexivity predicate
where $irreflexive(A,r) \equiv \forall\, x \in A\colon \langle x,x\rangle \notin r$

lemma *boolifyIrreflexive* [*simp*]: $boolify(irreflexive(A,r)) = irreflexive(A,r)$
unfolding *irreflexive-def* **by** *simp*

lemma *irreflexiveIsBool*[*intro*!,*simp*]: $isBool(irreflexive(A,r))$
unfolding *isBool-def* **by** (*rule boolifyIrreflexive*)

definition *equivalence* :: $[c,c] \Rightarrow c$ — (partial) equivalence relation
where $equivalence(A,r) \equiv reflexive(A,r) \wedge symmetric(r) \wedge transitive(r)$

lemma *boolifyEquivalence* [*simp*]: $boolify(equivalence(A,r)) = equivalence(A,r)$
unfolding *equivalence-def* **by** *simp*

lemma *equivalenceIsBool*[*intro*!,*simp*]: $isBool(equivalence(A,r))$
unfolding *isBool-def* **by** (*rule boolifyEquivalence*)

8.7.1 Domain and Range

lemma *prod-in-dom-x-ran*:
assumes $r \subseteq A \times B$ **and** $p \in r$
shows $\langle p[1],p[2]\rangle \in rel\text{-}domain(r) \times rel\text{-}range(r)$
unfolding *inProd rel-domain-def rel-range-def*
using *assms* **by** *auto*

lemma *in-rel-domainI* [*iff*]:
assumes $\langle x,y\rangle \in r$
shows $x \in rel\text{-}domain(r)$

unfolding *rel-domain-def* **using** *assms* **by** *auto*

lemma *in-rel-domainE* [*elim*]:
  assumes *x*: $x \in$ *rel-domain*(r) **and** *r*: $r \subseteq A \times B$ **and** *p*: $\bigwedge y.\ \langle x,y\rangle \in r \Longrightarrow P$
  shows P
proof −
  from *x* **obtain** *p* **where** *1*: $p \in r$ **and** *2*: $p[1] = x$
    by (*auto simp add*: *rel-domain-def*)
  from *1 r* **have** $p = \langle p[1],p[2]\rangle$ **by** (*intro prodProj*, *auto*)
  with *1 2* **show** P **by** (*intro p*[**where** $y=p[2]$], *simp*)
qed

lemma *rel-domain* : $r \subseteq A \times B \Longrightarrow$ *rel-domain*(r) $\subseteq A$
unfolding *rel-domain-def* **using** *pairProj-in-prod* **by** *auto*

lemma *rel-range* : $r \subseteq A \times B \Longrightarrow$ *rel-range*(r) $\subseteq B$
unfolding *rel-range-def* **using** *pairProj-in-prod* **by** *auto*

lemma *in-rel-rangeI* [*iff*]:
  assumes $\langle x,y\rangle \in r$
  shows $y \in$ *rel-range*(r)
unfolding *rel-range-def* **using** *assms* **by** *auto*

lemma *in-rel-rangeE* [*elim*]:
  assumes *y*: $y \in$ *rel-range*(r) **and** *r*: $r \subseteq A \times B$ **and** *p*: $\bigwedge x.\ \langle x,y\rangle \in r \Longrightarrow P$
  shows P
proof −
  from *y* **obtain** *p* **where** *1*: $p \in r$ **and** *2*: $p[2] = y$
    by (*auto simp add*: *rel-range-def*)
  from *1 r* **have** $p = \langle p[1],p[2]\rangle$ **by** (*intro prodProj*, *auto*)
  with *1 2* **show** P **by** (*intro p*[**where** $x=p[1]$], *simp*)
qed

lemma *dom-in-A* : *rel-domain* ($\{ p \in A \times B : P(p) \}$) $\subseteq A$
by *auto*

lemma *ran-in-B* : *rel-range* ($\{ p \in A \times B : P(p) \}$) $\subseteq B$
by *auto*

lemma *subrel-dom*: $r' \subseteq r \Longrightarrow x \in$ *rel-domain*(r') $\Longrightarrow x \in$ *rel-domain*(r)
unfolding *rel-domain-def* **by** *auto*

lemma *subrel-ran*: $r' \subseteq r \Longrightarrow x \in$ *rel-range*(r') $\Longrightarrow x \in$ *rel-range*(r)
unfolding *rel-range-def* **by** *auto*

lemma *in-dom-imp-in-A*: $r \subseteq A \times B \Longrightarrow x \in$ *rel-domain*(r) $\Longrightarrow x \in A$
by *force*

lemma *in-ran-imp-in-B*: $r \subseteq A \times B \Longrightarrow p \in$ *rel-range*(r) $\Longrightarrow p \in B$

by *force*

8.7.2 Converse relation

lemmas *converseEqualI* =
  setEqualI [**where** $A = r\hat{\ }{-1}$, *standard*, *intro*!]
  setEqualI [**where** $B = r\hat{\ }{-1}$, *standard*, *intro*!]

lemma *converse-iff* [*iff*]:
  assumes *r*: $r \subseteq A \times B$
  shows $(\langle a,b\rangle \in r\hat{\ }{-1}) = (\langle b,a\rangle \in r)$
using *r prodProj* **by** (*auto simp*: *converse-def*)

lemma *converseI* [*intro*!]:
  shows $\langle a,b\rangle \in r \Longrightarrow \langle b,a\rangle \in r\hat{\ }{-1}$
unfolding *converse-def* **by** *auto*

lemma *converseD* [*sym*]:
  assumes *r*: $r \subseteq A \times B$
  shows $\langle a,b\rangle \in r\hat{\ }{-1} \Longrightarrow \langle b,a\rangle \in r$
using *converse-iff* [*OF r*] **by** *simp*

lemma *converseSubset*: $r \subseteq A \times B \Longrightarrow r\hat{\ }{-1} \subseteq B \times A$
unfolding *converse-def* **using** *pairProj-in-prod* **by** *auto*

lemma *converseE* [*elim*]:
  assumes *yx*: $yx \in r\hat{\ }{-1}$ **and** *r*: $r \subseteq A \times B$
    and *p*: $\bigwedge x\ y.\ yx = \langle y,x\rangle \Longrightarrow \langle x,y\rangle \in r \Longrightarrow P$
  shows *P*
  — More general than *converseD*, as it “splits” the member of the relation.
proof −
  from *prodProj*[*OF subsetD*[*OF converseSubset*[*OF r*] *yx*]] **have** *2*: $yx = \langle yx[1], yx[2]\rangle$.
  with *yx* **have** *3*: $\langle yx[2], yx[1]\rangle \in r$
    unfolding *converse-def* **apply** *auto*
    using *r prodProj* **by** *auto*
  from *p*[*of yx*[*1*] *yx*[*2*]] *2 3*
  show *P* **by** *simp*
qed

lemma *converse-converse* [*simp*]:
  assumes *r*: $r \subseteq A \times B$
  shows $(r\hat{\ }{-1})\hat{\ }{-1} = r$
using *assms prodProj* **by** (*auto elim*!: *converseE*)

lemma *converse-prod* [*simp*]: $(A \times B)\hat{\ }{-1} = B \times A$
using *prodProj* **by** *auto*

lemma *converse-empty* [*simp*]: *converse*($\{\}$) = $\{\}$

by *auto*

lemma *converse-mono-1*:
  assumes *r*: $r \subseteq A \times B$ **and** *s*: $s \subseteq A \times B$ **and** *sub*: $r\hat{\ }{-1} \subseteq s\hat{\ }{-1}$
  shows $r \subseteq s$
proof
  fix *p*
  assume *p*: $p \in r$
  with *r* **have** *1*: $p = \langle p[1],p[2]\rangle$ **by** (*intro prodProj*, *auto*)
  with *p* **have** $\langle p[2],p[1]\rangle \in r\hat{\ }{-1}$ **by** *auto*
  with *sub s 1* **show** $p \in s$ **by** *auto*
qed

lemma *converse-mono-2*:
  assumes $r \subseteq A \times B$ **and** $s \subseteq A \times B$ **and** $r \subseteq s$
  shows $r\hat{\ }{-1} \subseteq s\hat{\ }{-1}$
using *assms prodProj* **by** *auto*

lemma *converse-mono*:
  assumes *r*:$r \subseteq A \times B$ **and** *s*:$s \subseteq A \times B$
  shows $r\hat{\ }{-1} \subseteq s\hat{\ }{-1} = (r \subseteq s)$
using *converse-mono-1*[*OF r s*] *converse-mono-2*[*OF r s*]
by *blast*

lemma *reflexive-converse* [*simp*]:
  $r \subseteq A \times B \Longrightarrow$ *reflexive*(A, $r\hat{\ }{-1}$) = *reflexive*(A,r)
  unfolding *reflexive-def* **by** *auto*

lemma *symmetric-converse* [*simp*]:
  $r \subseteq A \times B \Longrightarrow$ *symmetric*($r\hat{\ }{-1}$) = *symmetric*(r)
  unfolding *symmetric-def* **by** *auto*

lemma *antisymmetric-converse* [*simp*]:
  $r \subseteq A \times B \Longrightarrow$ *antisymmetric*($r\hat{\ }{-1}$) = *antisymmetric*(r)
  unfolding *antisymmetric-def* **by** *auto*

lemma *transitive-converse* [*simp*]:
  $r \subseteq A \times B \Longrightarrow$ *transitive*($r\hat{\ }{-1}$) = *transitive*(r)
  unfolding *transitive-def* **by** *auto*

lemma *symmetric-iff-converse-eq*:
  assumes *r*:$r \subseteq A \times B$
  shows *symmetric*(r) = ($r\hat{\ }{-1} = r$)
proof *auto*
  fix *p*
  assume *symmetric*(r) **and** $p \in r\hat{\ }{-1}$
  with *r* **show** $p \in r$ **by** (*auto elim*!: *converseE simp add*: *symmetric-def*)

next
  fix *p*
  assume *1*: *symmetric*(*r*) **and** *2*: $p \in r$
  from *r 2* **have** *3*: $p = \langle p[1],p[2]\rangle$ **by** (*intro prodProj*, *auto*)
  with *1 2* **have** $\langle p[2],p[1]\rangle \in r$ **by** (*force simp add*: *symmetric-def*)
  with *3* **show** $p \in r\hat{}{-1}$ **by** (*auto dest*: *converseI*)
next
  assume $r\hat{}{-1} = r$ **thus** *symmetric*(*r*)
    by (*auto simp add*: *symmetric-def*)
qed

8.7.3 Identity relation over a set

lemmas *idEqualI* =
  setEqualI [**where** $A = Id(S)$, *standard*, *intro*!]
  setEqualI [**where** $B = Id(S)$, *standard*, *intro*!]

lemma *IdI* [*iff*]: $x \in S \Longrightarrow \langle x,x\rangle \in Id(S)$
unfolding *Id-def* **by** *auto*

lemma *IdI'* [*intro*]: $x \in S \Longrightarrow p = \langle x,x\rangle \Longrightarrow p \in Id(S)$
unfolding *Id-def* **by** *auto*

lemma *IdE* [*elim*!]:
  $p \in Id(S) \Longrightarrow (\bigwedge x.\ x \in S \land p = \langle x,x\rangle \Longrightarrow P) \Longrightarrow P$
unfolding *Id-def* **by** *auto*

lemma *Id-iff*: $(\langle a,b\rangle \in Id(S)) = (a = b \land a \in S)$
by *auto*

lemma *Id-subset-Prod* [*simp*]: $Id(S) \subseteq S \times S$
unfolding *Id-def* **by** *auto*

lemma *reflexive-Id*: *reflexive*(*S*,*Id*(*S*))
unfolding *reflexive-def* **by** *auto*

lemma *antisymmetric-Id* [*simp*]: *antisymmetric*(*Id*(*S*))
unfolding *antisymmetric-def* **by** *auto*

lemma *symmetric-Id* [*simp*]: *symmetric*(*Id*(*S*))
unfolding *symmetric-def* **by** *auto*

lemma *transitive-Id* [*simp*]: *transitive*(*Id*(*S*))
unfolding *transitive-def* **by** *auto*

lemma *Id-empty* [*simp*]: $Id(\{\}) = \{\}$
unfolding *Id-def* **by** *simp*

lemma *Id-eqI*: $a = b \Longrightarrow a \in A \Longrightarrow \langle a,b\rangle \in Id(A)$

by *simp*

lemma *converse-Id* [*simp*]: $Id(A)\hat{\ }{-1} = Id(A)$
by *auto*

lemma *dom-Id* [*simp*]: *rel-domain*$(Id(A)) = A$
unfolding *rel-domain-def Id-def* **by** *auto*

lemma *ran-Id* [*simp*]: *rel-range*$(Id(A)) = A$
unfolding *rel-range-def Id-def* **by** *auto*

8.7.4 Composition of relations

lemmas *compEqualI* =
  setEqualI [**where** $A = r \circ s$, *standard*, *intro*!]
  setEqualI [**where** $B = r \circ s$, *standard*, *intro*!]

lemma *compI* [*intro*]:
  assumes r: $r \subseteq B \times C$ **and** s: $s \subseteq A \times B$
  shows $[\![\langle a,b\rangle \in s; \langle b,c\rangle \in r]\!] \Longrightarrow \langle a,c\rangle \in r \circ s$
using *assms* **unfolding** *rel-comp-def* **by** *auto*

lemma *compE* [*elim*!]:
  assumes $xz \in r \circ s$ **and** $r \subseteq B \times C$ **and** $s \subseteq A \times B$
  shows $(\bigwedge x\ y\ z.\ xz = \langle x,z\rangle \Longrightarrow \langle x,y\rangle \in s \Longrightarrow \langle y,z\rangle \in r \Longrightarrow P) \Longrightarrow P$
using *assms* **unfolding** *rel-comp-def* **by** *auto*

lemma *compEpair*:
  assumes $\langle a,c\rangle \in r \circ s$ **and** $r \subseteq B \times C$ **and** s: $s \subseteq A \times B$
  shows $[\![\bigwedge b.\ [\![\langle a,b\rangle \in s; \langle b,c\rangle \in r]\!] \Longrightarrow P]\!] \Longrightarrow P$
using *assms* **by** *auto*

lemma *rel-comp-in-prod* [*iff*]:
  assumes s: $s \subseteq A \times B$ **and** r: $r \subseteq B \times C$
  shows $r \circ s \subseteq A \times C$
using *assms* **by** *force*

lemma *rel-comp-in-prodE* :
  assumes $p \in r \circ s$ **and** $s \subseteq A \times B$ **and** r: $r \subseteq B \times C$
  shows $p \in A \times C$
using *assms* **by** *force*

lemma *converse-comp*:
  assumes r: $r \subseteq B \times C$ **and** s: $s \subseteq A \times B$
  shows $((r \circ s)\hat{\ }{-1}) = (s\hat{\ }{-1} \circ r\hat{\ }{-1})$ (**is** *?lhs* = *?rhs*)
proof
  fix x
  assume x: $x \in$ *?lhs*
  from $s\ r$ **have** $r \circ s \subseteq A \times C$ **by** (*rule rel-comp-in-prod*)

```
  with x show x ∈ ?rhs
  proof
    fix u w
    assume 1: x = ⟨w,u⟩ and 2: ⟨u,w⟩ ∈ r ∘ s
    from 2 r s obtain v where 3: ⟨u,v⟩ ∈ s and 4: ⟨v,w⟩ ∈ r
      by auto
    with converseSubset[OF r] converseSubset[OF s] have ⟨w,u⟩ ∈ ?rhs
      by auto
    with 1 show x ∈ ?rhs by simp
  qed
next
  fix x
  assume x ∈ ?rhs
  with r s show x ∈ ?lhs by (auto dest: converseSubset)
qed

lemma R-comp-Id [simp]:
  assumes r: R ⊆ B × C
  shows R ∘ Id(B) = R
using r proof auto
  fix p
  assume p: p ∈ R
  with r have 1: p = ⟨p[1], p[2]⟩ (is p = ?pp) by (intro prodProj, auto)
  from p r have p[1] ∈ B by (auto dest: pairProj-in-prod)
  with 1 p r have ?pp ∈ R ∘ Id(B) by (intro compI, auto)
  with 1 show p ∈ R ∘ Id(B) by simp
qed

lemma Id-comp-R [simp]:
  assumes r: R ⊆ A × B
  shows Id(B) ∘ R = R
using r proof auto
  fix p
  assume p: p ∈ R
  with r have 1: p = ⟨p[1], p[2]⟩ (is p = ?pp) by (intro prodProj, auto)
  from p r have p[2] ∈ B by (auto dest: pairProj-in-prod)
  with 1 p r have ?pp ∈ Id(B) ∘ R by (intro compI, auto)
  with 1 show p ∈ Id(B) ∘ R by simp
qed

lemma rel-comp-empty1 [simp]: {} ∘ R = {}
unfolding rel-comp-def by auto

lemma rel-comp-empty2 [simp]: R ∘ {} = {}
unfolding rel-comp-def by auto

lemma comp-assoc:
  assumes t: T ⊆ A × B and s: S ⊆ B × C and r: R ⊆ C × D
  shows (R ∘ S) ∘ T = R ∘ (S ∘ T)
```

```
proof
  fix p
  assume p: p ∈ (R ∘ S) ∘ T
  from r s have R ∘ S ⊆ B × D by simp
  from p this t show p ∈ R ∘ (S ∘ T)
  proof
    fix x y z
    assume 1: p = ⟨x,z⟩ and 2: ⟨x, y⟩ ∈ T and 3: ⟨y, z⟩ ∈ R ∘ S
    from 3 r s show ?thesis
    proof (rule compEpair)
      fix u
      assume ⟨y,u⟩ ∈ S and ⟨u,z⟩ ∈ R
      with r s t 2 have ⟨x,z⟩ ∈ R ∘ (S ∘ T)
        by (intro compI, auto elim!: relProj1 relProj2)
      with 1 show ?thesis by simp
    qed
  qed
next
  fix p
  assume p: p ∈ R ∘ (S ∘ T)
  from s t have S ∘ T ⊆ A × C by simp
  from p r this show p ∈ (R ∘ S) ∘ T
  proof
    fix x y z
    assume 1: p = ⟨x,z⟩ and 2: ⟨x, y⟩ ∈ S ∘ T and 3: ⟨y, z⟩ ∈ R
    from 2 s t show ?thesis
    proof (rule compEpair)
      fix u
      assume ⟨x,u⟩ ∈ T and ⟨u,y⟩ ∈ S
      with r s t 3 have ⟨x,z⟩ ∈ (R ∘ S) ∘ T
        by (intro compI, auto elim!: relProj1 relProj2)
      with 1 show ?thesis by simp
    qed
  qed
qed

lemma rel-comp-mono:
  assumes hr': r' ⊆ r and hs': s' ⊆ s
  shows (r' ∘ s') ⊆ (r ∘ s)
unfolding rel-comp-def using subrel-dom[OF hs'] subrel-ran[OF hr']
proof auto
  fix x y z
  assume xy': ⟨x, y⟩ ∈ s' and yz': ⟨y, z⟩ ∈ r'
  from hs' xy' have xy: ⟨x,y⟩ ∈ s by auto
  from hr' yz' have yz: ⟨y,z⟩ ∈ r by auto
  show ∃ y : ⟨x, y⟩ ∈ s ∧ ⟨y, z⟩ ∈ r
    using xy yz by auto
qed
```

lemma *rel-comp-distrib* [*simp*]: $R \circ (S \cup T) = (R \circ S) \cup (R \circ T)$
unfolding *rel-comp-def* **proof** *auto*

fix $x\ y\ z$
assume *xy*: $\langle x,y\rangle \in T$ **and** *yz*: $\langle y,z\rangle \in R$
and *1*: $\forall yy : \langle x,\ yy\rangle \in T = FALSE \vee \langle yy,\ z\rangle \in R = FALSE$
from *1* **have** $\langle x,\ y\rangle \in T = FALSE \vee \langle y,\ z\rangle \in R = FALSE$ **..**
with *xy yz* **show** $\exists y : \langle x,y\rangle \in S \wedge \langle y,z\rangle \in R$ **by** *simp*
qed

lemma *rel-comp-distrib2* [*simp*]: $(S \cup T) \circ R = (S \circ R) \cup (T \circ R)$
unfolding *rel-comp-def* **proof** *auto*

fix $x\ y\ z$
assume *xy*: $\langle x,y\rangle \in R$ **and** *yz*: $\langle y,z\rangle \in T$
and *1*: $\forall yy : \langle x,\ yy\rangle \in R = FALSE \vee \langle yy,\ z\rangle \in T = FALSE$
from *1* **have** $\langle x,\ y\rangle \in R = FALSE \vee \langle y,\ z\rangle \in T = FALSE$ **..**
with *xy yz* **show** $\exists y : \langle x,y\rangle \in R \wedge \langle y,z\rangle \in S$ **by** *simp*
qed

8.7.5 Properties of relations

Reflexivity

lemma *reflI* [*intro*]: $(\bigwedge x.\ x \in A \Longrightarrow \langle x,x\rangle \in r) \Longrightarrow reflexive(A,r)$
unfolding *reflexive-def* **by** *blast*

lemma *reflexiveD* [*elim!*]: $reflexive(A,r) \Longrightarrow a \in A \Longrightarrow \langle a,a\rangle \in r$
unfolding *reflexive-def* **by** *blast*

lemma *reflexive-empty* : $reflexive(\{\},\ \{\})$
by *auto*

Symmetry

lemma *symmetricI*: $[\![\ \bigwedge x\ y.\ \langle x,y\rangle \in r \Longrightarrow \langle y,x\rangle \in r\ ]\!] \Longrightarrow symmetric(r)$
unfolding *symmetric-def* **by** *blast*

lemma *symmetricE*: $[\![\ symmetric(r);\ \langle x,y\rangle \in r\ ]\!] \Longrightarrow \langle y,x\rangle \in r$
unfolding *symmetric-def* **by** *blast*

lemma *symmetric-Int*: $[\![\ symmetric(r);\ symmetric(s)\ ]\!] \Longrightarrow symmetric(r \cap s)$
by (*blast intro*: *symmetricI dest*: *symmetricE*)

Antisymmetry

lemma *antisymmetricI* [*intro*]:
$[\![\ \bigwedge x\ y.\ [\![\ \langle x,y\rangle \in r;\ \langle y,x\rangle \in r\ ]\!] \Longrightarrow x = y\ ]\!] \Longrightarrow antisymmetric(r)$
unfolding *antisymmetric-def* **by** *blast*

lemma *antisymmetricE* [*elim*]: $[\![\ antisymmetric(r);\ \langle x,y\rangle \in r;\ \langle y,x\rangle \in r\ ]\!] \Longrightarrow x = y$

unfolding *antisymmetric-def* **by** *blast*

lemma *antisymmetricSubset*: $r \subseteq s \Longrightarrow antisymmetric(s) \Longrightarrow antisymmetric(r)$
unfolding *antisymmetric-def* **by** *blast*

lemma *antisym-empty* : $antisymmetric(\{\})$
by *blast*

Transitivity

lemma *transitiveI* [*intro*]:
$(\bigwedge x\ y\ z.\ \langle x,y\rangle \in r \Longrightarrow \langle y,z\rangle \in r \Longrightarrow \langle x,z\rangle \in r) \Longrightarrow transitive(r)$
unfolding *transitive-def* **by** *blast*

lemma *transD* [*elim*]: $[\![\ transitive(r);\ \ \langle x,y\rangle \in r;\ \langle y,z\rangle \in r\ ]\!] \Longrightarrow \langle x,z\rangle \in r$
unfolding *transitive-def* **by** *blast*

lemma *trans-Int*: $transitive(r) \Longrightarrow transitive(s) \Longrightarrow transitive(r \cap s)$
by *fast*

lemma *transitive-iff-comp-subset*: $transitive(r) = (r \circ r \subseteq r)$
unfolding *transitive-def rel-comp-def* **by** (*auto elim*!: *subsetD*)

Irreflexivity

lemma *irreflexiveI* [*intro*]: $[\![\ \bigwedge x.\ x \in A \Longrightarrow \langle x,x\rangle \notin r\ ]\!] \Longrightarrow irreflexive(A,r)$
unfolding *irreflexive-def* **by** *blast*

lemma *irreflexiveE* [*dest*]: $[\![\ irreflexive(A,r);\ x \in A\ ]\!] \Longrightarrow\ \langle x,x\rangle \notin r$
unfolding *irreflexive-def* **by** *blast*

8.7.6 Equivalence Relations

r is an equivalence relation iff $r\hat{\ }{-1} \circ r = r$

First half: "only if" part

lemma *sym-trans-comp-subset*:
assumes $r \subseteq A \times A$ **and** *symmetric*(r) **and** *transitive*(r)
shows $r\hat{\ }{-1} \circ r \subseteq r$
using *assms* **by** (*simp add*: *symmetric-iff-converse-eq transitive-iff-comp-subset*)

lemma *refl-comp-subset*:
assumes *r*: $r \subseteq A \times A$ **and** *refl*: *reflexive*(A,r)
shows $r \subseteq r\hat{\ }{-1} \circ r$
proof
fix *p*
assume *p*: $p \in r$
with *r* **obtain** *x z* **where** *1*: $p = \langle x,z\rangle$ **by** (*blast dest*: *prodProj*)
with *p r* **have** $z \in A$ **by** *auto*
with *refl* **have** $\langle z,z\rangle \in r$ **by** *auto*
moreover

from *1 p* **have** $\langle z,x\rangle \in r\hat{}-1$ **by** *auto*
moreover
from *r* **have** $r\hat{}-1 \subseteq A \times A$ **by** (*rule converseSubset*)
moreover
note *r*
ultimately
have $\langle x,z\rangle \in r\hat{}-1 \circ r$ **by** (*intro compI*, *auto*)
with *1* **show** $p \in r\hat{}-1 \circ r$ **by** *simp*
qed

lemma *equiv-comp-eq*:
assumes *r*: $r \subseteq A \times A$ **and** *eq*: *equivalence*(*A*,*r*)
shows $r\hat{}-1 \circ r = r$
using *eq sym-trans-comp-subset*[*OF r*] *refl-comp-subset*[*OF r*]
unfolding *equivalence-def*
by (*intro setEqual*, *simp+*)

Second half: "if" part, needs totality of relation *r*

lemma *comp-equivI*:
assumes *dom*: *rel-domain*(*r*) = *A* **and** *r*: $r \subseteq A \times A$ **and** *comp*: $r\hat{}-1 \circ r = r$
shows *equivalence*(*A*,*r*)
proof −
from *r* **have** *r1*: $r\hat{}-1 \subseteq A \times A$ **by** (*rule converseSubset*)
have *refl*: *reflexive*(*A*,*r*)
proof
fix *a*
assume *a*: $a \in A$
with *dom r* **obtain** *b* **where** *b*: $\langle a,b\rangle \in r$ **by** *auto*
hence $\langle b,a\rangle \in r\hat{}-1$ **..**
with *b r r1* **have** $\langle a,a\rangle \in r\hat{}-1 \circ r$ **by** (*intro compI*, *auto*)
with *comp* **show** $\langle a,a\rangle \in r$ **by** *simp*
qed
have *sym*: *symmetric*(*r*)
proof −
from *comp* **have** $r\hat{}-1 = (r\hat{}-1 \circ r)\hat{}-1$ **by** *simp*
also from *r r1* **have** ... $= r\hat{}-1 \circ r$ **by** (*simp add*: *converse-comp*)
finally have $r\hat{}-1 = r$ **by** (*simp add*: *comp*)
with *r* **show** *?thesis* **by** (*simp add*: *symmetric-iff-converse-eq*)
qed
have *trans*: *transitive*(*r*)
proof −
from *r sym* **have** $r \circ r = r\hat{}-1 \circ r$ **by** (*simp add*: *symmetric-iff-converse-eq*)
with *comp* **have** $r \circ r = r$ **by** *simp*
thus *?thesis* **by** (*simp add*: *transitive-iff-comp-subset*)
qed
from *refl sym trans* **show** *?thesis*
unfolding *equivalence-def* **by** *simp*
qed

end

9 The division operators div and mod on Naturals

theory *NatDivision*
imports *NatArith Tuples*
begin

9.1 The divisibility relation

definition *dvd* (**infixl** *dvd 50*)
where $a \in Nat \Longrightarrow b \in Nat \Longrightarrow b\ dvd\ a \equiv (\exists\, k \in Nat : a = b * k)$

lemma *boolify-dvd* [*simp*]:
  assumes $a \in Nat$ **and** $b \in Nat$
  shows $boolify(b\ dvd\ a) = (b\ dvd\ a)$
using *assms* **by** (*simp add*: *dvd-def*)

lemma *dvdIsBool* [*intro*!,*simp*]:
  assumes *a*: $a \in Nat$ **and** *b*: $b \in Nat$
  shows $isBool(b\ dvd\ a)$
using *assms* **by** (*simp add*: *dvd-def*)

lemma [*intro*!]:
  $[\![isBool(P);\ isBool(a\ dvd\ b);\ (a\ dvd\ b) \Leftrightarrow P]\!] \Longrightarrow (a\ dvd\ b) = P$
  $[\![isBool(P);\ isBool(a\ dvd\ b);\ P \Leftrightarrow (a\ dvd\ b)]\!] \Longrightarrow P = (a\ dvd\ b)$
by *auto*

lemma *dvdI* [*intro*]:
  assumes *a*: $a \in Nat$ **and** *b*: $b \in Nat$ **and** *k*: $k \in Nat$
  shows $a = b * k \Longrightarrow b\ dvd\ a$
unfolding *dvd-def*[*OF a b*] **using** *k* **by** *blast*

lemma *dvdE* [*elim*]:
  assumes $b\ dvd\ a$ **and** $a \in Nat$ **and** $b \in Nat$
  shows $(\bigwedge k.\ [\![k \in Nat;\ a = b * k]\!] \Longrightarrow P) \Longrightarrow P$
using *assms* **by** (*auto simp add*: *dvd-def*)

lemma *dvd-refl* [*iff*]:
  assumes *a*: $a \in Nat$
  shows $a\ dvd\ a$
proof −
  from *a* **have** $a = a*1$ **by** *simp*
  with *a* **show** *?thesis* **by** *blast*
qed

lemma *dvd-trans* [*trans*]:
  assumes *a*: $a \in Nat$ **and** *b*: $b \in Nat$ **and** *c*: $c \in Nat$

and *1*: *a dvd b* **and** *2*: *b dvd c*
shows *a dvd c*
proof −
from *a b 1* **obtain** *k* **where** *k*: *k* ∈ *Nat* **and** *b* = *a* ∗ *k* **by** *blast*
moreover
from *b c 2* **obtain** *l* **where** *l*: *l* ∈ *Nat* **and** *c* = *b* ∗ *l* **by** *blast*
ultimately have *h*:*c* = *a* ∗ (*k* ∗ *l*)
using *a b c* **by** (*simp add*: *mult-assoc-nat*)
thus *?thesis* **using** *a c k l* **by** *blast*
qed

lemma *dvd-0-left-iff* [*simp*]:
assumes *a* ∈ *Nat*
shows (*0 dvd a*) = (*a* = *0*)
using *assms* **by** *force*

lemma *dvd-0-right* [*iff*]:
assumes *a*: *a* ∈ *Nat* **shows** *a dvd 0*
using *assms* **by** *force*

lemma *one-dvd* [*iff*]:
assumes *a*: *a* ∈ *Nat*
shows *1 dvd a*
using *assms* **by** *force*

lemma *dvd-mult* :
assumes *dvd*: *a dvd c* **and** *a*: *a* ∈ *Nat* **and** *b*: *b* ∈ *Nat* **and** *c*: *c* ∈ *Nat*
shows *a dvd* (*b* ∗ *c*)
proof −
from *dvd a c* **obtain** *k* **where** *k*: *k* ∈ *Nat* **and** *c* = *a*∗*k* **by** *blast*
with *a b c* **have** *b*∗*c* = *a* ∗ (*b*∗*k*) **by** (*simp add*: *mult-left-commute-nat*)
with *a b c k* **show** *?thesis* **by** *blast*
qed

lemma *dvd-mult2* :
assumes *dvd*: *a dvd b* **and** *a*: *a* ∈ *Nat* **and** *b*: *b* ∈ *Nat* **and** *c*: *c* ∈ *Nat*
shows *a dvd* (*b* ∗ *c*)
using *mult-commute-nat*[*OF b c*] *dvd-mult*[*OF dvd a c b*] **by** *simp*

lemma *dvd-triv-right* [*iff*]:
assumes *a*: *a* ∈ *Nat* **and** *b*: *b* ∈ *Nat*
shows *a dvd* (*b* ∗ *a*)
using *assms* **by** (*intro dvd-mult*, *simp*+)

lemma *dvd-triv-left* [*iff*]:
assumes *a*: *a* ∈ *Nat* **and** *b*: *b* ∈ *Nat*
shows *a dvd a* ∗ *b*
using *assms* **by** (*intro dvd-mult2*, *simp*+)

lemma *mult-dvd-mono*:
  assumes *a*: *a* ∈ *Nat* **and** *b*: *b* ∈ *Nat* **and** *c*: *c* ∈ *Nat* **and** *d*: *d* ∈ *Nat*
    and *1*: *a dvd b* **and** *2*: *c dvd d*
  shows $(a * c)$ *dvd* $(b * d)$
proof −
  from *a b 1* **obtain** b' **where** b': $b = a * b'$ $b' \in Nat$ **by** *blast*
  from *c d 2* **obtain** d' **where** d': $d = c * d'$ $d' \in Nat$ **by** *blast*
  with b' *a b c d*
    mult-assoc-nat[*of a* b' $c * d'$]
    mult-left-commute-nat[*of* b' *c* d']
    mult-assoc-nat[*of a c* $b' * d'$]
  have $b * d = (a * c) * (b' * d')$ **by** *simp*
  with *a c* b' d' **show** *?thesis* **by** *blast*
qed

lemma *dvd-mult-left*:
  assumes *a*: *a* ∈ *Nat* **and** *b*: *b* ∈ *Nat* **and** *c*: *c* ∈ *Nat*
    and *h*: $a * b$ *dvd c*
  shows *a dvd c*
proof −
  from *h a b c* **obtain** *k* **where** *k*: $k \in Nat$ $c = a*(b*k)$
    by (*auto simp add*: *mult-assoc-nat*)
  with *a b c* **show** *?thesis* **by** *blast*
qed

lemma *dvd-mult-right*:
  assumes *a*: *a* ∈ *Nat* **and** *b*: *b* ∈ *Nat* **and** *c*: *c* ∈ *Nat* **and** *h*: $a*b$ *dvd c*
  shows *b dvd c*
proof −
  from *h a b c* **have** $b*a$ *dvd c* **by** (*simp add*: *mult-ac-nat*)
  with *b a c* **show** *?thesis* **by** (*rule dvd-mult-left*)
qed

lemma *dvd-0-left*:
  assumes *a* ∈ *Nat*
  shows *0 dvd a* ⟹ *a* = *0*
using *assms* **by** *simp*

lemma *dvd-add* [*iff*]:
  assumes *a*: *a* ∈ *Nat* **and** *b*: *b* ∈ *Nat* **and** *c*: *c* ∈ *Nat*
    and *1*: *a dvd b* **and** *2*: *a dvd c*
  shows *a dvd* $(b + c)$
proof −
  from *a b 1* **obtain** b' **where** b': $b' \in Nat$ $b = a * b'$ **by** *blast*
  from *a c 2* **obtain** c' **where** c': $c' \in Nat$ $c = a * c'$ **by** *blast*
  from *a b c* b' c'
  have $b + c = a * (b' + c')$ **by** (*simp add*: *add-mult-distrib-left-nat*)
  with *a* b' c' **show** *?thesis* **by** *blast*
qed

9.2 Division on *Nat*

We define division and modulo over *Nat* by means of a characteristic relation with two input arguments m, n and two output arguments q(uotient) and r(emainder).

The following definition works for natural numbers, but also for possibly negative integers. Obviously, the second disjunct cannot hold for natural numbers.

definition *divmod-rel* **where**
  $\textit{divmod-rel}(m,n,q,r) \equiv m = q * n + r$
    $\wedge ((0 < n \wedge 0 \leq r \wedge r < n) \vee (n < 0 \wedge r \leq 0 \wedge n < r))$

divmod-rel is total if n is non-zero.

lemma *divmod-rel-ex*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: $0 < n$
  obtains $q\ r$ **where** $q \in Nat$ $r \in Nat$ $\textit{divmod-rel}(m,n,q,r)$
proof −
  have $\exists\, q,r \in Nat : m = q{*}n + r \wedge r < n$
  using *m* **proof** (*induct*)
    case *0*
    from *n pos* **have** $0 = 0 * n + 0 \wedge 0 < n$ **by** *simp*
    then show *?case* **by** *blast*
  next
    fix m'
    assume *m'*: $m' \in Nat$ **and** *ih*: $\exists\, q,\ r \in Nat : m' = q{*}n + r \wedge r < n$
    from *ih* **obtain** $q'\ r'$
      where *h1*: $m' = q' * n + r'$ **and** *h2*: $r' < n$
      and *q'*: $q' \in Nat$ **and** *r'*: $r' \in Nat$ **by** *blast*
    show $\exists\, q,\ r \in Nat : Succ[m'] = q * n + r \wedge r < n$
    proof (*cases* $Succ[r'] < n$)
      case *True*
      from *h1 h2 m' q' n r'* **have** $Succ[m'] = q' * n + Succ[r']$ **by** *simp*
      with *True q' r'* **show** *?thesis* **by** *blast*
    next
      case *False*
      with *n r'* **have** $n \leq Succ[r']$ **by** (*simp add*: *nat-not-less*[*simplified*])
      with *r' n h2* **have** $n = Succ[r']$ **by** (*intro nat-leq-antisym*, *simp+*)
      with *h1 m' q' r'* **have** $Succ[m'] = Succ[q'] * n + 0$ **by** (*simp add*: *add-ac-nat*)
      with *pos q'* **show** *?thesis* **by** *blast*
    qed
  qed
  with *pos that* **show** *?thesis* **by** (*auto simp*: *divmod-rel-def*)
qed

divmod-rel has unique solutions in the natural numbers.

lemma *divmod-rel-unique-div*:
  assumes *1*: $\textit{divmod-rel}(m,n,q,r)$ **and** *2*: $\textit{divmod-rel}(m,n,q',r')$
    and *m*: $m \in Nat$ **and** *n*: $n \in Nat$

and *q*: $q \in Nat$ **and** *r*: $r \in Nat$ **and** *q'*: $q' \in Nat$ **and** *r'*: $r' \in Nat$
shows $q = q'$
proof −
from *n 1* **have** *pos*: $0 < n$ **and** *mqr*: $m = q*n+r$ **and** *rn*: $r < n$
by (*auto simp*: *divmod-rel-def*)
from *n 2* **have** *mqr'*: $m = q'*n+r'$ **and** *rn'*: $r' < n$
by (*auto simp*: *divmod-rel-def*)
{
fix *x y x' y'*
assume *nat*: $x \in Nat$ $y \in Nat$ $x' \in Nat$ $y' \in Nat$
and *eq*: $x*n + y = x'*n + y'$ **and** *less*: $y' < n$
have $x \leq x'$
proof (*rule contradiction*)
assume $\neg(x \leq x')$
with *nat* **have** $x' < x$ **by** (*simp add*: *nat-not-leq*[*simplified*])
with *nat* **obtain** *k* **where** *k*: $k \in Nat$ $x = Succ[x'+k]$
by (*auto simp*: *less-iff-Succ-add*)
with *eq nat n* **have** $x'*n + (k*n + n + y) = x'*n + y'$
by (*simp add*: *add-mult-distrib-right-nat add-assoc-nat*)
with *nat k n* **have** $k*n + n + y = y'$ **by** *simp*
with *less k n nat* **have** $(k*n + y) + n < n$ **by** (*simp add*: *add-ac-nat*)
with *k n nat* **show** *FALSE* **by** *simp*
qed
}
from *this*[*OF q r q' r'*] *this*[*OF q' r' q r*] *q q' mqr mqr' rn rn'*
show *?thesis* **by** (*intro nat-leq-antisym*, *simp+*)
qed

lemma *divmod-rel-unique-mod*:
assumes *divmod-rel*(m,n,q,r) **and** *divmod-rel*(m,n,q',r')
and $m \in Nat$ **and** $n \in Nat$ **and** $q \in Nat$ **and** $r \in Nat$ **and** $q' \in Nat$ **and** r'
$\in Nat$
shows $r = r'$
proof −
from *assms* **have** $q = q'$ **by** (*rule divmod-rel-unique-div*)
with *assms* **show** *?thesis* **by** (*auto simp*: *divmod-rel-def*)
qed

We instantiate divisibility on the natural numbers by means of *divmod-rel*:

definition *divmodNat*
where *divmodNat*$(m,n) \equiv$ *CHOOSE* $z \in Nat \times Nat$: *divmod-rel*$(m,n,z[1],z[2])$

lemma *divmodNatPairEx*:
assumes $m \in Nat$ **and** $n \in Nat$ **and** $0 < n$
shows $\exists\, z \in Nat \times Nat$: *divmod-rel*$(m,n,z[1],z[2])$
proof −
from *assms* **obtain** *q r* **where** $q \in Nat$ $r \in Nat$ *divmod-rel*(m,n,q,r)
by (*rule divmod-rel-ex*)
thus *?thesis* **by** *force*

qed

lemma *divmodNatInNatNat*:
assumes *m*: *m* ∈ *Nat* **and** *n*: *n* ∈ *Nat* **and** *pos*: *0* < *n*
shows *divmodNat*(*m*,*n*) ∈ *Nat* × *Nat*
unfolding *divmodNat-def* **by** (*rule bChooseI2*[*OF divmodNatPairEx*[*OF assms*]])

lemma *divmodNat-divmod-rel* [*rule-format*]:
assumes *m*: *m* ∈ *Nat* **and** *n*: *n* ∈ *Nat* **and** *pos*: *0* < *n*
shows *z* = *divmodNat*(*m*,*n*) ⇒ *divmod-rel*(*m*,*n*,*z*[*1*],*z*[*2*])
unfolding *divmodNat-def* **by** (*rule bChooseI2*[*OF divmodNatPairEx*[*OF assms*]], *auto*)

lemma *divmodNat-unique*:
assumes *h*: *divmod-rel*(*m*,*n*,*q*,*r*)
and *m*: *m* ∈ *Nat* **and** *n*: *n* ∈ *Nat* **and** *pos*: *0* < *n*
and *q*: *q* ∈ *Nat* **and** *r*: *r* ∈ *Nat*
shows *divmodNat*(*m*,*n*) = ⟨*q*,*r*⟩
unfolding *divmodNat-def*
proof (*rule bChooseI2*[*OF divmodNatPairEx*[*OF m n pos*]])
fix *z*
assume *z* ∈ *Nat* × *Nat* **and** *divmod-rel*(*m*,*n*,*z*[*1*],*z*[*2*])
with *m n q r h* **show** *z* = ⟨*q*,*r*⟩
by (*auto elim*!: *inProdE elim*: *divmod-rel-unique-div divmod-rel-unique-mod*)
qed

We now define division and modulus over natural numbers.

definition *div* (**infixr** *div 70*)
where *div-nat-def*: ⟦*m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *m div n* ≡ *divmodNat*(*m*,*n*)[*1*]

definition *mod* (**infixr** *mod 70*)
where *mod-nat-def*: ⟦*m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *m mod n* ≡ *divmodNat*(*m*,*n*)[*2*]

lemma *divIsNat* [*iff*]:
assumes *m* ∈ *Nat* **and** *n* ∈ *Nat* **and** *0* < *n*
shows *m div n* ∈ *Nat*
using *divmodNatInNatNat*[*OF assms*] *assms* **by** (*auto simp*: *div-nat-def*)

lemma *modIsNat* [*iff*]:
assumes *m* ∈ *Nat* **and** *n* ∈ *Nat* **and** *0* < *n*
shows *m mod n* ∈ *Nat*
using *divmodNatInNatNat*[*OF assms*] *assms* **by** (*auto simp*: *mod-nat-def*)

lemma *divmodNat-div-mod*:
assumes *m*: *m* ∈ *Nat* **and** *n*: *n* ∈ *Nat* **and** *pos*: *0* < *n*
shows *divmodNat*(*m*,*n*) = ⟨*m div n*, *m mod n*⟩
unfolding *div-nat-def*[*OF m n*] *mod-nat-def*[*OF m n*] **using** *divmodNatInNatNat*[*OF assms*]

by *force*

lemma *divmod-rel-div-mod-nat*:
  assumes $m \in Nat$ **and** $n \in Nat$ **and** $0 < n$
  shows *divmod-rel*(m, n, m *div* n, m *mod* n)
using *divmodNat-divmod-rel*[*OF assms sym*[*OF divmodNat-div-mod*[*OF assms*]]]
by *simp*

lemma *div-nat-unique*:
  assumes *h*: *divmod-rel*(m,n,q,r)
    and *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: $0 < n$ **and** *q*: $q \in Nat$ **and** *r*: $r \in Nat$
  shows m *div* $n = q$
unfolding *div-nat-def*[*OF m n*] **using** *divmodNat-unique*[*OF assms*] **by** *simp*

lemma *mod-nat-unique*:
  assumes *h*: *divmod-rel*(m,n,q,r)
    and *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: $0 < n$ **and** *q*: $q \in Nat$ **and** *r*: $r \in Nat$
  shows m *mod* $n = r$
unfolding *mod-nat-def*[*OF m n*] **using** *divmodNat-unique*[*OF assms*] **by** *simp*

lemma *mod-nat-less-divisor*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: $0 < n$
  shows m *mod* $n < n$
using *assms divmod-rel-div-mod-nat*[*OF assms*] **by** (*simp add*: *divmod-rel-def*)

"Recursive" computation of *div* and *mod*.

lemma *divmodNat-base*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *less*: $m < n$
  shows *divmodNat*(m,n) = ⟨*0*, m⟩
proof −
  from *assms* **have** *pos*: $0 < n$ **by** (*intro nat-leq-less-trans*[*of 0 m n*], *simp+*)
  let *?dm* = *divmodNat*(m,n)
  from *m n pos* **have** *1*: *divmod-rel*(m,n,*?dm*[*1*],*?dm*[*2*])
    by (*simp add*: *divmodNat-divmod-rel*)
  from *m n pos* **have** *2*: *?dm* ∈ *Nat* × *Nat* **by** (*rule divmodNatInNatNat*)
   with *1 2 less n* **have** *?dm*[*1*] ∗ $n < n$ **by** (*auto simp*: *divmod-rel-def elim*!: *add-lessD1*)
  with *2 n* **have** *3*: *?dm*[*1*] = *0* **by** (*intro mult-less-self-right*, *auto*)

  with *1 2 m n* **have** *?dm*[*2*] = m **by** (*auto simp*: *divmod-rel-def*)

  with *3 prodProj*[*OF 2*] **show** *?thesis* **by** *simp*
qed

lemma *divmodNat-step*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: $0 < n$ **and** *geq*: $n \leq m$
  shows *divmodNat*(m,n) = ⟨*Succ*[(m −− n) *div* n], (m −− n) *mod* n⟩

```
proof −
  from m n pos have 1: divmod-rel(m, n, m div n, m mod n)
    by (rule divmod-rel-div-mod-nat)
  have 2: m div n ≠ 0
  proof
    assume m div n = 0
    with 1 m n pos have m < n by (auto simp: divmod-rel-def)
    with geq m n show FALSE by (auto simp: nat-less-leq-not-leq)
  qed
  with m n pos obtain k where k1: k ∈ Nat and k2: m div n = Succ[k]
    using not0-implies-Suc[of m div n] by auto
  with 1 m n pos have m = n + k∗n + m mod n
    by (auto simp: divmod-rel-def add-commute-nat)
  moreover
  from m n k1 pos geq have ... −− n = k∗n + m mod n
    by (simp add: adiff-add-assoc2)
  ultimately
  have m −− n = k∗n + m mod n by simp
  with pos m n 1 have divmod-rel(m −− n, n, k, m mod n)
    by (auto simp: divmod-rel-def)

  with k1 m n pos have divmodNat(m −− n, n) = ⟨k, m mod n⟩
    by (intro divmodNat-unique, simp+)
  moreover
  from m n pos have divmodNat(m −− n, n) = ⟨(m−−n) div n, (m−−n) mod
n⟩
    by (intro divmodNat-div-mod, simp+)
  ultimately
  have m div n = Succ[(m−−n) div n] and m mod n = (m−−n) mod n
    using m n k2 by auto
  thus ?thesis by (simp add: divmodNat-div-mod[OF m n pos])
qed
```

The "recursion" equations for *div* and *mod*

```
lemma div-nat-less [simp]:
  assumes m: m ∈ Nat and n: n ∈ Nat and less: m < n
  shows m div n = 0
proof −
  from assms have pos: 0 < n by (intro nat-leq-less-trans[of 0 m n], simp+)
   from divmodNat-base[OF m n less] divmodNat-div-mod[OF m n pos] show
?thesis
    by simp
qed

lemma div-nat-geq:
  assumes m: m ∈ Nat and n: n ∈ Nat and pos: 0 < n and geq: n ≤ m
  shows m div n = Succ[(m −− n) div n]
using divmodNat-step[OF assms] divmodNat-div-mod[OF m n pos]
by simp
```

lemma *mod-nat-less* [*simp*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *less*: $m < n$
shows *m mod n* = *m*
proof −
from *assms* **have** *pos*: *0* < *n* **by** (*intro nat-leq-less-trans*[*of 0 m n*], *simp*+)
from *divmodNat-base*[*OF m n less*] *divmodNat-div-mod*[*OF m n pos*] **show** *?thesis*
by *simp*
qed

lemma *mod-nat-geq*:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: *0* < *n* **and** *geq*: $n \leq m$
shows *m mod n* = (*m* −− *n*) *mod n*
using *divmodNat-step*[*OF assms*] *divmodNat-div-mod*[*OF m n pos*]
by *simp*

9.3 Facts about *op div* and *op mod*

lemma *mod-div-nat-equality* [*simp*]:
assumes $m \in Nat$ **and** $n \in Nat$ **and** *0* < *n*
shows (*m div n*) ∗ *n* + *m mod n* = *m*
using *divmod-rel-div-mod-nat* [*OF assms*] **by** (*simp add*: *divmod-rel-def*)

lemma *mod-div-nat-equality2* [*simp*]:
assumes $m \in Nat$ **and** $n \in Nat$ **and** *0* < *n*
shows *n* ∗ (*m div n*) + *m mod n* = *m*
using *assms mult-commute-nat*[*of n m div n*] **by** *simp*

lemma *mod-div-nat-equality3* [*simp*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *0* < *n*
shows *m mod n* + (*m div n*) ∗ *n* = *m*
using *assms add-commute-nat*[*of m mod n*] **by** *simp*

lemma *mod-div-nat-equality4* [*simp*]:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *0* < *n*
shows *m mod n* + *n* ∗ (*m div n*) = *m*
using *assms mult-commute-nat*[*of n m div n*] **by** *simp*

lemma *div-nat-mult-self1* [*simp*]:
assumes *q*: $q \in Nat$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: *0* < *n*
shows (*q* + *m* ∗ *n*) *div n* = *m* + (*q div n*) (**is** *?P*(*m*))
using *m* **proof** (*induct m*)
from *assms* **show** *?P*(*0*) **by** *simp*
next
fix *k*
assume *k*: $k \in Nat$ **and** *ih*: *?P*(*k*)

from *n q k* **have** $n \leq q + (k*n + n)$ **by** (*simp add*: *add-assoc-nat*)
with *q k n pos* **have** $(q + (k*n + n))$ *div* $n = Succ[(q + k*n)$ *div* $n]$
by (*simp add*: *div-nat-geq add-assoc-nat*)
with *ih q m n k pos* **show** *?P*(*Succ*[*k*]) **by** *simp*
qed

lemma *div-nat-mult-self2* [*simp*]:
assumes $q \in Nat$ **and** $n \in Nat$ **and** $m \in Nat$ **and** $0 < n$
shows $(q + n * m)$ *div* $n = m + q$ *div* n
using *assms* **by** (*simp add*: *mult-commute-nat*)

lemma *div-nat-mult-self3* [*simp*]:
assumes $q \in Nat$ **and** $n \in Nat$ **and** $m \in Nat$ **and** $0 < n$
shows $(m * n + q)$ *div* $n = m + q$ *div* n
using *assms* **by** (*simp add*: *add-commute-nat*)

lemma *div-nat-mult-self4* [*simp*]:
assumes $q \in Nat$ **and** $n \in Nat$ **and** $m \in Nat$ **and** $0 < n$
shows $(n * m + q)$ *div* $n = m + q$ *div* n
using *assms* **by** (*simp add*: *add-commute-nat*)

lemma *div-nat-0*:
assumes $n \in Nat$ **and** $0 < n$
shows *0 div* $n = 0$
using *assms* **by** *simp*

lemma *mod-0*:
assumes $n \in Nat$ **and** $0 < n$
shows *0 mod* $n = 0$
using *assms* **by** *simp*

lemma *mod-nat-mult-self1* [*simp*]:
assumes *q*: $q \in Nat$ **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: $0 < n$
shows $(q + m * n)$ *mod* $n = q$ *mod* n
proof −
from *assms* **have** $m*n + q = q + m*n$
by (*simp add*: *add-commute-nat*)
also from *assms* **have** ... $= ((q + m*n)$ *div* $n) * n + (q + m*n)$ *mod* n
by (*intro sym*[*OF mod-div-nat-equality*], *simp+*)
also from *assms* **have** ... $= (m + q$ *div* $n) * n + (q + m*n)$ *mod* n
by *simp*
also from *assms* **have** ... $= m*n + ((q$ *div* $n) * n + (q + m*n)$ *mod* $n)$
by (*simp add*: *add-mult-distrib-right-nat add-assoc-nat*)
finally have $q = (q$ *div* $n) * n + (q + m*n)$ *mod* n
using *assms* **by** *simp*
with *q n pos* **have** $(q$ *div* $n) * n + (q + m*n)$ *mod* $n = (q$ *div* $n) * n + q$ *mod* n
by *simp*
with *assms* **show** *?thesis* **by** (*simp del*: *mod-div-nat-equality*)
qed

lemma *mod-nat-mult-self2* [*simp*]:
assumes $q \in Nat$ **and** $m \in Nat$ **and** $n \in Nat$ **and** $0 < n$
shows $(q + n * m)$ *mod* $n = q$ *mod* n
using *assms* **by** (*simp add*: *mult-commute-nat*)

lemma *mod-nat-mult-self3* [*simp*]:
assumes $q \in Nat$ **and** $m \in Nat$ **and** $n \in Nat$ **and** $0 < n$
shows $(m * n + q)$ *mod* $n = q$ *mod* n
using *assms* **by** (*simp add*: *add-commute-nat*)

lemma *mod-nat-mult-self4* [*simp*]:
assumes $q \in Nat$ **and** $m \in Nat$ **and** $n \in Nat$ **and** $0 < n$
shows $(n * m + q)$ *mod* $n = q$ *mod* n
using *assms* **by** (*simp add*: *add-commute-nat*)

lemma *div-nat-mult-self1-is-id* [*simp*]:
assumes $m \in Nat$ **and** $n \in Nat$ **and** $0 < n$
shows $(m * n)$ *div* $n = m$
using *assms div-nat-mult-self1* [*of 0 m n*] **by** *simp*

lemma *div-nat-mult-self2-is-id* [*simp*]:
assumes $m \in Nat$ **and** $n \in Nat$ **and** $0 < n$
shows $(n * m)$ *div* $n = m$
using *assms div-nat-mult-self2*[*of 0 n m*] **by** *simp*

lemma *mod-nat-mult-self1-is-0* [*simp*]:
assumes $m \in Nat$ **and** $n \in Nat$ **and** $0 < n$
shows $(m * n)$ *mod* $n = 0$
using *assms mod-nat-mult-self1* [*of 0 m n*] **by** *simp*

lemma *mod-nat-mult-self2-is-0* [*simp*]:
assumes $m \in Nat$ **and** $n \in Nat$ **and** $0 < n$
shows $(n * m)$ *mod* $n = 0$
using *assms mod-nat-mult-self2* [*of 0 m n*] **by** *simp*

lemma *div-nat-by-1* [*simp*]:
assumes $m \in Nat$
shows m *div* $1 = m$
using *assms div-nat-mult-self1-is-id* [*of m 1*] **by** *simp*

lemma *mod-nat-by-1* [*simp*]:
assumes $m \in Nat$
shows m *mod* $1 = 0$
using *assms mod-nat-mult-self1-is-0*[*of m 1*] **by** *simp*

lemma *mod-nat-self* [*simp*]:
assumes $n \in Nat$ **and** $0 < n$
shows n *mod* $n = 0$

using *assms mod-nat-mult-self1-is-0*[*of 1*] **by** *simp*

lemma *div-nat-self* [*simp*]:
  assumes $n \in Nat$ **and** $0 < n$
  shows n *div* $n = 1$
using *assms div-nat-mult-self1-is-id* [*of 1 n*] **by** *simp*

lemma *div-nat-add-self1* [*simp*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: $0 < n$
  shows $(m + n)$ *div* $n = m$ *div* $n + 1$
using *assms div-nat-mult-self1*[*OF m oneIsNat n pos*] **by** *simp*

lemma *div-nat-add-self2* [*simp*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: $0 < n$
  shows $(n + m)$ *div* $n = m$ *div* $n + 1$
using *assms div-nat-mult-self3*[*OF m n oneIsNat pos*] **by** *simp*

lemma *mod-nat-add-self1* [*simp*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: $0 < n$
  shows $(m + n)$ *mod* $n = m$ *mod* n
using *assms mod-nat-mult-self1*[*OF m oneIsNat n pos*] **by** *simp*

lemma *mod-nat-add-self2* [*simp*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: $0 < n$
  shows $(n + m)$ *mod* $n = m$ *mod* n
using *assms mod-nat-mult-self3*[*OF m oneIsNat n pos*] **by** *simp*

lemma *div-mod-nat-decomp*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: $0 < n$
  obtains *q r* **where** $q \in Nat$ **and** $r \in Nat$
    and $q = m$ *div* n **and** $r = m$ *mod* n **and** $m = q * n + r$
proof –
  from *m n pos* **have** $m = (m$ *div* $n) * n + (m$ *mod* $n)$ **by** *simp*
  with *assms that* **show** *?thesis* **by** *blast*
qed

lemma *dvd-nat-eq-mod-eq-0*:
  assumes $m \in Nat$ **and** $n \in Nat$ **and** $0 < m$
  shows $(m$ *dvd* $n) = (n$ *mod* $m = 0)$ (**is** *?lhs* = *?rhs*)
proof –
  from *assms* **have** *1*: *?lhs* ⇒ *?rhs* **by** *auto*
  have *2*: *?rhs* ⇒ *?lhs*
  proof
    assume *mod*: n *mod* $m = 0$
    with *assms mod-div-nat-equality*[*of n m*] **have** $(n$ *div* $m) * m = n$ **by** *simp*
    with *assms* **have** $n = m * (n$ *div* $m)$ **by** (*simp add*: *mult-commute-nat*)
    with *assms* **show** *m dvd n* **by** *blast*
  qed
  from *1 2 assms* **show** *?thesis* **by** *blast*

qed

lemma *mod-div-nat-trivial* [*simp*]:
  assumes $m \in Nat$ **and** $n \in Nat$ **and** $0 < n$
  shows $(m\ mod\ n)\ div\ n = 0$
proof −
  from *assms*
  have $m\ div\ n + (m\ mod\ n)\ div\ n = (m\ mod\ n + (m\ div\ n) * n)\ div\ n$
    by (*simp add*: *mod-nat-less-divisor*)
  also from *assms* **have** ... $= m\ div\ n + 0$ **by** *simp*
  finally show *?thesis*
    using *assms* **by** *simp*
qed

lemma *mod-mod-nat-trivial* [*simp*]:
  assumes $m \in Nat$ **and** $n \in Nat$ **and** $0 < n$
  shows $(m\ mod\ n)\ mod\ n = m\ mod\ n$
proof −
  from *assms mod-nat-mult-self1* [*of m mod n m div n n*]
  have $(m\ mod\ n)\ mod\ n = (m\ mod\ n + (m\ div\ n) * n)\ mod\ n$ **by** *simp*
  also from *assms* **have** ... $= m\ mod\ n$ **by** *simp*
  finally show *?thesis* **.**
qed

lemma *dvd-nat-imp-mod-0*:
  assumes *n dvd m* **and** $m \in Nat$ **and** $n \in Nat$ **and** $0 < n$
  shows $m\ mod\ n = 0$
using *assms* **by** (*simp add*: *dvd-nat-eq-mod-eq-0*)

lemma *dvd-nat-div-mult-self*:
  assumes *dvd*: *n dvd m* **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: $0 < n$
  shows $(m\ div\ n) * n = m$
using *assms*
  dvd-nat-imp-mod-0 [*OF assms*]
  mod-div-nat-equality [*OF m n pos*]
by *simp*

lemma *dvd-nat-div-mult*:
  assumes *dvd*: *n dvd m* **and** *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *pos*: $0 < n$
    and *k*: $k \in Nat$
  shows $(m\ div\ n) * k = (m * k)\ div\ n$
proof −
  from *dvd m n* **obtain** *l* **where** *l*: $l \in Nat$ $m = n*l$ **by** *auto*
  with *m n k* **have** $m * k = n * (l*k)$ **by** (*simp add*: *mult-assoc-nat*)
  with *m n k l pos* **show** *?thesis* **by** *simp*
qed

lemma *div-nat-dvd-div* [*simp*]:
  assumes *1*: *a dvd b* **and** *2*: *a dvd c*

```
    and a: a ∈ Nat and b: b ∈ Nat and c: c ∈ Nat and pos: 0 < a
  shows (b div a) dvd (c div a) = (b dvd c)
proof (auto)
  assume lhs: (b div a) dvd (c div a)
  with a b c pos have ((b div a) ∗ a) dvd ((c div a) ∗ a)
    by (intro mult-dvd-mono, simp+)
  moreover
  from 1 a b pos have (b div a) ∗ a = b by (simp add: dvd-nat-div-mult-self)
  moreover
  from 2 a c pos have (c div a) ∗ a = c by (simp add: dvd-nat-div-mult-self)
  ultimately show b dvd c by simp
next
  assume rhs: b dvd c
  with b c obtain k where k: k ∈ Nat c = b∗k by auto
  from 1 a b obtain l where l: l ∈ Nat b = a∗l by auto
  with a pos have 3: b div a = l by simp
  from 2 a c obtain m where m: m ∈ Nat c = a∗m by auto
  with a pos have 4: c div a = m by simp
  from k l m a pos mult-assoc-nat[of a l k, symmetric] have m = l∗k by auto
  with k l m have l dvd m by auto
  with 3 4 show (b div a) dvd (c div a) by simp
qed (auto simp: assms)

lemma dvd-mod-nat-imp-dvd:
  assumes 1: k dvd (m mod n) and 2: k dvd n
    and k: k ∈ Nat and m: m ∈ Nat and n: n ∈ Nat and pos: 0 < n
  shows k dvd m
proof −
  from assms have k dvd ((m div n) ∗ n + m mod n)
    by (simp add: dvd-mult del: mod-div-nat-equality)
  with m n pos show ?thesis by simp
qed



end
```

10 Case expressions

theory *CaseExpressions*
imports *Tuples*
begin

A CASE expression in TLA$^+$ has the form

$$\text{CASE } p_1 \to e_1 \Box \ldots \Box p_n \to e_n \text{ OTHER } e_{n+1}$$

where the OTHER-branch is optional. We represent this construct by Isabelle operators $Case(ps, es)$ and $CaseOther(ps, es, oth)$ where ps is the sequence of guards p_i, *es* is the sequence of expressions e_i and *oth* is the expression that occurs in the optional OTHER-branch. The *Case* operator could be considered as a special case of *CaseOther*, and thus be avoided, by adding an OTHER-branch returning *default* (which is the result when all guards evaluate to *FALSE*). However, doing so slows down evaluation because the guard of the OTHER-branch, when present, is the conjunction of the negated guards of all other branches, so every guard appears twice (and will be simplified twice) in a *CaseOther* expression.

definition *CaseArm* — preliminary construct to convert case arm into set
where *CaseArm*(*p*,*e*) ≡ *IF p THEN* {*e*} *ELSE* {}

definition *Case* **where**
Case(*ps*, *es*) ≡ *CHOOSE x* : *x* ∈ (*UNION* { *CaseArm*(*ps*[*i*], *es*[*i*]) : *i* ∈ *DOMAIN ps* })

definition *CaseOther* **where**
CaseOther(*ps*, *es*, *oth*) ≡
CHOOSE x : *x* ∈ (*UNION* { *CaseArm*(*ps*[*i*], *es*[*i*]) : *i* ∈ *DOMAIN ps* })
∪ *CaseArm*((∀ *i* ∈ *DOMAIN ps* : ¬*ps*[*i*]), *oth*)

nonterminal *case-arm* **and** *case-arms*

syntax

```
-case-syntax:: case-arms ⇒ c                   ((CASE - ) 10)
-case1      :: [c, c] ⇒ case-arm              ((2- −>/ -) 10)
            :: case-arm ⇒ case-arms           (-)
-other      :: c ⇒ case-arms                  (OTHER −> -)
-case2      :: [case-arm, case-arms] ⇒ case-arms  (-/ [] -)
```

syntax (*xsymbols*)

```
-case1      :: [c, c] ⇒ case-arm              ((2- →/ -) 10)
-other      :: c ⇒ case-arms                  (OTHER → -)
-case2      :: [case-arm, case-arms] ⇒ case-arms  (-/ □ -)
```

syntax (*HTML* **output**)

```
-case1      :: [c, c] ⇒ case-arm              ((2- →/ -) 10)
-other      :: c ⇒ case-arms                  (OTHER → -)
-case2      :: [case-arm, case-arms] ⇒ case-arms  (-/ □ -)
```

parse-ast-translation ⟨⟨

```
let
  (* make-tuple converts a list of ASTs to a tuple formed from these ASTs.
     The order of elements is reversed. *)
  fun make-tuple [] = Ast.Constant emptySeq
    | make-tuple (t :: ts) = Ast.Appl[Ast.Constant Append, make-tuple ts, t]
  (* get-case-constituents extracts the lists of predicates, terms, and
     default value from the arms of a case expression.
     The order of the ASTs is reversed. *)
```

```
    fun get-case-constituents (Ast.Appl[Ast.Constant -other, t]) =
          (* 1st case: single OTHER arm *)
          ([], [], SOME t)
      | get-case-constituents (Ast.Appl[Ast.Constant -case1, p, t]) =
          (* 2nd case: a single arm, no OTHER branch *)
          ([p], [t], NONE)
      | get-case-constituents (Ast.Appl[Ast.Constant -case2,
                                   Ast.Appl[Ast.Constant -case1, p, t],
                                   arms]) =
          (* 3rd case: one arm, followed by remaining arms *)
          let val (ps, ts, oth) = get-case-constituents arms
          in  (ps @ [p], ts @ [t], oth)
          end
    fun case-syntax-tr [arms] =
        let val (preds, trms, oth) = get-case-constituents arms
            val pTuple = make-tuple preds
            val tTuple = make-tuple trms
        in
          if oth = NONE
          then Ast.Appl[Ast.Constant Case, pTuple, tTuple]
          else Ast.Appl[Ast.Constant CaseOther, pTuple, tTuple, the oth]
        end
      | case-syntax-tr - = raise Match;
  in
    [(-case-syntax, case-syntax-tr)]
  end
»
```

print-ast-translation «

```
  let
    fun list-from-tuple (Ast.Constant @{const-syntax emptySeq}) = []
      | list-from-tuple (Ast.Appl[Ast.Constant @tuple, tp]) =
        let fun list-from-tp (Ast.Appl[Ast.Constant @app, tp, t]) =
                    (list-from-tp tp) @ [t]
              | list-from-tp t = [t]
        in  list-from-tp tp
        end
    (* make-case-arms constructs an AST representing the arms of the
       CASE expression. The result is an AST of type case-arms.
       The lists of predicates and terms are of equal length,
       oth is optional. The lists can be empty only if oth is present,
       corresponding to a degenerated expression CASE OTHER -> e. *)
    fun make-case-arms [] [] (SOME oth) =
          (* only a single OTHER clause *)
          Ast.Appl[Ast.Constant @{syntax-const -other}, oth]
      | make-case-arms [p] [t] oth =
          let val arm = Ast.Appl[Ast.Constant @{syntax-const -case1}, p, t]
```

```
      in (* last arm: check if OTHER defaults *)
        if oth = NONE
        then arm
        else Ast.Appl[Ast.Constant @{syntax-const -case2}, arm,
              Ast.Appl[Ast.Constant @{syntax-const -other}, the oth]]
      end
  | make-case-arms (p::ps) (t::ts) oth =
      (* first arm, followed by others *)
      let val arms = make-case-arms ps ts oth
        val arm = Ast.Appl[Ast.Constant @{syntax-const -case1}, p, t]
      in Ast.Appl[Ast.Constant @{syntax-const -case2}, arm, arms]
      end
 (* CASE construct without OTHER branch *)
 fun case-syntax-tr' [pTuple, tTuple] =
    let val prds = list-from-tuple pTuple
      val trms = list-from-tuple tTuple
    in (* make sure that tuples are of equal length, otherwise give up *)
      if length prds = length trms
      then Ast.Appl[Ast.Constant @{syntax-const -case-syntax},
                make-case-arms prds trms NONE]
      else Ast.Appl[Ast.Constant Case, pTuple, tTuple]
    end
  | case-syntax-tr' - = raise Match
 (* CASE construct with OTHER branch present *)
 fun caseother-tr' [pTuple, tTuple, oth] =
    let val prds = list-from-tuple pTuple
      val trms = list-from-tuple tTuple
    in (* make sure that tuples are of equal length, otherwise give up *)
      if length prds = length trms
      then Ast.Appl[Ast.Constant @{syntax-const -case-syntax},
                make-case-arms prds trms (SOME oth)]
      else Ast.Appl[Ast.Constant CaseOther, pTuple, tTuple, oth]
    end
  | caseother-tr' - = raise Match
in
 [(@{const-syntax Case}, case-syntax-tr'),
  (@{const-syntax CaseOther}, caseother-tr')]
end
⟩⟩
```

lemmas *Case-simps* [*simp*] = *CaseArm-def Case-def CaseOther-def*

end

11 Characters and strings

theory *Strings*
imports *Tuples*
begin

11.1 Characters

Characters are represented as pairs of hexadecimal digits (also called *nibbles*).

definition *Nibble*
where $Nibble \equiv \{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\}$

definition *char* — *char* is intended to be applied to nibbles
where $char(a,b) \equiv \langle a,b \rangle$

lemma *charInj* [*simp*]: $(char(a,b) = char(c,d)) = (a=c \wedge b=d)$
by (*simp add*: *char-def*)

definition *Char*
where $Char \equiv \{ \ char(a, \ b) : \langle a,b \rangle \in Nibble \times Nibble \ \}$

lemma *isChar* [*simp*]: $(c \in Char) = (\exists \, a,b \in Nibble : c = char(a,b))$
unfolding *Char-def* **by** *auto*

11.2 Strings

definition *String*
where $String \equiv Seq(Char)$

syntax
  -Char :: *xstr* ⇒ *c*   (*CHAR* -)
  -String :: *xstr* ⇒ *c*   (-)

The following parse and print translations convert between the internal and external representations of strings. Strings are written using two single quotes in Isabelle, such as `''abc''`. Note that the empty string is just the empty sequence in TLA$^+$, so `''''` gets printed as ⟨⟩. Single characters are printed in the form `CHAR ''a''`: Isabelle doesn't provide single characters in its lexicon.

parse-ast-translation ⟪
```
  let
    (* convert an ML integer to a nibble *)
    fun mkNibble n =
      if n = 0
      then Ast.Constant Peano.zero
```

```
      else Ast.Appl [Ast.Constant Functions.fapply, Ast.Constant Peano.Succ,
mkNibble (n−1)];

    (∗ convert an ML character to a TLA+ Char ∗)
    fun mkChar c =
      if Symbol.is-ascii c
      then Ast.Appl [Ast.Constant Strings.char,
                     mkNibble (ord c div 16), mkNibble (ord c mod 16)]
      else error (Non−ASCII symbol:  ^ quote c);

    (∗ convert a list of ML characters into a TLA+ string, in reverse order ∗)
    fun list2TupleReverse [] = Ast.Constant Tuples.emptySeq
      | list2TupleReverse (c :: cs) =
          Ast.Appl [Ast.Constant Tuples.Append, list2TupleReverse cs, mkChar c];

    (∗ parse AST translation for characters ∗)
    fun char-ast-tr [Ast.Variable xstr] =
          (case Lexicon.explode-xstr xstr of
            [c] => mkChar c
          | - => error (Expected single character, not  ^ xstr))
      | char-ast-tr asts = raise Ast.AST (char-ast-tr, asts);

    (∗ parse AST translation for strings ∗)
    fun string-ast-tr [Ast.Variable xstr] =
          list2TupleReverse (rev (Lexicon.explode-xstr xstr))
      | string-ast-tr asts = raise Ast.AST (string-ast-tr, asts);
  in
    [(-Char, char-ast-tr), (-String, string-ast-tr)]
  end
⟫
```

lemma *''a''*
oops

lemma *CHAR ''a''*
oops

print-ast-translation ⟪
```
  let
    (∗ convert a nibble to an ML integer −− because translation macros have
       already been applied, we see constants 0 through 15, not Succ[...] terms! ∗)
    fun destNibble (Ast.Constant @{const-syntax zero}) = 0
```

```
  | destNibble (Ast.Constant @{const-syntax one}) = 1
  | destNibble (Ast.Constant @{const-syntax two}) = 2
  | destNibble (Ast.Constant @{const-syntax three}) = 3
  | destNibble (Ast.Constant @{const-syntax four}) = 4
  | destNibble (Ast.Constant @{const-syntax five}) = 5
  | destNibble (Ast.Constant @{const-syntax six}) = 6
  | destNibble (Ast.Constant @{const-syntax seven}) = 7
  | destNibble (Ast.Constant @{const-syntax eight}) = 8
  | destNibble (Ast.Constant @{const-syntax nine}) = 9
  | destNibble (Ast.Constant @{const-syntax ten}) = 10
  | destNibble (Ast.Constant @{const-syntax eleven}) = 11
  | destNibble (Ast.Constant @{const-syntax twelve}) = 12
  | destNibble (Ast.Constant @{const-syntax thirteen}) = 13
  | destNibble (Ast.Constant @{const-syntax fourteen}) = 14
  | destNibble (Ast.Constant @{const-syntax fifteen}) = 15
  | destNibble - = raise Match;

(* convert a pair of nibbles to an ML character *)
fun destNbls nb1 nb2 =
    let val specials = raw-explode \\\‘′
        val c = chr (destNibble nb1 * 16 + destNibble nb2)
    in  if not (member (op =) specials c) andalso Symbol.is-ascii c
          andalso Symbol.is-printable c
        then c else raise Match
    end;

(* convert a TLA+ Char to an ML character *)
fun destChar (Ast.Appl [Ast.Constant @{const-syntax char}, nb1, nb2]) =
      destNbls nb1 nb2
  | destChar arg = raise Match

(* convert a TLA+ tuple (an argument of @tuple) into a list *)
fun tuple2List (Ast.Appl[Ast.Constant @app, tp, t]) = (tuple2List tp) @ [t]
  | tuple2List t = [t];

 (* convert a list of TLA+ characters to the output representation of a TLA+
string *)
fun list2String cs =
      Ast.Appl [Ast.Constant -inner-string,
                Ast.Variable (Lexicon.implode-xstr cs)];

(* print AST translation for single characters that do not occur in a string *)
fun char-ast-tr′ [nb1, nb2] =
      Ast.Appl [Ast.Constant @{syntax-const -Char},
                list2String [destNbls nb1 nb2]]
  | char-ast-tr′ - = raise Match;

(* print AST translation for non−empty literal strings,
   fails (by raising exception Match)
```

```
    when applied to anything but a character sequence *)
  fun string-ast-tr' [args] = list2String (map destChar (tuple2List args))
    | string-ast-tr' - = raise Match;
in
  [(@{const-syntax char}, char-ast-tr'), (@tuple, string-ast-tr')]
end
⟫
```

11.3 Records and sets of records

Records are simply represented as enumerated functions with string arguments, such as (*''foo''* :> *1*) @ (*''bar''* :> *TRUE*). Similarly, there is no specific *EXCEPT* construct for records; use the standard one for functions, such as [*r EXCEPT* ![*''foo''* = *3*]]. Finally, sets of records are represented as sets of enumerated functions as in [*''foo''* : *Nat*, *''bar''* : *BOOLEAN*]. Support for standard TLA$^+$ record syntax in Isabelle seems difficult, because the Isabelle lexer distinguishes between identifiers and strings: the latter must be surrounded by two single quotes.

end

12 The Integers as a superset of natural numbers

theory *Integers*
imports *Tuples NatArith*
begin

12.1 The minus sign

consts
  minus :: *c* ⇒ *c*   (−.- [75] 75)

syntax — syntax for negative naturals
  −.0 :: c (−.0)
  −.1 :: c (−.1)
  −.2 :: c (−.2)
translations
  −.0 ⇌ −.(0)
  −.1 ⇌ −.(1)
  −.2 ⇌ −.(2)

axiomatization where
  neg0 [*simp*]: −.0 = 0

and
  neg-neg [*simp*]: $-.-.n = n$
and
  negNotInNat [*simp*]: $-.(Succ[n]) \notin Nat$

lemma *negNat-noteq-Nat* [*simp*]:
  $[\![m \in Nat; n \in Nat]\!] \Longrightarrow (-. Succ[m] = Succ[n]) = FALSE$
proof (*rule contradiction*)
  assume $(-. Succ[m] = Succ[n]) \neq FALSE$
    and *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  hence $-. Succ[m] = Succ[n]$ **by** *auto*
  hence $-. Succ[m] \in Nat$ **using** *n* **by** *auto*
  with *negNotInNat*[*of m*] **show** *FALSE* **by** *simp*
qed

lemma *negNat-noteq-Nat2* [*simp*]:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$
  shows $(Succ[m] = -. Succ[n]) = FALSE$
proof *auto*
  assume $Succ[m] = -. Succ[n]$
  hence $-. Succ[n] = Succ[m]$ **by** *simp*
  with *m n* **show** *FALSE* **by** *simp*
qed

lemma *nat-not-eq-inv*: $n \in Nat \Longrightarrow n = 0 \lor -.n \neq n$
  using *not0-implies-Suc*[*of n*] **by** *auto*

lemma *minusInj* [*dest*]:
  assumes *hyp*: $-.n = -.m$
  shows $n = m$
proof −
  from *hyp* **have** $-.-.n = -.-.m$ **by** *simp*
  thus *?thesis* **by** *simp*
qed

lemma *minusInj-iff* [*simp*]:
  $-.x = -.y = (x = y)$
by *auto*

lemma *neg0-imp-0* [*simp*]: $-.n = 0 = (n = 0)$
proof *auto*
  assume $-.n = 0$
  hence $-.-.n = 0$ **by** *simp*
  thus $n = 0$ **by** *simp*
qed

lemma *neg0-eq-0* [*dest*]: $-.n = 0 \Longrightarrow (n = 0)$
by *simp*

lemma *notneg0-imp-not0* [*dest*]: $-.n \neq 0 \Longrightarrow n \neq 0$
by *auto*

lemma *not0-imp-notNat* [*simp*]: $n \in Nat \Longrightarrow n \neq 0 \Longrightarrow -.n \notin Nat$
  using *not0-implies-Suc*[*of n*] **by** *auto*

lemma *negSuccNotZero* [*simp*]: $n \in Nat \Longrightarrow (-.\ Succ[n] = 0) = FALSE$
by *auto*

lemma *negSuccNotZero2* [*simp*]: $n \in Nat \Longrightarrow (0 = -.\ Succ[n]) = FALSE$
proof *auto*
  assume *n*: $n \in Nat$ **and** *1*: $0 = -.\ Succ[n]$
  from *1* **have** $-.\ Succ[n] = 0$ **by** *simp*
  with *n* **show** *FALSE* **by** *simp*
qed

lemma *negInNat-imp-false* [*dest*]: $-.Succ[n] \in Nat \Longrightarrow FALSE$
  using *negNotInNat*[*of n*] **by** *simp*

lemma *negInNatFalse* [*simp*]: $-.Succ[n] \in Nat = FALSE$
  using *negNotInNat*[*of n*] **by** *auto*

lemma *n-negn-inNat-is0* [*simp*]:
  assumes $n \in Nat$
  shows $-.n \in Nat = (n = 0)$
using *assms* **by** (*cases n*, *auto*)

lemma *minus-sym*: $-.a = b = (a = -.b)$
by *auto*

lemma *negNat-exists*: $-.n \in Nat \Longrightarrow \exists\, k \in Nat: n = -.k$
by *force*

lemma *nat-eq-negnat-is-0* [*simp*]:
  assumes $n \in Nat$
  shows $(n = -.n) = (n = 0)$
using *assms* **by** (*cases n*, *auto*)

lemma $\exists\, x \in Nat : -.1 = -.x$ **by** *auto*
lemma $x \in Nat \Longrightarrow (1 = -.x) = FALSE$ **by** (*auto simp*: *sym*[*OF minus-sym*])

12.2 The set of Integers

definition *Int*
where $Int \equiv Nat \cup \{-.n : n \in Nat\}$

lemma *natInInt* [*simp*]: $n \in Nat \Longrightarrow n \in Int$
by (*simp add*: *Int-def*)

lemma *intDisj*: $n \in Int \Longrightarrow n \in Nat \lor n \in \{-.n : n \in Nat\}$
by (*auto simp*: *Int-def*)

lemma *negint-eq-int* [*simp*]: $-.n \in Int = (n \in Int)$
unfolding *Int-def* **by** *force*

lemma *intCases* [*case-names Positive Negative*, *cases set*: *Int*]:
  assumes *n*: $n \in Int$
  and *sc*: $n \in Nat \Longrightarrow P$
  and *nsc*: $\bigwedge m.\ [\![m \in Nat;\ n = -.m]\!] \Longrightarrow P$
  shows *P*
using *assms* **unfolding** *Int-def* **by** *auto*

— Integer cases over two parameters
lemma *intCases2*:
  assumes *m*: $m \in Int$ **and** *n*: $n \in Int$
    and *pp*: $\bigwedge m\ n.\ [\![m \in Nat;\ n \in Nat]\!] \Longrightarrow P(m,n)$
    and *pn*: $\bigwedge m\ n.\ [\![m \in Nat;\ n \in Nat]\!] \Longrightarrow P(m,-.n)$
    and *np*: $\bigwedge m\ n.\ [\![m \in Nat;\ n \in Nat]\!] \Longrightarrow P(-.m,n)$
    and *nn*: $\bigwedge m\ n.\ [\![m \in Nat;\ n \in Nat]\!] \Longrightarrow P(-.m,-.n)$
  shows $P(m,n)$
using *m* **proof** (*cases m*)
  assume $m \in Nat$
  from *n this pp pn* **show** $P(m,n)$ **by** (*cases n*, *auto*)
next
  fix m'
  assume $m' \in Nat\ m = -.\ m'$
  from *n this np nn* **show** $P(m,n)$ **by** (*cases n*, *auto*)
qed

lemma *intCases3*:
  assumes *m*: $m \in Int$ **and** *n*: $n \in Int$ **and** *p*: $p \in Int$
    and *ppp*: $\bigwedge m\ n\ p.\ [\![m \in Nat;\ n \in Nat;\ p \in Nat]\!] \Longrightarrow P(m,n,p)$
    and *ppn*: $\bigwedge m\ n\ p.\ [\![m \in Nat;\ n \in Nat;\ p \in Nat]\!] \Longrightarrow P(m,n,-.p)$
    and *pnp*: $\bigwedge m\ n\ p.\ [\![m \in Nat;\ n \in Nat;\ p \in Nat]\!] \Longrightarrow P(m,-.n,p)$
    and *pnn*: $\bigwedge m\ n\ p.\ [\![m \in Nat;\ n \in Nat;\ p \in Nat]\!] \Longrightarrow P(m,-.n,-.p)$
    and *npp*: $\bigwedge m\ n\ p.\ [\![m \in Nat;\ n \in Nat;\ p \in Nat]\!] \Longrightarrow P(-.m,n,p)$
    and *npn*: $\bigwedge m\ n\ p.\ [\![m \in Nat;\ n \in Nat;\ p \in Nat]\!] \Longrightarrow P(-.m,n,-.p)$
    and *nnp*: $\bigwedge m\ n\ p.\ [\![m \in Nat;\ n \in Nat;\ p \in Nat]\!] \Longrightarrow P(-.m,-.n,p)$
    and *nnn*: $\bigwedge m\ n\ p.\ [\![m \in Nat;\ n \in Nat;\ p \in Nat]\!] \Longrightarrow P(-.m,-.n,-.p)$
  shows $P(m,n,p)$
proof (*rule intCases2*[*OF m n*])
  fix *m n*
  assume $m \in Nat$ **and** $n \in Nat$
  from *p this ppp ppn* **show** $P(m,n,p)$ **by** (*cases p*, *auto*)
next
  fix *m n*
  assume $m \in Nat$ **and** $n \in Nat$

from *p this pnp pnn* **show** $P(m, -.n, p)$ **by** (*cases p*, *auto*)
next
fix *m n*
assume $m \in Nat$ **and** $n \in Nat$
from *p this npp npn* **show** $P(-.m, n, p)$ **by** (*cases p*, *auto*)
next
fix *m n*
assume $m \in Nat$ **and** $n \in Nat$
from *p this nnp nnn* **show** $P(-.m, -.n, p)$ **by** (*cases p*, *auto*)
qed

lemma *int-eq-negint-is-0* [*simp*]: $n \in Int \Longrightarrow n = -.n = (n = 0)$
by(*rule intCases*, *auto*)

lemma *intNotNatIsNeg*: $[\![n \notin Nat; n \in Int]\!] \Longrightarrow \exists k \in Nat: n = -.k$
unfolding *Int-def* **by** *auto*

lemma *intNotNatIsNegNat*: $[\![n \notin Nat; n \in Int]\!] \Longrightarrow -.n \in Nat$
unfolding *Int-def* **by** *auto*

12.3 Predicates "is positive" and 'is negative'

definition *isPos* — Predicate "is positive"
where $isPos(n) \equiv \exists k \in Nat: n = Succ[k]$

definition *isNeg* — Predicate "is negative"
where $isNeg(n) \equiv \exists k \in Nat: n = -.Succ[k]$

lemma *boolify-isPos* [*simp*]: $boolify(isPos(n)) = (isPos(n))$
by (*simp add*: *isPos-def*)

lemma *isPos-isBool* [*intro*!,*simp*]: $isBool(isPos(n))$
by (*simp add*: *isPos-def*)

lemma *boolify-isNeg* [*simp*]: $boolify(isNeg(n)) = (isNeg(n))$
by (*simp add*: *isNeg-def*)

lemma *isNeg-isBool* [*intro*!,*simp*]: $isBool(isNeg(n))$
by (*simp add*: *isNeg-def*)

lemma *zeroNotPos* [*dest*]: $isPos(0) \Longrightarrow FALSE$ **by** (*auto simp*: *isPos-def*)
lemma *zeroNotNeg* [*dest*]: $isNeg(0) \Longrightarrow FALSE$ **by** (*auto simp*: *isNeg-def*)

lemma *natIsPos* [*simp*]: $n \in Nat \Longrightarrow isPos(Succ[n])$ **by**(*simp add*: *isPos-def*)
lemma *negIsNeg* [*simp*]: $n \in Nat \Longrightarrow isNeg(-.Succ[n])$ **by**(*simp add*: *isNeg-def*)

lemma *negIsNotPos* [*simp*]: $n \in Nat \Longrightarrow isPos(-.Succ[n]) = FALSE$
by(*simp add*: *isPos-def*)

lemma *isPos-eq-inNat1*: $isPos(n) = (n \in Nat \wedge n \neq 0)$
unfolding *isPos-def* **using** *not0-implies-Suc*[*of n*] **by** *auto*

lemma *isNeg-eq-inNegNat*:
$isNeg(n) = (n \in \{-.n : n \in Nat\ \} \wedge n \neq 0)$
unfolding *isNeg-def* **by** *force*

lemma *intIsPos-isNat*: $n \in Int \Longrightarrow isPos(n) \Longrightarrow n \in Nat$
by(*auto simp*: *isPos-def*)

lemma *negNotNat-isNat*:
assumes *n*: $n \in Int$ **shows** $(-.n \in Nat) = FALSE \Longrightarrow n \in Nat$
using *n* **by** (*cases*, *auto*)

lemma *noNatisNeg* [*simp*]:
$n \in Nat \Longrightarrow isNeg(n) = FALSE$ — No natural number is negative
unfolding *isNeg-def* **using** *negNotInNat* **by** *blast*

lemma *negNat-isNeg* [*intro*]: $[\![m \in Nat;\ m \neq 0]\!] \Longrightarrow isNeg(-.m)$
unfolding *isNeg-eq-inNegNat* **by** *auto*

lemma *nat-is-0-or-pos*: $(n = 0 \vee isPos(n)) = (n \in Nat)$
unfolding *isPos-def* **by** *force*

lemma *isNeg-dichotomy* : $n \in Int \Longrightarrow isNeg(-.n) \Longrightarrow isNeg(n) = FALSE$
unfolding *isNeg-def* **by** *auto*

lemma *isPos-isNeg-false* [*simp*]: $n \in Int \Longrightarrow isPos(n) \Longrightarrow isNeg(n) = FALSE$
unfolding *isPos-def* **by** *force*

lemma *isPos-neg-isNeg* [*simp*]:
assumes *n*: $n \in Int$ **shows** $isPos(-.n) = isNeg(n)$
by (*auto simp*: *minus-sym isPos-def isNeg-def*)

lemma *notIsNeg0-isPos*:
assumes *n*: $n \in Int$
shows $[\![\neg\ isNeg(n);\ n \neq 0]\!] \Longrightarrow isPos(n)$
using *n* **by** (*cases*, *auto simp*: *isPos-eq-inNat1 dest*: *negNat-isNeg*)

lemma *notIsPos-notNat* [*simp*]: $[\![\neg\ isPos(n);\ n \neq 0]\!] \Longrightarrow n \in Nat = FALSE$
by (*auto simp*: *isPos-eq-inNat1*)

lemma *intThenPosZeroNeg*:

assumes *n*: $n \in Int$
shows $isNeg(n) \lor n = 0 \lor isPos(n)$
by (*auto elim*: *notIsNeg0-isPos*[*OF n*])

12.4 Signum function and absolute value

definition *sgn* — signum function
where $sgn(n) \equiv IF\ n = 0\ THEN\ 0\ ELSE\ (IF\ isPos(n)\ THEN\ 1\ ELSE\ -.1)$

definition *abs* — absolute value
where $abs(n) \equiv IF\ sgn(n) = -.1\ THEN\ -.n\ ELSE\ n$

lemma *sgnInInt* [*simp*]: $n \in Int \Longrightarrow sgn(n) \in Int$
by (*auto simp*: *sgn-def*)

lemma *sgn0* [*simp*]: $sgn(0) = 0$
by (*simp add*: *sgn-def*)

lemma *sgnPos* [*simp*]: $n \in Nat \Longrightarrow sgn(Succ[n]) = 1$
by (*simp add*: *sgn-def*)

lemma *sgnNeg* [*simp*]: $n \in Nat \Longrightarrow sgn(-.Succ[n]) = -.1$
by (*simp add*: *sgn-def*)

lemma *sgn0-imp-0*: $sgn(n) = 0 \Longrightarrow n = 0$
by (*auto simp*: *sgn-def*)

lemma *sgn0-iff-0* [*simp*]: $(sgn(n) = 0) = (n = 0)$
by (*auto simp*: *sgn-def*)

lemma *sgn1-imp-pos* : $sgn(n) = 1 \Longrightarrow n \in Nat \land n \neq 0$
unfolding *sgn-def isPos-eq-inNat1* **by** *auto*

lemma *sgnm1-imp-neg*:
assumes *n*:$n \in Int$ **shows** $sgn(n) = -.1 \Longrightarrow isNeg(n)$
unfolding *sgn-def* **using** *intThenPosZeroNeg*[*OF n*] **by** *auto*

lemma *isPos-sgn* [*simp*]: $isPos(sgn(n)) = isPos(n)$
unfolding *isPos-def sgn-def* **by** *force*

lemma *sgnNat-is-0or1* :
$n \in Nat \Longrightarrow sgn(n) = 0 \lor sgn(n) = 1$
unfolding *sgn-def isPos-eq-inNat1* **by** *auto*

lemma *sgnNat-not0*:
$[\![n \in Nat; sgn(n) \neq 0]\!] \Longrightarrow sgn(n) = 1$
using *sgnNat-is-0or1*[*of n*] **by** *auto*

lemma *sgnNat-not1*:
$[\![n \in Nat; sgn(n) \neq 1]\!] \Longrightarrow n = 0$
using *sgnNat-is-0or1*[*of n*] **by** *auto*

lemma *sgnNat-not-neg* [*simp*]:
$n \in Nat \Longrightarrow sgn(n) = -.1 = FALSE$
unfolding *sgn-def isPos-eq-inNat1* **by** *auto*

lemma *notNat-imp-sgn-neg1* [*intro*]: $n \notin Nat \Longrightarrow sgn(n) = -.1$
unfolding *sgn-def isPos-eq-inNat1* **by** *auto*

lemma *eqSgnNat-imp-nat*: $sgn(m) = sgn(n) \Longrightarrow m \in Nat \Longrightarrow n \in Nat$
unfolding *sgn-def isPos-eq-inNat1* **by** *auto*

lemma *eqSgn-imp-0-nat* [*simp*]: $n \in Nat \Longrightarrow sgn(n) = sgn(-.n) = (n = 0)$
unfolding *sgn-def isPos-def* **by** *force*

lemma *eqSgn-imp-0-nat2* [*simp*]: $n \in Nat \Longrightarrow sgn(-.n) = sgn(n) = (n = 0)$
unfolding *sgn-def isPos-def* **by** *force*

lemma *eqSgn-imp-0* [*simp*]: $n \in Int \Longrightarrow sgn(n) = sgn(-.n) = (n = 0)$
by(*rule intCases*, *auto*)

lemma *sgn-eq-neg1-is-not-nat* : $(sgn(n) = -.1) = (n \notin Nat \land n \neq 0)$
unfolding *sgn-def isPos-eq-inNat1* **by** *auto*

lemma *sgn-not-neg1-is-nat* [*simp*]: $((sgn(n) = -.1) = FALSE) = (n \in Nat)$
by (*auto simp*: *sgn-eq-neg1-is-not-nat*)

lemma *sgn-neg-eq-1-false*: $[\![sgn(-.m) = 1; m \in Nat]\!] \Longrightarrow P$
unfolding *sgn-def* **by** *auto*

lemma *sgn-minus* [*simp*]:
assumes *n*: $n \in Int$
shows $sgn(-.n) = -.sgn(n)$
unfolding *sgn-def* **using** *n* **by** (*cases*, *auto*)

Absolute value

lemma *absIsNat* [*simp*]:
assumes *n*: $n \in Int$ **shows** $abs(n) \in Nat$
unfolding *abs-def* **using** *intNotNatIsNegNat*[*OF - n*] **by** *auto*

lemma *absNat* [*simp*]: $n \in Nat \Longrightarrow abs(n) = n$
unfolding *abs-def* **by** *auto*

lemma *abs0* [*simp*]: $abs(0) = 0$

unfolding *abs-def* **by** *simp*

lemma *abs-negNat* [*simp*]: $n \in Nat \Longrightarrow abs(-.n) = n$
unfolding *abs-def* **by** (*auto dest*: *sgnNat-not1*)

lemma *abs-neg* [*simp*]:
assumes n: $n \in Int$ **shows** $abs(-.n) = abs(n)$
unfolding *abs-def* **using** n **by** (*auto dest*: *sgnNat-not1*)

12.5 Orders on integers

We distinguish four cases, depending on the arguments being in Nat or negative.

lemmas *int-leq-pp-def* = *nat-leq-def*
— 'positive-positive' case, ie: both arguments are naturals

axiomatization where
int-leq-pn-def [*simp*]: $[\![a \in Nat; b \in Nat]\!] \Longrightarrow a \leq -.b = FALSE$
and
int-leq-np-def [*simp*]: $[\![a \in Nat; b \in Nat]\!] \Longrightarrow -.a \leq b = TRUE$
and
int-leq-nn-def [*simp*]: $[\![a \in Nat; b \in Nat]\!] \Longrightarrow -.a \leq -.b = (b \leq a)$

lemma *int-boolify-leq* [*simp*]:
$[\![a \in Int; b \in Int]\!] \Longrightarrow boolify(a \leq b) = (a \leq b)$
by(*rule intCases2*[*of a b*], *simp-all*)

lemma *int-leq-isBool* [*intro*!,*simp*]:
$[\![a \in Int; b \in Int]\!] \Longrightarrow isBool(a \leq b)$
unfolding *isBool-def* **by** *auto*

lemma *int-leq-refl* [*iff*]: $n \in Int \Longrightarrow n \leq n$
by(*rule intCases*, *auto*)

lemma *eq-leq-bothE*: — reduce equality over integers to double inequality
assumes $m \in Int$ **and** $n \in Int$ **and** $m = n$ **and** $[\![m \leq n; n \leq m]\!] \Longrightarrow P$
shows P
using *assms* **by** *simp*

lemma *neg-le-iff-le* [*simp*]:
$[\![m \in Int; n \in Int]\!] \Longrightarrow -.n \leq -.m = (m \leq n)$
by(*rule intCases2*[*of m n*], *simp-all*)

12.6 Addition of integers

Again, we distinguish four cases in the definition of $a + b$, according to each argument being positive or negative.

lemmas
int-add-pp-def $=$ *nat-add-def* — both numbers are positive, ie. naturals

axiomatization where
int-add-pn-def: $[\![a \in Nat;\ b \in Nat]\!] \Longrightarrow a + (-.b) \equiv IF\ a \leq b\ THEN\ -.(b\ --\ a)\ ELSE\ a\ --\ b$
and
int-add-np-def: $[\![a \in Nat;\ b \in Nat]\!] \Longrightarrow (-.a) + b \equiv IF\ b \leq a\ THEN\ -.(a\ --\ b)\ ELSE\ b\ --\ a$
and
int-add-nn-def [*simp*]: $[\![a \in Nat;\ b \in Nat]\!] \Longrightarrow (-.a) + (-.b) = -.(a + b)$

theorems *int-add-def* $=$ *int-add-pn-def int-add-np-def*
— When we use these definitions, we don't want to unfold the 'pp' case

lemma *int-add-neg-eq-natDiff* [*simp*]: $[\![n \leq m;\ m \in Nat;\ n \in Nat]\!] \Longrightarrow m + (-.n) = m\ --\ n$
by (*auto simp*: *int-add-pn-def dest*: *nat-leq-antisym*)

Closure

lemma *addIsInt* [*simp*]: $[\![m \in Int;\ n \in Int]\!] \Longrightarrow m + n \in Int$
by (*rule intCases2*[*of m n*], *auto simp*: *int-add-def*)

Neutral element

lemma *add-0-right-int* [*simp*]: $n \in Int \Longrightarrow n + 0 = n$
by(*rule intCases*, *auto simp add*: *int-add-np-def*)
lemma *add-0-left-int* [*simp*]: $n \in Int \Longrightarrow 0 + n = n$
by(*rule intCases*, *auto simp add*: *int-add-pn-def*)

Additive inverse element

lemma *add-inverse-nat* [*simp*]: $n \in Nat \Longrightarrow n + -.n = 0$
by(*simp add*: *int-add-pn-def*)

lemma *add-inverse2-nat* [*simp*]: $n \in Nat \Longrightarrow -.n + n = 0$
by(*simp add*: *int-add-np-def*)

lemma *add-inverse-int* [*simp*]: $n \in Int \Longrightarrow n + -.n = 0$
by (*rule intCases*, *auto simp*: *int-add-def*)

lemma *add-inverse2-int* [*simp*]: $n \in Int \Longrightarrow -.n + n = 0$
by (*rule intCases*, *auto simp*: *int-add-def*)

Commutativity

lemma *add-commute-pn-nat*: $[\![m \in Nat;\ n \in Nat]\!] \Longrightarrow m + -.n = -.n + m$

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

lemma *add-commute-int*: $[\![m \in Int; n \in Int]\!] \Longrightarrow m + n = n + m$
by(*rule intCases2*[*of m n*], *auto simp add*: *int-add-def add-commute-nat*)

Associativity

lemma *add-pn-eq-adiff* [*simp*]:
$[\![m \leq n; m \in Nat; n \in Nat]\!] \Longrightarrow m + -.n = -.(n -- m)$
by (*simp add*: *int-add-def*)

lemma *adiff-add-assoc5*:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $[\![n \leq p; \ p \leq m + n; \ m \leq p -- n]\!] \Longrightarrow -.(p -- n -- m) = m + n -- p$
apply (*induct p n rule*: *diffInduct*)
using *assms* **by** (*auto dest*: *nat-leq-antisym*)

lemma *adiff-add-assoc6*:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $[\![n \leq p; m + n \leq p; p -- n \leq m]\!] \Longrightarrow m -- (p -- n) = -.(p -- (m + n))$
apply (*induct p n rule*: *diffInduct*)
using *assms* **by** (*auto dest*: *nat-leq-antisym*)

lemma *adiff-add-assoc7*:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $[\![p + n \leq m; m \leq n]\!] \Longrightarrow -.(m -- (p + n)) = n -- m + p$
apply (*induct n m rule*: *diffInduct*)
using *assms* **by** *simp-all*

lemma *adiff-add-assoc8*:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $[\![n \leq m; p \leq m -- n; p \leq m; m -- p \leq n]\!] \Longrightarrow m -- n -- p = -.(n -- (m -- p))$
using *adiff-add-assoc6*[*OF n p m*] **apply** *simp*
using *leq-adiff-right-add-left*[*OF - p m n*] *add-commute-nat*[*OF p n*] **apply** *simp*
by(*rule adiff-adiff-left-nat*[*OF m n p*])

declare *leq-neq-iff-less* [*simplified*,*simp*]

lemma *int-add-assoc1*:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $m + (n + -.p) = (m + n) + -.p$
apply(*rule nat-leq-cases*[*OF p n*])
using *assms* **apply** *simp-all*
apply(*rule nat-leq-cases*[*of p m + n*], *simp-all*)
apply(*simp add*: *adiff-add-assoc*[*OF - m n p*])
apply(*rule nat-leq-cases*[*of p m + n*], *simp+*)
apply(*rule nat-leq-cases*[*of p -- n m*], *simp+*)

apply(*rule adiff-add-assoc3*, *simp+*)
apply(*rule adiff-add-assoc5*, *simp+*)
apply(*rule nat-leq-cases*[*of p* −− *n m*], *simp-all*)
apply(*rule adiff-add-assoc6*, *simp-all*)
apply(*simp only*: *add-commute-nat*[*of m n*])
apply(*rule adiff-adiff-left-nat*, *simp+*)
done

lemma *int-add-assoc2*:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $m + (-.p + n) = (m + -.p) + n$
using *assms* **apply** (*simp add*: *add-commute-int*[*of* −.*p n*])
using *int-add-assoc1*[*OF m n p*] **apply** *simp*
apply(*rule nat-leq-cases*[*of p m* + *n*], *simp-all*)
apply(*rule nat-leq-cases*[*OF p m*], *simp-all*)
apply(*rule adiff-add-assoc2*, *simp-all*)
apply (*simp add*: *add-commute-int*[*of* −.(*p* −− *m*) *n*])
apply(*simp only*: *add-commute-nat*[*OF m n*])
apply(*rule nat-leq-cases*[*of p* −− *m n*], *simp-all*)
apply(*rule adiff-add-assoc3*[*symmetric*], *simp+*)
apply(*rule adiff-add-assoc5*[*symmetric*], *simp+*)
apply(*rule nat-leq-cases*[*OF p m*], *simp-all*)
apply (*simp add*: *add-commute-nat*[*OF m n*])
apply (*simp add*: *add-commute-int*[*of* −.(*p* −− *m*) *n*])
apply(*rule nat-leq-cases*[*of p* −− *m n*], *simp-all*)
apply(*simp add*: *add-commute-nat*[*OF m n*])
apply(*rule adiff-add-assoc6*[*symmetric*], *simp+*)
apply(*rule adiff-adiff-left-nat*[*symmetric*], *simp+*)
done

declare *leq-neq-iff-less* [*simplified*,*simp del*]

lemma *int-add-assoc3*:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $m + -.(n + p) = m + -.n + -.p$
apply(*rule nat-leq-cases*[*of n* + *p m*])
using *assms* **apply** *simp-all*
apply(*rule nat-leq-cases*[*OF n m*], *simp-all*)
apply(*rule nat-leq-cases*[*of p m* −− *n*], *simp-all*)
apply(*rule adiff-adiff-left-nat*[*symmetric*], *simp+*)
using *adiff-add-assoc6 add-commute-nat*[*OF n p*] **apply** *simp*
using *adiff-add-assoc2*[*OF - p n m*, *symmetric*] **apply** (*simp add*: *adiff-is-0-eq′*)
apply(*rule nat-leq-cases*[*OF n m*], *simp-all*)
apply(*rule nat-leq-cases*[*of p m* −− *n*], *simp-all*)
using *adiff-add-assoc5*[*symmetric*] *add-commute-nat*[*OF n p*] **apply** *simp*
using *adiff-add-assoc3*[*symmetric*] *add-commute-nat*[*OF n p*] **apply** *simp*
using *adiff-add-assoc2*[*symmetric*] *add-commute-nat*[*OF n p*] **apply** *simp*
done

lemma *int-add-assoc4*:
  assumes *m*: *m* ∈ *Nat* **and** *n*: *n* ∈ *Nat* **and** *p*: *p* ∈ *Nat*
  shows −.*m* + (*n* + *p*) = (−.*m* + *n*) + *p*
using *assms add-commute-int*[*of* −.*m n* + *p*] *add-commute-int*[*of* −.*m n*] **apply** *simp*
apply(*rule nat-leq-cases*[*of m n* + *p*], *simp-all*)
  apply(*rule nat-leq-cases*[*OF m n*], *simp-all*)
    apply(*rule adiff-add-assoc2*, *simp*+)
    apply(*simp add*: *add-commute-int*[*of* −.(*m* −− *n*) *p*])
    apply(*rule nat-leq-cases*[*of m* −− *n p*], *simp-all*)
      apply(*simp only*: *add-commute-nat*[*of n p*])
      apply(*simp only*: *adiff-add-assoc3*[*symmetric*])
      apply(*simp only*: *add-commute-nat*[*of n p*])
      apply(*simp only*: *adiff-add-assoc5*[*symmetric*])
  apply(*rule nat-leq-cases*[*OF m n*], *simp-all*)
    apply(*simp only*: *add-commute-nat*[*of n p*])
    apply(*rule adiff-add-assoc7*, *simp-all*)
    apply(*simp add*: *add-commute-int*[*of* −.(*m* −− *n*) *p*])
    apply(*rule nat-leq-cases*[*of m* −− *n p*], *simp*+)
      apply(*simp only*: *add-commute-nat*[*of n p*])
      apply(*simp only*: *adiff-add-assoc6*[*symmetric*])
      apply(*simp only*: *add-commute-nat*[*of n p*])
        apply(*simp add*: *add-commute-nat*[*of p n*])
 apply(*rule adiff-adiff-left-nat*[*symmetric*], *simp*+)
done

lemma *int-add-assoc5*:
  assumes *m*: *m* ∈ *Nat* **and** *n*: *n* ∈ *Nat* **and** *p*: *p* ∈ *Nat*
  shows −.*m* + (*n* + −.*p*) = −.*m* + *n* + −.*p*
using *assms*
apply(*simp add*: *add-commute-int*[*of* −.*m n* + −.*p*] *add-commute-int*[*of* −.*m n*])
apply(*rule nat-leq-cases*[*OF p n*], *simp-all*)
  apply(*rule nat-leq-cases*[*of m n* −− *p*], *simp*+)
    apply(*rule nat-leq-cases*[*of m n*], *simp*+)
      apply(*rule nat-leq-cases*[*of p n* −− *m*], *simp-all*)
 apply(*rule adiff-commute-nat*[*OF n p m*])
 apply(*rule adiff-add-assoc8*, *simp*+)
 using *nat-leq-trans*[*of n m n* −− *p*] **apply** *simp*
 using *leq-adiff-right-imp-0*[*OF* - - *n p*] *nat-leq-antisym*[*of m n*] **apply** *simp*
    apply(*rule nat-leq-cases*[*OF m n*], *simp-all*)
      apply(*rule nat-leq-cases*[*of p n* −− *m*], *simp-all*)
 apply(*rule adiff-add-assoc8*[*symmetric*], *simp-all*)
 using *leq-adiff-left-add-right*[*OF* - *p n m*]
  add-commute-nat[*OF p m*]
  apply(*simp add*: *adiff-add-assoc3*)
  apply(*simp add*: *adiff-add-assoc4*)
    apply(*rule nat-leq-cases*[*of m n*], *simp-all*)
      apply(*rule nat-leq-cases*[*of p n* −− *m*], *simp*+)
 using *nat-leq-trans*[*of n p n* −− *m*] **apply** *simp*

using *leq-adiff-right-imp-0*[*OF* - - *n m*] **apply** *simp*
using *nat-leq-antisym*[*of n p*] **apply** *simp*
apply(*rule minusInj*, *simp*)
apply(*rule adiff-add-assoc4*[*symmetric*], *simp*+)
apply(*simp add*: *adiff-add-assoc2*[*symmetric*])
apply(*simp add*: *add-commute-nat*)
done

lemma *int-add-assoc6*:
assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
shows $-.m + (-.n + p) = -.(m + n) + p$
using *assms*
add-commute-int[*of* $-.n$ *p*]
add-commute-int[*of* $-.m$ $p + -.n$]
add-commute-int[*of* $-.(m + n)$ *p*] **apply** *simp*
apply(*rule nat-leq-cases*[*OF n p*], *simp-all*)
apply(*rule nat-leq-cases*[*of m p* $--$ *n*], *simp*+)
apply(*rule nat-leq-cases*[*of* $m + n$ *p*], *simp*+)
apply(*simp only*: *add-commute-nat*[*of m n*])
apply(*rule adiff-adiff-left-nat*, *simp-all*)
apply(*simp only*: *minus-sym*[*symmetric*])
apply(*rule adiff-add-assoc5*, *simp-all*)
apply(*rule nat-leq-cases*[*of* $m + n$ *p*], *simp-all*)
apply(*simp only*: *minus-sym*)
apply(*rule adiff-add-assoc6*, *simp-all*)
apply(*rule adiff-add-assoc3*, *simp-all*)
apply(*rule nat-leq-cases*[*of* $m + n$ *p*], *simp-all*)
apply(*simp only*: *minus-sym*)
apply(*rule adiff-add-assoc7*[*symmetric*], *simp-all*)
apply(*simp add*: *add-commute-nat*[*of n* $--$ *p m*])
apply(*rule adiff-add-assoc*[*symmetric*], *simp*+)
done

lemma *add-assoc-int*:
assumes *m*: $m \in Int$ **and** *n*: $n \in Int$ **and** *p*: $p \in Int$
shows $m + (n + p) = (m + n) + p$
using *m n p*
by (*rule intCases3*,
auto simp: *add-assoc-nat int-add-assoc1 int-add-assoc2 int-add-assoc3*
int-add-assoc4 int-add-assoc5 int-add-assoc6)

Minus sign distributes over addition

lemma *minus-distrib-pn-int* [*simp*]:
$m \in Nat \Longrightarrow n \in Nat \Longrightarrow -.(m + -.n) = -.m + n$
apply(*simp add*: *add-commute-int*[*of* $-.m$ *n*])
apply(*rule nat-leq-cases*[*of n m*], *simp-all*)
done

lemma *minus-distrib-np-int* [*simp*]:

$m \in Nat \Longrightarrow n \in Nat \Longrightarrow -.(-.m + n) = m + -.n$
by(*simp add*: *add-commute-int*)

lemma *int-add-minus-distrib* [*simp*]:
assumes *m*: $m \in Int$ **and** *n*: $n \in Int$
shows $-.(m + n) = -.m + -.n$
by (*rule intCases2*[*OF m n*], *simp-all*)

12.7 Multiplication of integers

axiomatization where
int-mult-pn-def: $[\![a \in Nat;\ b \in Nat]\!] \Longrightarrow a * -.b = -.(a * b)$
and
int-mult-np-def: $[\![a \in Nat;\ b \in Nat]\!] \Longrightarrow -.a * b = -.(a * b)$
and
int-mult-nn-def [*simp*]: $[\![a \in Nat;\ b \in Nat]\!] \Longrightarrow -.a * -.b = a * b$

theorems *int-mult-def* = *int-mult-pn-def int-mult-np-def*

Closure

lemma *multIsInt* [*simp*]: $[\![a \in Int;\ b \in Int]\!] \Longrightarrow a * b \in Int$
by (*rule intCases2*[*of a b*], *simp-all add*: *int-mult-def*)

Neutral element

lemma *mult-0-right-int* [*simp*]: $a \in Int \Longrightarrow a * 0 = 0$
by (*rule intCases*[*of a*], *simp-all add*: *int-mult-np-def*)

lemma *mult-0-left-int* [*simp*]: $a \in Int \Longrightarrow 0 * a = 0$
by (*rule intCases*[*of a*], *simp-all add*: *int-mult-pn-def*)

Commutativity

lemma *mult-commute-int*: $[\![a \in Int;\ b \in Int]\!] \Longrightarrow a * b = b * a$
by (*rule intCases2*[*of a b*], *simp-all add*: *int-mult-def mult-commute-nat*)

Identity element

lemma *mult-1-right-int* [*simp*]: $a \in Int \Longrightarrow a * 1 = a$
by (*rule intCases*[*of a*], *simp-all add*: *int-mult-def*)

lemma *mult-1-left-int* [*simp*]: $a \in Int \Longrightarrow 1 * a = a$
by (*rule intCases*[*of a*], *simp-all add*: *int-mult-def*)

Associativity

lemma *mult-assoc-int*:
assumes *m*: $m \in Int$ **and** *n*: $n \in Int$ **and** *p*: $p \in Int$
shows $m * (n * p) = (m * n) * p$
by(*rule intCases3*[*OF m n p*], *simp-all add*: *mult-assoc-nat int-mult-def*)

Distributivity

lemma *ppn-distrib-left-nat*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
  shows $m * (n + -.p) = m * n + -.(m * p)$
apply(*rule nat-leq-cases*[*OF p n*])
  apply(*rule nat-leq-cases*[*of m * p m * n*])
  using *assms* **apply**(*simp-all add*: *adiff-mult-distrib2-nat int-mult-def*)
done

lemma *npn-distrib-left-nat*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
  shows $-.m * (n + -.p) = -.(m * n) + m * p$
using *assms* **apply** (*simp add*: *add-commute-int*[*of* $-.(m * n)$ $m * p$])
apply(*rule nat-leq-cases*[*OF p n*])
  apply(*rule nat-leq-cases*[*of m * p m * n*], *simp-all*)
    apply (*auto simp*: *adiff-mult-distrib2-nat int-mult-def dest*: *nat-leq-antisym*)
done

lemma *nnp-distrib-left-nat*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
  shows $-.m * (-.n + p) = m * n + -.(m * p)$
using *assms* **apply** (*simp add*: *add-commute-int*[*of* $-.n$ *p*])
apply(*rule nat-leq-cases*[*OF p n*])
  apply(*rule nat-leq-cases*[*of m * p m * n*], *simp-all*)
    apply (*auto simp*: *adiff-mult-distrib2-nat int-mult-def dest*: *nat-leq-antisym*)
done

lemma *distrib-left-int*:
  assumes *m*: $m \in Int$ **and** *n*: $n \in Int$ **and** *p*: $p \in Int$
  shows $m * (n + p) = (m * n + m * p)$
apply(*rule intCases3*[*OF m n p*],
  simp-all only: *int-mult-def int-add-nn-def int-mult-nn-def addIsNat*)
      apply(*rule add-mult-distrib-left-nat*, *assumption+*)
      apply(*rule ppn-distrib-left-nat*, *assumption+*)
      apply(*simp add*: *add-commute-int*, *rule ppn-distrib-left-nat*, *assumption+*)
      apply(*simp only*: *int-add-nn-def multIsNat add-mult-distrib-left-nat*)+
      apply(*rule npn-distrib-left-nat*, *assumption+*)
      apply(*rule nnp-distrib-left-nat*, *assumption+*)
      apply(*simp only*: *add-mult-distrib-left-nat*)
done

lemma *pnp-distrib-right-nat*:
  assumes *m*: $m \in Nat$ **and** *n*: $n \in Nat$ **and** *p*: $p \in Nat$
  shows $(m + -.n) * p = m * p + -.(n * p)$
apply(*rule nat-leq-cases*[*OF n m*])
  apply(*rule nat-leq-cases*[*of n * p m * p*])
  using *assms* **apply**(*simp-all add*: *adiff-mult-distrib-nat int-mult-def*)
done

lemma *pnn-distrib-right-nat*:

assumes m: $m \in Nat$ **and** n: $n \in Nat$ **and** p: $p \in Nat$
shows $(m + -.n) * -.p = -.(m * p) + n * p$
using *assms* **apply** (*simp add*: *add-commute-int*[*of* $-.(m * p)$ $n * p$])
apply(*rule nat-leq-cases*[*OF n m*])
apply(*rule nat-leq-cases*[*of* $n * p$ $m * p$])
apply (*auto simp*: *adiff-mult-distrib-nat int-mult-def dest*: *nat-leq-antisym*)
done

lemma *npn-distrib-right-nat*:
assumes m: $m \in Nat$ **and** n: $n \in Nat$ **and** p: $p \in Nat$
shows $(-.m + n) * -.p = m * p + -.(n * p)$
using *assms* **apply** (*simp add*: *add-commute-int*[*of* $-.m$ n])
apply(*rule nat-leq-cases*[*OF n m*])
apply(*rule nat-leq-cases*[*of* $n * p$ $m * p$])
apply (*auto simp*: *adiff-mult-distrib-nat int-mult-def dest*: *nat-leq-antisym*)
done

lemma *distrib-right-int*:
assumes m: $m \in Int$ **and** n: $n \in Int$ **and** p: $p \in Int$
shows $(m + n) * p = (m * p + n * p)$
apply(*rule intCases3*[*OF m n p*],
simp-all only: *int-mult-def int-add-nn-def int-mult-nn-def addIsNat*)
apply(*rule add-mult-distrib-right-nat*, *assumption*+)
apply(*simp only*: *int-add-nn-def multIsNat add-mult-distrib-right-nat*)
apply(*rule pnp-distrib-right-nat*, *assumption*+)
apply(*rule pnn-distrib-right-nat*, *assumption*+)
apply(*simp add*: *add-commute-int*, *rule pnp-distrib-right-nat*, *assumption*+)
apply(*rule npn-distrib-right-nat*, *assumption*+)
apply(*simp only*: *int-add-nn-def multIsNat add-mult-distrib-right-nat*)
apply(*simp only*: *add-mult-distrib-right-nat*)
done

Minus sign distributes over multiplication

lemma *minus-mult-left-int*:
assumes m: $m \in Int$ **and** n: $n \in Int$
shows $-.(m * n) = -.m * n$
by (*rule intCases2*[*OF m n*], *simp-all add*: *int-mult-def*)

lemma *minus-mult-right-int*:
assumes m: $m \in Int$ **and** n: $n \in Int$
shows $-.(m * n) = m * -.n$
by (*rule intCases2*[*OF m n*], *simp-all add*: *int-mult-def*)

12.8 Difference of integers

Difference over integers is simply defined as addition of the complement. Note that this difference, noted $-$, is different from the difference over natural numbers, noted $--$, even for two natural numbers, because the latter cuts off at 0.

definition *diff* (**infixl** $-$ *65*)
where *int-diff-def*: $[\![m \in Int; n \in Int]\!] \Longrightarrow m - n = m + -.n$

lemma *diffIsInt* [*simp*]: — Closure
$[\![m \in Int; n \in Int]\!] \Longrightarrow m - n \in Int$
by (*simp add*: *int-diff-def*)

lemma *diff-neg-is-add* [*simp*]: $[\![m \in Int; n \in Int]\!] \Longrightarrow m - -.n = m + n$
by (*simp add*: *int-diff-def*)

lemma *diff-0-right-int* [*simp*]: $m \in Int \Longrightarrow m - 0 = m$
by (*simp add*: *int-diff-def*)

lemma *diff-0-left-int* [*simp*]: $n \in Int \Longrightarrow 0 - n = -.n$
by (*simp add*: *int-diff-def*)

lemma *diff-self-eq-0-int* [*simp*]: $m \in Int \Longrightarrow m - m = 0$
by (*simp add*: *int-diff-def*)

lemma *neg-diff-is-diff* [*simp*]: $[\![m \in Int; n \in Int]\!] \Longrightarrow -.(m - n) = n - m$
using *assms* **by** (*simp add*: *int-diff-def add-commute-int*)

lemma *diff-nat-is-add-neg*: $[\![m \in Nat; n \in Nat]\!] \Longrightarrow m - n = m + -.n$
by (*simp add*: *int-diff-def*)

end

13 Main theory for constant-level Isabelle/TLA$^+$

theory *Constant*
imports *NatDivision CaseExpressions Strings Integers*
begin

This is just an umbrella for the component theories.

end

theory *Zenon*
imports *Constant*
begin

The following lemmas make a cleaner meta-object reification

lemma *atomize-meta-bAll* [*atomize*]:

$(\bigwedge x.\ (x \in S \Longrightarrow P(x)))$
$\equiv$ *Trueprop* $(\forall\ x \in S : P(x))$
proof
assume $(\bigwedge x.\ (x \in S \Longrightarrow P(x)))$
thus $\forall\ x \in S : P(x)$ **..**
next
assume $\forall\ x \in S : P(x)$
thus $(\bigwedge x.\ (x \in S \Longrightarrow P(x)))$ **..**
qed

lemma *atomize-object-bAll* [*atomize*]:
Trueprop $(\forall x : (x \in S) \Rightarrow P(x))$
$\equiv$ *Trueprop* $(\forall x \in S : P(x))$
proof
assume $\forall x : x \in S \Rightarrow P(x)$
thus $\forall x \in S : P(x)$ **by** *fast*
next
assume $\forall x \in S : P(x)$
thus $\forall x : x \in S \Rightarrow P(x)$ **by** *fast*
qed

lemma *zenon-nnpp*: (~*P* ==> *FALSE*) ==> *P*
by *blast*

lemma *zenon-em*: (*P* ==> *FALSE*) ==> (~*P* ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-eqrefl*: *t* = *t*
by *simp*

lemma *zenon-nottrue*: ~*TRUE* ==> *FALSE*
by *blast*

lemma *zenon-noteq*: ~*x*=*x* ==> *FALSE*
by *blast*

lemma *zenon-eqsym* : *a* = *b* ==> *b* ~= *a* ==> *FALSE*
using *not-sym* **by** *blast*

lemma *zenon-FALSE-neq-TRUE*: *FALSE* ~= *TRUE*
by (*rule false-neq-true*)

lemma *zenon-and*: *P* & *Q* ==> (*P* ==> *Q* ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-and-0*: *P* & *Q* ==> *P*
by *blast*

lemma *zenon-and-1*: *P* & *Q* ==> *Q*
by *blast*

lemma *zenon-or*: *P* | *Q* ==> (*P* ==> *FALSE*) ==> (*Q* ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-imply*: *P* => *Q* ==> (~*P* ==> *FALSE*) ==> (*Q* ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-equiv*:
P <=> *Q* ==> (~*P* ==> ~*Q* ==> *FALSE*) ==> (*P* ==> *Q* ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-notnot*: ~~*P* ==> (*P* ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-notnot-0*: ~~*P* ==> *P*
by *blast*

lemma *zenon-notand*: ~(*P* & *Q*) ==> (~*P* ==> *FALSE*) ==> (~*Q* ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-notor*: ~(*P* | *Q*) ==> (~*P* ==> ~*Q* ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-notor-0*: ~(*P* | *Q*) ==> ~*P*
by *blast*

lemma *zenon-notor-1*: ~(*P* | *Q*) ==> ~*Q*
by *blast*

lemma *zenon-notimply*: ~(*A*=>*B*) ==> (*A* ==> ~*B* ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-notimply-0*: ~(*A*=>*B*) ==> *A*
by *blast*

lemma *zenon-notimply-1*: ~(*A*=>*B*) ==> ~*B*
by *blast*

lemma *zenon-notequiv*:
~(*P* <=> *Q*) ==> (~*P* ==> *Q* ==> *FALSE*) ==> (*P* ==> ~*Q* ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-ex*: $\backslash E\ x : P\ (x)$ ==> (!!x. $P\ (x)$ ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-ex-choose*:
  $\backslash E\ x : P\ (x)$ ==> (P (*CHOOSE* $x : P\ (x)$) ==> *FALSE*) ==> *FALSE*
proof −
  assume *goal*: $\backslash E\ x : P(x)$
  and *sub*: P (*CHOOSE* $x : P(x)$) ==> *FALSE*
  show *FALSE*
  proof (*rule sub*)
    from *goal* **show** P (*CHOOSE* $x : P(x)$)
    by (*rule chooseI-ex*)
  qed
qed

lemma *zenon-ex-choose-0*: $\backslash E\ x : P\ (x)$ ==> P (*CHOOSE* $x : P\ (x)$)
proof (*rule zenon-nnpp*)
  assume *goal*: $\backslash E\ x : P(x)$
  assume *nh*: ~ P (*CHOOSE* $x : P(x)$)
  show *FALSE*
  proof (*rule zenon-ex-choose*)
    assume *h*: P (*CHOOSE* $x : P(x)$)
    show *FALSE*
    by (*rule notE* [*OF nh h*])
  next
    show $\backslash E\ x : P(x)$
    by *fact*
  qed
qed

lemma *zenon-all*: $\backslash A\ x : P\ (x)$ ==> ($P\ (t)$ ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-all-0*: $\backslash A\ x : P\ (x)$ ==> $P\ (t)$
by *blast*

lemma *zenon-notex*: ~($\backslash E\ x : P\ (x)$) ==> (~$P\ (t)$ ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-notex-0*: ~($\backslash E\ x : P\ (x)$) ==> ~$P\ (t)$
by *blast*

lemma *zenon-notall*: ~($\backslash A\ x : P\ (x)$) ==> (!!x. ~$P\ (x)$ ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-notallex*: ~($\backslash A\ x : P\ (x)$) ==> ($\backslash E\ x :$ ~$P(x)$ ==> *FALSE*) ==> *FALSE*
by *blast*

```
lemma zenon-notallex-0: ~(\A x : P (x)) ==> \E x : ~P(x)
by blast

lemma zenon-notall-choose:
  ~(\A x : P (x)) ==> (~P (CHOOSE x : ~P (x)) ==> FALSE) ==> FALSE
proof −
  assume goal: ~(\A x : P (x))
  and sub: ~P (CHOOSE x : ~P (x)) ==> FALSE
  show FALSE
  proof (rule notE [OF goal])
    have pch: P(CHOOSE x : ~P(x)) by (rule contradiction [OF sub])
    have univ: !!x . P(x)
    proof −
      fix x
      show P(x)
      proof (rule contradiction)
        assume npx: ~P(x)
        show FALSE
        proof (rule notE [OF - pch])
          from npx show ~P (CHOOSE x : ~P(x))
          by (rule chooseI [of λ v . ~P(v) x])
        qed
      qed
    qed
    show \A x : P(x) by (rule allI [OF univ])
  qed
qed

lemma zenon-notall-choose-0:
  ~(\A x : P (x)) ==> ~P (CHOOSE x : ~P (x))
proof (rule zenon-nnpp)
  assume goal: ~(\A x : P(x))
  assume nnh: ~~P(CHOOSE x : ~P(x))
  show FALSE
  proof (rule zenon-notall-choose)
    show ~(\A x : P(x)) by fact
  next
    assume nh: ~P(CHOOSE x : ~P(x))
    show FALSE
    by (rule notE [OF nnh nh])
  qed
qed

lemma zenon-choose-diff-choose:
  (CHOOSE x : P(x)) ~= (CHOOSE x : Q (x)) ==>
   ((\E x : ~(P(x) <=> Q(x))) ==> FALSE) ==> FALSE
proof −
  assume h1: (CHOOSE x : P(x)) ~= (CHOOSE x : Q (x))
```

```
  assume h2: ((\E x : ~(P(x) <=> Q(x))) ==> FALSE)
  show FALSE
  proof (rule notE [OF h1])
    show (CHOOSE x : P(x)) = (CHOOSE x : Q (x))
    proof (rule choose-det)
      fix x
      show P(x) <=> Q(x)
      using h2 by blast
    qed
  qed
qed

lemma zenon-choose-diff-choose-0:
  (CHOOSE x : P(x)) ~= (CHOOSE x : Q (x)) ==> \E x : ~(P(x) <=> Q(x))
proof -
  assume h1: (CHOOSE x : P(x)) ~= (CHOOSE x : Q (x))
  show \E x : ~(P(x) <=> Q(x))
  proof (rule zenon-nnpp)
    assume h2: ~(\E x : ~(P(x) <=> Q(x)))
    show FALSE
    proof (rule zenon-choose-diff-choose [OF h1])
      assume h3: \E x : ~(P(x) <=> Q(x))
      with h2 show FALSE ..
    qed
  qed
qed

lemma zenon-notequalchoose:
  ((\E x : P(x)) ==> FALSE) ==>
   ((~(\E x : P(x))) ==> ~P(e) ==> FALSE) ==>
   FALSE
by blast

lemma zenon-p-eq-l:
  e ==> e1 = e2 ==>
   (e ~= e1 ==> FALSE) ==>
   (e2 ==> FALSE) ==>
   FALSE
by blast

lemma zenon-p-eq-r:
  e ==> e1 = e2 ==>
   (e ~= e2 ==> FALSE) ==>
   (e1 ==> FALSE) ==>
   FALSE
by blast

lemma zenon-np-eq-l:
  ~e ==> e1 = e2 ==>
```

(*e* ~= *e1* ==> *FALSE*) ==>
(~*e2* ==> *FALSE*) ==>
FALSE
by *blast*

lemma *zenon-np-eq-r*:
~*e* ==> *e1* = *e2* ==>
(*e* ~= *e2* ==> *FALSE*) ==>
(~*e1* ==> *FALSE*) ==>
FALSE
by *blast*

lemma *zenon-in-emptyset* : *x* \in {} ==> *FALSE*
by *blast*

lemma *zenon-in-upair* :
x \in *upair* (*y*, *z*) ==> (*x* = *y* ==> *FALSE*) ==> (*x* = *z* ==> *FALSE*) ==>
FALSE
using *upairE* **by** *blast*

lemma *zenon-notin-upair* :
x \notin *upair* (*y*, *z*) ==> (*x* ~= *y* ==> *x* ~= *z* ==> *FALSE*) ==> *FALSE*
using *upairI1 upairI2* **by** *blast*

lemma *zenon-notin-upair-0* :
x \notin *upair* (*y*, *z*) ==> *x* ~= *y*
using *upairI1* **by** *blast*

lemma *zenon-notin-upair-1* :
x \notin *upair* (*y*, *z*) ==> *x* ~= *z*
using *upairI2* **by** *blast*

lemma *zenon-in-addElt* :
x \in *addElt* (*a*, *A*) ==> (*x* = *a* ==> *FALSE*) ==> (*x* \in *A* ==> *FALSE*)
==> *FALSE*
by *blast*

lemma *zenon-notin-addElt* :
x \notin *addElt* (*a*, *A*) ==> (*x* ~= *a* ==> *x* \notin *A* ==> *FALSE*) ==>
FALSE
by *blast*

lemma *zenon-notin-addElt-0* :
x \notin *addElt* (*a*, *A*) ==> *x* ~= *a*
by *blast*

lemma *zenon-notin-addElt-1* :

x \notin addElt (a, A) ==> x \notin A
by *blast*

lemma *zenon-in-SUBSET* :
A \in SUBSET (S) ==> (A \subseteq S ==> FALSE) ==> FALSE
by *blast*

lemma *zenon-in-SUBSET-0* :
A \in SUBSET (S) ==> A \subseteq S
by *blast*

lemma *zenon-notin-SUBSET* :
A \notin SUBSET (S) ==> (~A \subseteq S ==> FALSE) ==> FALSE
by *blast*

lemma *zenon-notin-SUBSET-0* :
A \notin SUBSET (S) ==> ~A \subseteq S
by *blast*

lemma *zenon-in-UNION* :
x \in UNION s ==> (\E b : b \in s & x \in b ==> FALSE) ==> FALSE
by *blast*

lemma *zenon-in-UNION-0* :
x \in UNION s ==> \E b : b \in s & x \in b
by *blast*

lemma *zenon-notin-UNION* :
x \notin UNION s ==> (~(\E b : b \in s & x \in b) ==> FALSE) ==> FALSE
by *blast*

lemma *zenon-notin-UNION-0* :
x \notin UNION s ==> ~(\E b : b \in s & x \in b)
by *blast*

lemma *zenon-in-cup* :
x \in A \cup B ==> (x \in A ==> FALSE) ==> (x \in B ==> FALSE) ==> FALSE
by *blast*

lemma *zenon-notin-cup* :
x \notin A \cup B ==> (x \notin A ==> x \notin B ==> FALSE) ==> FALSE
by *blast*

lemma *zenon-notin-cup-0* :
  x \notin A \cup B ==> x \notin A
by *blast*

lemma *zenon-notin-cup-1* :
  x \notin A \cup B ==> x \notin B
by *blast*

lemma *zenon-in-cap* :
  x \in A \cap B ==> (x \in A ==> x \in B ==> FALSE) ==> FALSE
by *blast*

lemma *zenon-in-cap-0* :
  x \in A \cap B ==> x \in A
by *blast*

lemma *zenon-in-cap-1* :
  x \in A \cap B ==> x \in B
by *blast*

lemma *zenon-notin-cap* :
  x \notin A \cap B ==> (x \notin A ==> FALSE) ==> (x \notin B ==> FALSE)
  ==> FALSE
by *blast*

lemma *zenon-in-setminus* :
  x \in A \ B ==> (x \in A ==> x \notin B ==> FALSE) ==> FALSE
by *blast*

lemma *zenon-in-setminus-0* :
  x \in A \ B ==> x \in A
by *blast*

lemma *zenon-in-setminus-1* :
  x \in A \ B ==> x \notin B
by *blast*

lemma *zenon-notin-setminus* :
  x \notin A \ B ==> (x \notin A ==> FALSE) ==> (x \in B ==> FALSE)
  ==> FALSE
by *blast*

lemma *zenon-in-subsetof* :
  x \in subsetOf (S, P) ==> (x \in S ==> P(x) ==> FALSE) ==> FALSE
by *blast*

lemma *zenon-in-subsetof-0* :

x \in *subsetOf* (*S*, *P*) ==> *x* \in *S*
by *blast*

lemma *zenon-in-subsetof-1* :
x \in *subsetOf* (*S*, *P*) ==> *P*(*x*)
by *blast*

lemma *zenon-notin-subsetof* :
~(*t* \in *subsetOf* (*S*, *P*)) ==> (~*t* \in *S* ==> *FALSE*) ==> (~*P*(*t*) ==> *FALSE*)
==> *FALSE*
by *blast*

lemma *zenon-in-setofall* :
x \in *setOfAll* (*S*, *e*) ==> (\E *y* : *y* \in *S* & *x* = *e*(*y*) ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-in-setofall-0* :
x \in *setOfAll* (*S*, *e*) ==> \E *y* : *y* \in *S* & *x* = *e*(*y*)
by *blast*

lemma *zenon-notin-setofall* :
~(*x* \in *setOfAll* (*S*, *e*)) ==> (~ (\E *y* : *y* \in *S* & *x* = *e*(*y*)) ==> *FALSE*)
==> *FALSE*
by *blast*

lemma *zenon-notin-setofall-0* :
~(*x* \in *setOfAll* (*S*, *e*)) ==> ~ (\E *y* : *y* \in *S* & *x* = *e*(*y*))
by *blast*

lemma *zenon-all-in-0* :
$\forall$ *x* $\in$ *S* : *P* (*x*) ==> *a* \in *S* ==> *P* (*a*)
by *blast*

lemma *zenon-notex-in-0* :
~($\exists$ *x* $\in$ *S* : *P* (*x*)) ==> *a* \in *S* ==> ~*P* (*a*)
by *blast*

lemma *zenon-cup-subseteq* :
A \cup *B* \subseteq *C* ==>
(*A* \subseteq *C* ==> *B* \subseteq *C* ==> *FALSE*) ==>
FALSE
by *blast*

```
lemma zenon-cup-subseteq-0 :
  A \cup B \subseteq C ==> A \subseteq C
by blast

lemma zenon-cup-subseteq-1 :
  A \cup B \subseteq C ==> B \subseteq C
by blast

lemma zenon-not-cup-subseteq :
  ~ A \cup B \subseteq C ==>
   (~A \subseteq C ==> FALSE) ==>
   (~B \subseteq C ==> FALSE) ==>
   FALSE
by blast

lemma zenon-subseteq-cap :
  A \subseteq B \cap C ==>
   (A \subseteq B ==> A \subseteq C ==> FALSE) ==>
   FALSE
by blast

lemma zenon-subseteq-cap-0 :
  A \subseteq B \cap C ==> A \subseteq B
by blast

lemma zenon-subseteq-cap-1 :
  A \subseteq B \cap C ==> A \subseteq C
by blast

lemma zenon-not-subseteq-cap :
  ~ A \subseteq B \cap C ==>
   (~A \subseteq B ==> FALSE) ==>
   (~A \subseteq C ==> FALSE) ==>
   FALSE
by blast

lemma zenon-nouniverse : ~ (\E x : x \notin S) ==> FALSE
proof -
  assume h0: ~ (\E x : x \notin S)
  let ?w = {x \in S : x \notin x}
  have h4: ?w \in ?w ∨ ?w \notin ?w by (rule excluded-middle)
  have h6: ?w \in S using h0 by auto
  show FALSE
  proof (rule disjE [OF h4])
    assume h5: ?w \in ?w
    have h7: ?w \notin ?w using h5 h6 by blast
    show FALSE using h5 h7 by blast
  next
    assume h5: ?w \notin ?w
```

```
    have h7: ?w \in ?w using h5 h6 by blast
    show FALSE using h5 h7 by blast
  qed
qed


lemma zenon-in-funcset :
  f \in FuncSet (A, B) ==>
    (isAFcn(f) ==> DOMAIN f = A ==> \A x : x \in A => f[x] \in B ==>
FALSE)
   ==> FALSE
using FuncSet by blast

lemma zenon-in-funcset-0 :
  f \in FuncSet (A, B) ==> isAFcn(f)
using FuncSet by blast

lemma zenon-in-funcset-1 :
  f \in FuncSet (A, B) ==> DOMAIN f = A
using FuncSet by blast

lemma zenon-in-funcset-2 :
  f \in FuncSet (A, B) ==> (\A x : x \in A => f[x] \in B)
using FuncSet by blast

lemma zenon-notin-funcset :
  f \notin FuncSet (A, B) ==>
    (~isAFcn(f) ==> FALSE) ==>
    (DOMAIN f ~= A ==> FALSE) ==>
    (~(\A x : x \in A => f[x] \in B) ==> FALSE)
   ==> FALSE
using FuncSet by blast

lemma zenon-setequal :
  A = B ==> (\A x : x \in A <=> x \in B ==> FALSE) ==> FALSE
by blast

lemma zenon-setequal-0 :
  A = B ==> \A x : x \in A <=> x \in B
by blast

lemma zenon-setequalempty :
  A = {} ==> (\A x : ~x \in A ==> FALSE) ==> FALSE
by blast

lemma zenon-setequalempty-0 :
  A = {} ==> \A x : ~x \in A
by blast
```

```
lemma zenon-notsetequal :
  A ~= B ==> (~(\A x : x \in A <=> x \in B) ==> FALSE) ==> FALSE
using extension by blast

lemma zenon-notsetequal-0 :
  A ~= B ==> ~(\A x : x \in A <=> x \in B)
using extension by blast

lemma zenon-funequal :
  f = g ==> (((isAFcn(f) <=> isAFcn(g))
             & DOMAIN f = DOMAIN g)
            & (\A x : f[x] = g[x])
            ==> FALSE)
   ==> FALSE
by blast

lemma zenon-funequal-0 :
  f = g ==> ((isAFcn(f) <=> isAFcn(g))
            & DOMAIN f = DOMAIN g)
           & (\A x : f[x] = g[x])
by blast

lemma zenon-notfunequal :
  f ~= g ==> (~(((isAFcn(f)
                 & isAFcn(g))
                & DOMAIN f = DOMAIN g)
               & (\A x : x \in DOMAIN g => f[x] = g[x]))
             ==> FALSE)
   ==> FALSE
proof -
  have h1: f ~= g ==>
          isAFcn(f) ==>
          isAFcn(g) ==>
          DOMAIN f = DOMAIN g ==>
          (\A x : x \in DOMAIN g => f[x] = g[x]) ==>
          FALSE
  proof -
    assume main: ~ (f = g)
    assume h1: isAFcn (f)
    assume h2: isAFcn (g)
    assume h3: DOMAIN f = DOMAIN g
    assume h4: \A x : x \in DOMAIN g => f[x] = g[x]
    have h5: f = g ==> FALSE using main by blast
    have h6: ∀ x ∈ DOMAIN g : f[x] = g[x] using h4 by blast
    show FALSE
    proof (rule h5)
      have h7: DOMAIN f = DOMAIN g & (∀ x ∈ DOMAIN g : f[x] = g[x])
              (is ?cond)
```

```
      using h3 h6 by blast
      have h8: ?cond = (f = g)
      proof (rule sym)
        show (f = g) = ?cond
        by (rule fcnEqualIff [OF h1 h2])
      qed
      show f = g by (rule subst [OF h8], fact h7)
    qed
  qed
  thus f ~= g ==> (~(((isAFcn(f)
                      & isAFcn(g))
                     & DOMAIN f = DOMAIN g)
                    & (\A x : x \in DOMAIN g => f[x] = g[x]))
              ==> FALSE)
   ==> FALSE
  by blast
qed

lemma zenon-notfunequal-0 :
  f ~= g ==> ~(((isAFcn(f)
                & isAFcn(g))
               & DOMAIN f = DOMAIN g)
              & (\A x : x \in DOMAIN g => f[x] = g[x]))
using zenon-notfunequal by blast

lemma zenon-fapplyfcn :
  P(Fcn(S,e)[x]) ==> (x \notin S ==> FALSE) ==> (P(e(x)) ==> FALSE)
==> FALSE
proof -
  assume main: P(Fcn(S,e)[x])
  assume h1: x \notin S ==> FALSE
  have h1x: x \in S using h1 by blast
  assume h2: P(e(x)) ==> FALSE
  show FALSE
  proof (rule h2)
    have h3: Fcn(S,e)[x] = e(x)
    using h1x by (rule fapply)
    show P(e(x))
    using main by (rule subst [OF h3])
  qed
qed

lemma zenon-fapplyexcept :
  P(except(f,v,e)[w]) ==>
   (w \in DOMAIN f ==> v = w ==> P(e) ==> FALSE) ==>
   (w \in DOMAIN f ==> v ~= w ==> P(f[w]) ==> FALSE) ==>
   (~ w \in DOMAIN f ==> FALSE) ==>
   FALSE
proof -
```

```
assume main: P(except(f,v,e)[w])
assume h1: w \in DOMAIN f ==> v = w ==> P(e) ==> FALSE
assume h2: w \in DOMAIN f ==> v ~= w ==> P(f[w]) ==> FALSE
assume h3: ~ w \in DOMAIN f ==> FALSE
show FALSE
proof (rule disjE [of w \in DOMAIN f ~ w \in DOMAIN f])
  show w \in DOMAIN f | ~ w \in DOMAIN f by (rule excluded-middle)
next
  assume h5: w \in DOMAIN f
  show FALSE
  proof (cases w = v)
    assume h6: w = v
    show FALSE
    proof (rule h1)
      have h7: P(IF w = v THEN e ELSE f[w])
      proof (rule subst [of [f EXCEPT ![v] = e][w] - P])
        show [f EXCEPT ![v] = e][w] = (IF w = v THEN e ELSE f[w])
        by (rule applyExcept [OF h5])
      next
        show P([f EXCEPT ![v] = e][w])
        by (rule main)
      qed
      have h8: P(IF TRUE THEN e ELSE f[w])
      proof (rule subst [of w = v TRUE])
        show w = v = TRUE by (rule eqTrueI [OF h6])
      next
        show P(IF w = v THEN e ELSE f[w]) by (rule h7)
      qed
      have h9: P(e)
      proof (rule subst [of IF TRUE THEN e ELSE f[w] - P])
        show (IF TRUE THEN e ELSE f[w]) = e by blast
      next
        show P(IF TRUE THEN e ELSE f[w]) by (rule h8)
      qed
      show P(e) by (rule h9)
    next
      show w \in DOMAIN f by (rule h5)
    next
      show v = w by (rule sym[OF h6])
    qed
  next
    assume h10: w ~= v
    show FALSE
    proof (rule h2)
      have h12: P(f[w])
      proof (rule subst [of IF FALSE THEN e ELSE f[w] - P])
        show (IF FALSE THEN e ELSE f[w]) = f[w] by blast
      next
        show P(IF FALSE THEN e ELSE f[w])
```

```
        proof (rule subst [of IF w=v THEN e ELSE f[w] - P])
          show (IF w = v THEN e ELSE f[w]) = (IF FALSE THEN e ELSE
f[w])
          using h10 by blast
        next
          show P(IF w = v THEN e ELSE f[w])
          proof (rule subst [of [f EXCEPT ![v] = e][w] - P])
            show [f EXCEPT ![v] = e][w] = (IF w = v THEN e ELSE f[w])
            by (rule applyExcept, fact)
          next
            show P([f EXCEPT ![v] = e][w]) by (rule main)
          qed
        qed
      qed
      show P(f[w]) by (rule h12)
    next
      show w \in DOMAIN f by (rule h5)
    next
      show v ~= w by (rule not-sym[OF h10])
    qed
  qed
next
  show ~ w \in DOMAIN f ==> FALSE by (rule h3)
qed
qed

lemma zenon-boolcase :
  X = TRUE | X = FALSE ==>
   P(X) ==>
   (X = TRUE ==> P(TRUE) ==> FALSE) ==>
   (X = FALSE ==> P(FALSE) ==> FALSE) ==>
   FALSE
proof -
  assume isbool: X = TRUE | X = FALSE
  assume main: P(X)
  assume h1: X = TRUE ==> P(TRUE) ==> FALSE
  assume h2: X = FALSE ==> P(FALSE) ==> FALSE
  show FALSE
  proof -
    have h3: X = TRUE ==> FALSE
    proof -
      assume h4: X = TRUE
      show FALSE
      proof (rule h1 [OF h4])
        show P(TRUE) by (rule subst [of X TRUE P, OF h4 main])
      qed
    qed
    have h5: X = FALSE ==> FALSE
    proof -
```

```
    assume h6: X = FALSE
    show FALSE
    proof (rule h2 [OF h6])
      show P(FALSE) by (rule subst [of X FALSE P, OF h6 main])
    qed
  qed
  show FALSE using isbool h3 h5 by blast
 qed
qed

lemma zenon-boolcase-not :
 P(~A) ==>
  ((~A) = TRUE ==> P(TRUE) ==> FALSE) ==>
  ((~A) = FALSE ==> P(FALSE) ==> FALSE) ==>
  FALSE
proof -
 have h0: (~A) = TRUE | (~A) = FALSE
 by blast
 show
   P(~A) ==>
    ((~A) = TRUE ==> P(TRUE) ==> FALSE) ==>
    ((~A) = FALSE ==> P(FALSE) ==> FALSE) ==>
    FALSE
 by (rule zenon-boolcase [OF h0])
qed

lemma zenon-boolcase-and :
 P(A & B) ==>
  ((A & B) = TRUE ==> P(TRUE) ==> FALSE) ==>
  ((A & B) = FALSE ==> P(FALSE) ==> FALSE) ==>
  FALSE
proof -
 have h0: (A & B) = TRUE | (A & B) = FALSE
 by blast
 show
   P(A & B) ==>
    ((A & B) = TRUE ==> P(TRUE) ==> FALSE) ==>
    ((A & B) = FALSE ==> P(FALSE) ==> FALSE) ==>
    FALSE
 by (rule zenon-boolcase [OF h0])
qed

lemma zenon-boolcase-or :
 P(A | B) ==>
  ((A | B) = TRUE ==> P(TRUE) ==> FALSE) ==>
  ((A | B) = FALSE ==> P(FALSE) ==> FALSE) ==>
  FALSE
proof -
 have h0: (A | B) = TRUE | (A | B) = FALSE
```

by *blast*
show
P(*A* | *B*) ==>
((*A* | *B*) = *TRUE* ==> *P*(*TRUE*) ==> *FALSE*) ==>
((*A* | *B*) = *FALSE* ==> *P*(*FALSE*) ==> *FALSE*) ==>
FALSE
by (*rule zenon-boolcase* [*OF h0*])
qed

lemma *zenon-boolcase-imply* :
P(*A* => *B*) ==>
((*A* => *B*) = *TRUE* ==> *P*(*TRUE*) ==> *FALSE*) ==>
((*A* => *B*) = *FALSE* ==> *P*(*FALSE*) ==> *FALSE*) ==>
FALSE
proof −
have *h0*: (*A* => *B*) = *TRUE* | (*A* => *B*) = *FALSE*
by *blast*
show
P(*A* => *B*) ==>
((*A* => *B*) = *TRUE* ==> *P*(*TRUE*) ==> *FALSE*) ==>
((*A* => *B*) = *FALSE* ==> *P*(*FALSE*) ==> *FALSE*) ==>
FALSE
by (*rule zenon-boolcase* [*OF h0*])
qed

lemma *zenon-boolcase-equiv* :
P(*A* <=> *B*) ==>
((*A* <=> *B*) = *TRUE* ==> *P*(*TRUE*) ==> *FALSE*) ==>
((*A* <=> *B*) = *FALSE* ==> *P*(*FALSE*) ==> *FALSE*) ==>
FALSE
proof −
have *h0*: (*A* <=> *B*) = *TRUE* | (*A* <=> *B*) = *FALSE*
by *blast*
show
P(*A* <=> *B*) ==>
((*A* <=> *B*) = *TRUE* ==> *P*(*TRUE*) ==> *FALSE*) ==>
((*A* <=> *B*) = *FALSE* ==> *P*(*FALSE*) ==> *FALSE*) ==>
FALSE
by (*rule zenon-boolcase* [*OF h0*])
qed

lemma *zenon-boolcase-equal* :
P(*A* = *B*) ==>
((*A* = *B*) = *TRUE* ==> *P*(*TRUE*) ==> *FALSE*) ==>
((*A* = *B*) = *FALSE* ==> *P*(*FALSE*) ==> *FALSE*) ==>
FALSE
proof −
have *h0*: (*A* = *B*) = *TRUE* | (*A* = *B*) = *FALSE*
by *blast*

show
P(*A* = *B*) ==>
((*A* = *B*) = *TRUE* ==> *P*(*TRUE*) ==> *FALSE*) ==>
((*A* = *B*) = *FALSE* ==> *P*(*FALSE*) ==> *FALSE*) ==>
FALSE
by (*rule zenon-boolcase* [*OF h0*])
qed

lemma *zenon-boolcase-all* :
P(*All* (*Q*)) ==>
(*All* (*Q*) = *TRUE* ==> *P*(*TRUE*) ==> *FALSE*) ==>
(*All* (*Q*) = *FALSE* ==> *P*(*FALSE*) ==> *FALSE*) ==>
FALSE
proof −
have *h0*: *All* (*Q*) = *TRUE* | *All* (*Q*) = *FALSE*
by *blast*
show
P(*All* (*Q*)) ==>
((*All* (*Q*)) = *TRUE* ==> *P*(*TRUE*) ==> *FALSE*) ==>
((*All* (*Q*)) = *FALSE* ==> *P*(*FALSE*) ==> *FALSE*) ==>
FALSE
by (*rule zenon-boolcase* [*OF h0*])
qed

lemma *zenon-boolcase-ex* :
P(*Ex* (*Q*)) ==>
(*Ex* (*Q*) = *TRUE* ==> *P*(*TRUE*) ==> *FALSE*) ==>
(*Ex* (*Q*) = *FALSE* ==> *P*(*FALSE*) ==> *FALSE*) ==>
FALSE
proof −
have *h0*: *Ex* (*Q*) = *TRUE* | *Ex* (*Q*) = *FALSE*
by *blast*
show
P(*Ex* (*Q*)) ==>
((*Ex* (*Q*)) = *TRUE* ==> *P*(*TRUE*) ==> *FALSE*) ==>
((*Ex* (*Q*)) = *FALSE* ==> *P*(*FALSE*) ==> *FALSE*) ==>
FALSE
by (*rule zenon-boolcase* [*OF h0*])
qed

lemma *zenon-iftrue* : *P* (*IF TRUE THEN a ELSE b*) ==> (*P* (*a*) ==> *FALSE*)
==> *FALSE*
by *auto*

lemma *zenon-iftrue-0* : *P* (*IF TRUE THEN a ELSE b*) ==> *P* (*a*)
using *zenon-iftrue* **by** *auto*

lemma *zenon-iffalse* :
P (*IF FALSE THEN a ELSE b*) ==> (*P* (*b*) ==> *FALSE*) ==> *FALSE*

```
by auto

lemma zenon-iffalse-0 : P (IF FALSE THEN a ELSE b) ==> P (b)
using zenon-iffalse by auto

lemma zenon-ifthenelse :
  P (IF c THEN a ELSE b) ==>
   (c ==> P (a) ==> FALSE) ==>
   (~c ==> P (b) ==> FALSE) ==>
     FALSE
proof -
  assume main: P (IF c THEN a ELSE b)
  assume h1: c ==> P(a) ==> FALSE
  have h1x: c ==> ~P(a) using h1 by auto
  assume h2: ~c ==> P(b) ==> FALSE
  have h2x: ~c ==> ~P(b) using h2 by auto
  have h3: ~P (IF c THEN a ELSE b)
  by (rule condI [of c λ x . ~P (x), OF h1x h2x])
  show FALSE
  by (rule notE [OF h3 main])
qed

lemma zenon-trueistrue : P (A) ==> A ==> (P (TRUE) ==> FALSE) ==>
FALSE
by auto

lemma zenon-trueistrue-0 : P (A) ==> A ==> P (TRUE)
by auto

lemma zenon-eq-x-true : x = TRUE ==> (x ==> FALSE) ==> FALSE
by blast

lemma zenon-eq-x-true-0 : x = TRUE ==> x
by blast

lemma zenon-eq-true-x : TRUE = x ==> (x ==> FALSE) ==> FALSE
by blast

lemma zenon-eq-true-x-0 : TRUE = x ==> x
by blast

lemma zenon-noteq-x-true : x ~= TRUE ==> (~x ==> FALSE) ==> FALSE
using eqTrueI by auto

lemma zenon-noteq-x-true-0 : x ~= TRUE ==> ~x
using zenon-noteq-x-true by blast

lemma zenon-noteq-true-x : TRUE ~= x ==> (~x ==> FALSE) ==> FALSE
using eqTrueI by auto
```

lemma *zenon-noteq-true-x-0* : *TRUE* ~= *x* ==> ~*x*
using *zenon-noteq-x-true* **by** *blast*

lemma *zenon-eq-x-false* : *x* = *FALSE* ==> (~*x* ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-eq-x-false-0* : *x* = *FALSE* ==> ~*x*
by *blast*

lemma *zenon-eq-false-x* : *FALSE* = *x* ==> (~*x* ==> *FALSE*) ==> *FALSE*
by *blast*

lemma *zenon-eq-false-x-0* : *FALSE* = *x* ==> ~*x*
by *blast*

lemma *zenon-notisafcn-fcn* : ~*isAFcn* (*Fcn* (*s*,*l*)) ==> *FALSE*
by *blast*

lemma *zenon-notisafcn-except* : ~*isAFcn* (*except* (*f*,*v*,*e*)) ==> *FALSE*
by *blast*

lemma *zenon-notisafcn-onearg* : ~*isAFcn* (*oneArg* (*e1*,*e2*)) ==> *FALSE*
by *blast*

lemma *zenon-notisafcn-extend* : ~*isAFcn* (*extend* (*f*,*g*)) ==> *FALSE*
by *blast*

lemma *zenon-domain-except* :
  P (*DOMAIN* (*except* (*f*, *v*, *e*))) ==>
   (*P* (*DOMAIN f*) ==> *FALSE*) ==>
   FALSE
proof –
  assume *main*: *P* (*DOMAIN* (*except* (*f*, *v*, *e*)))
  assume *h1*: *P* (*DOMAIN f*) ==> *FALSE*
  show *FALSE*
  proof (*rule h1*)
    show *P* (*DOMAIN f*)
    using *main domainExcept* **by** *auto*
  qed
qed

lemma *zenon-domain-except-0* : *P* (*DOMAIN* (*except* (*f*, *v*, *e*))) ==> *P* (*DOMAIN*
f)
proof –
  assume *main*: *P* (*DOMAIN* (*except* (*f*, *v*, *e*)))
  show *P* (*DOMAIN f*)
  proof (*rule zenon-nnpp*)
    assume *h1*: ~ (*P* (*DOMAIN f*))

```
    show FALSE
    proof (rule zenon-domain-except)
      show P (DOMAIN (except (f, v, e))) by (rule main)
    next
      show P (DOMAIN f) ==> FALSE using h1 by blast
    qed
  qed
qed

lemma zenon-domain-fcn-0 : P (DOMAIN (Fcn (S, l))) ==> P (S)
using DOMAIN by auto



lemma zenon-in-product-i :
  assumes a: p \in Product (s)
  and b: isASeq(s)
  and c: i \in 1 .. Len (s)
  shows p[i] \in s[i]
proof (rule inProductE [OF a b])
  assume h1: isASeq(p)
  assume h2: Len(p) = Len(s)
  assume h3: p \in [1 .. Len (s) -> UNION Range (s)]
  assume h4: ALL k in 1 .. Len (s) : p[k] \in s[k]
  show p[i] \in s[i] (is ?c)
  proof (rule bAllE [where x = i, OF h4])
    assume h5: i \notin 1 .. Len(s)
    show ?c
    using c h5 by blast
  next
    assume h5: ?c
    show ?c
    by (rule h5)
  qed
qed



lemma zenon-stringdiffll : s ~= t ==> e = s ==> f = t ==> e ~= f
by auto

lemma zenon-stringdifflr : s ~= t ==> e = s ==> t = f ==> e ~= f
by auto

lemma zenon-stringdiffrl : s ~= t ==> s = e ==> f = t ==> e ~= f
by auto

lemma zenon-stringdiffrr : s ~= t ==> s = e ==> t = f ==> e ~= f
by auto
```

```
definition zenon-sa :: [c, c] ⇒ c
where zenon-sa (s, e) ≡ IF isASeq (s) THEN Append (s, e) ELSE <<e>>

lemma zenon-sa-seq: isASeq (zenon-sa (s, e))
  by (simp add: zenon-sa-def, rule disjE [of isASeq(s) ~isASeq(s)],
     rule excluded-middle, simp+)

lemma zenon-sa-1 : Append (<<>>, e) = zenon-sa (<<>>, e)
by (auto simp add: zenon-sa-def)

lemma zenon-sa-2 :
  Append (zenon-sa (s, e1), e2) = zenon-sa (zenon-sa (s, e1), e2)
using zenon-sa-seq by (auto simp add: zenon-sa-def)

lemma zenon-sa-diff-0a :
  zenon-sa (zenon-sa (s1, e1), e2) ~= zenon-sa (<<>>, f2)
using zenon-sa-def zenon-sa-seq by auto

lemma zenon-sa-diff-0b :
  zenon-sa (<<>>, f2) ~= zenon-sa (zenon-sa (s1, e1), e2)
using zenon-sa-def zenon-sa-seq by auto

lemma zenon-sa-diff-1 :
  assumes h0: e ~= f
  shows zenon-sa (<<>>, e) ~= zenon-sa (<<>>, f)
using zenon-sa-def h0 by auto

lemma zenon-sa-diff-2 :
  assumes h0: zenon-sa (e, s1) ~= zenon-sa (f, t1)
  shows zenon-sa (zenon-sa (e, s1), s2) ~= zenon-sa (zenon-sa (f, t1), s2)
using zenon-sa-def zenon-sa-seq h0 by auto

lemma zenon-sa-diff-3 :
  assumes h0: s2 ~= t2
  shows zenon-sa (zenon-sa (e, s1), s2) ~= zenon-sa (zenon-sa (f, t1), t2)
using zenon-sa-def zenon-sa-seq h0 by auto


lemma zenon-in-nat-0 : ~(0 \in Nat) ==> FALSE
by blast

lemma zenon-in-nat-1 : ~(1 \in Nat) ==> FALSE
by blast

lemma zenon-in-nat-2 : ~(2 \in Nat) ==> FALSE
by blast
```

```
lemma zenon-in-nat-succ :
  ~(Succ[x] \in Nat) ==> (~(x \in Nat) ==> FALSE) ==> FALSE
by blast

lemma zenon-in-nat-succ-0 : ~(Succ[x] \in Nat) ==> ~(x \in Nat)
by blast



lemma zenon-in-recordset-field :
  assumes a: r \in EnumFuncSet (doms, rngs)
  and b: i \in DOMAIN (doms)
  shows r[doms[i]] \in rngs[i]
proof (rule EnumFuncSetE [OF a])
  assume h1: r \in [Range(doms) -> UNION Range(rngs)]
  assume h2: ALL i in DOMAIN doms : r[doms[i]] \in rngs[i]
  show r[doms[i]] \in rngs[i] (is ?c)
  proof (rule bAllE [where x = i, OF h2])
    assume h3: i \notin DOMAIN doms
    show ?c
    using b h3 by blast
  next
    assume h3: ?c
    show ?c
    by (rule h3)
  qed
qed

lemma zenon-recfield-1:
  l \in DOMAIN r & r[l] = v ==>
   l \in DOMAIN (r @@ m :> w) & (r @@ m :> w)[l] = v
by auto

lemma zenon-recfield-2:
  l \notin DOMAIN r ==>
   l \in DOMAIN (r @@ l :> v) & (r @@ l :> v)[l] = v
by auto

lemma zenon-recfield-2b: l \in DOMAIN (l :> v) & (l :> v)[l] = v by auto

lemma zenon-recfield-3:
  l \notin DOMAIN r ==> l ~= m ==> l \notin DOMAIN (r @@ m :> v)
by auto

lemma zenon-recfield-3b: l ~= m ==> l \notin DOMAIN (m :> v) by auto



lemma zenon-range-1 : isASeq (<<>>) & {} = Range (<<>>) by auto
```

lemma zenon-range-2 :
 assumes h: (isASeq (s) & a = Range (s))
 shows (isASeq (Append (s, x)) & addElt (x, a) = Range (Append (s, x)))
using h **by** auto

lemma zenon-set-rev-1 : a = {} \cup c ==> c = a **by** auto

lemma zenon-set-rev-2 : a = addElt (x, b) \cup c ==> a = b \cup addElt (x, c)
by auto

lemma zenon-set-rev-3 : a = c ==> a = c \cup {} **by** auto

lemma zenon-tuple-dom-1 :
 isASeq (<<>>) & isASeq (<<>>) & DOMAIN <<>> = DOMAIN <<>>
by auto

lemma zenon-tuple-dom-2 :
 isASeq (s) & isASeq (t) & DOMAIN s = DOMAIN t ==>
  isASeq (Append (s, x)) & isASeq (Append (t, y))
  & DOMAIN (Append (s, x)) = DOMAIN (Append (t, y))
by auto

lemma zenon-tuple-dom-3 :
 isASeq (s) & isASeq (t) & DOMAIN s = DOMAIN t ==> DOMAIN s = DOMAIN t
by auto

lemma zenon-all-rec-1 : ALL i in {} : r[doms[i]] \in rngs[i] **by** auto

lemma zenon-all-rec-2 :
 ALL i in s : r[doms[i]] \in rngs[i] ==>
  r[doms[j]] \in rngs[j] ==>
  ALL i in addElt (j, s) : r[doms[i]] \in rngs[i]
by auto

lemma zenon-tuple-acc-1 :
 isASeq (r) ==> Len (r) = n ==> x = Append (r, x) [Succ[n]] **by** auto

lemma zenon-tuple-acc-2 :
 isASeq (r) ==> k \in Nat ==> 0 < k ==> k ≤ Len (r) ==>
  x = r[k] ==> x = Append (r, e) [k]
by auto

definition zenon-ss :: c ⇒ c
where zenon-ss (n) ≡ IF n \in Nat THEN Succ[n] ELSE 1

lemma zenon-ss-nat : zenon-ss(x) \in Nat **by** (auto simp add: zenon-ss-def)

lemma *zenon-ss-1* : *Succ*[*0*] = *zenon-ss*(*0*) **by** (*auto simp add*: *zenon-ss-def*)

lemma *zenon-ss-2* : *Succ*[*zenon-ss*(*x*)] = *zenon-ss*(*zenon-ss*(*x*)) **by** (*auto simp add*: *zenon-ss-def*)

lemma *zenon-zero-lt* : *0* < *zenon-ss*(*x*)
by (*simp add*: *zenon-ss-def*, *rule disjE* [*of x \in Nat x \notin Nat*], *rule excluded-middle*, *simp+*)

lemma *zenon-ss-le-sa-1* : *zenon-ss*(*0*) <= *Len* (*zenon-sa* (*s*, *x*))
by (*auto simp add*: *zenon-ss-def zenon-sa-def*, *rule disjE* [*of isASeq* (*s*) ~*isASeq* (*s*)], *rule excluded-middle*, *simp+*)

lemma *zenon-ss-le-sa-2* :
fixes *x y z*
assumes *h0*: *zenon-ss* (*x*) <= *Len* (*zenon-sa* (*s*, *y*))
shows *zenon-ss* (*zenon-ss* (*x*)) <= *Len* (*zenon-sa* (*zenon-sa* (*s*, *y*), *z*))
proof −
have *h1*: *Succ* [*Len* (*zenon-sa* (*s*, *y*))] = *Len* (*zenon-sa* (*zenon-sa* (*s*, *y*), *z*))
using *zenon-sa-seq* **by** (*auto simp add*: *zenon-sa-def*)

have *h2*: *Len* (*zenon-sa* (*s*, *y*)) *\in Nat*
using *zenon-sa-seq* **by** (*rule LenInNat*)
have *h3*: *Succ* [*zenon-ss* (*x*)] $\leq$ *Succ* [*Len* (*zenon-sa* (*s*, *y*))]
using *zenon-ss-nat h2 h0* **by** (*simp only*: *nat-Succ-leq-Succ*)
show *?thesis*
using *h3* **by** (*auto simp add*: *zenon-ss-2 h1*)
qed

lemma *zenon-dom-app-1* : *isASeq* (<<>>) & *0* = *Len* (<<>>) & {} = *1*..*Len*(<<>>)
by *auto*

lemma *zenon-dom-app-2* :
assumes *h*: *isASeq* (*s*) & *n* = *Len* (*s*) & *x* = *1* .. *Len* (*s*)
shows *isASeq* (*Append* (*s*, *y*)) & *Succ*[*n*] = *Len* (*Append* (*s*, *y*))
& *addElt* (*Succ*[*n*], *x*) = *1* .. *Len* (*Append* (*s*, *y*)) (**is** *?a* & *?b* & *?c*)
using *h* **by** *auto*

lemma *zenon-inrecordsetI1* :
fixes *r l1x s1x*
assumes *hfcn*: *isAFcn* (*r*)
assumes *hdom*: *DOMAIN r* = {*l1x*}
assumes *h1*: *r*[*l1x*] *\in s1x*
shows *r \in* [*l1x* : *s1x*]
proof −

```
let ?doms = <<l1x>>
let ?domset = {l1x}
let ?domsetrev = {l1x}
let ?rngs = <<s1x>>
let ?n0n = 0
let ?n1n = Succ[?n0n]
let ?indices = {?n1n}
have hdomx : ?domsetrev = DOMAIN r
by (rule zenon-set-rev-1, (rule zenon-set-rev-2)+, rule zenon-set-rev-3,
   rule hdom)
show r \in EnumFuncSet (?doms, ?rngs)
proof (rule EnumFuncSetI [of r, OF hfcn])
  have hx1: isASeq (?doms) & ?domsetrev = Range (?doms)
  by ((rule zenon-range-2)+, rule zenon-range-1)
  have hx2: ?domsetrev = Range (?doms)
  by (rule conjD2 [OF hx1])
  show DOMAIN r = Range (?doms)
  by (rule subst [where P = %x. x = Range (?doms), OF hdomx hx2])
next
  show DOMAIN ?rngs = DOMAIN ?doms
  by (rule zenon-tuple-dom-3, (rule zenon-tuple-dom-2)+,
     rule zenon-tuple-dom-1)
next
  have hdomseq: isASeq (?doms) by auto
  have hindx: isASeq (?doms) & ?n1n = Len (?doms)
             & ?indices = DOMAIN ?doms
  by (simp only: DomainSeqLen [OF hdomseq], (rule zenon-dom-app-2)+,
     rule zenon-dom-app-1)
 have hind: ?indices = DOMAIN ?doms by (rule conjE [OF hindx], elim conjE)
  show ALL i in DOMAIN ?doms : r[?doms[i]] \in ?rngs[i]
  proof (rule subst [OF hind], (rule zenon-all-rec-2)+, rule zenon-all-rec-1)

    have hn: l1x = ?doms[?n1n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
          rule zenon-zero-lt,
          simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
          ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
         )+)?,
        rule zenon-tuple-acc-1, auto)
    have hs: s1x = ?rngs[?n1n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
          rule zenon-zero-lt,
          simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
          ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
         )+)?,
        rule zenon-tuple-acc-1, auto)
    show r[?doms[?n1n]] \in ?rngs[?n1n]
    by (rule subst[OF hs], rule subst[OF hn], rule h1)
```

```
    qed
  qed
qed

lemma zenon-inrecordsetI2 :
  fixes r l1x s1x l2x s2x
  assumes hfcn: isAFcn (r)
  assumes hdom: DOMAIN r = {l1x, l2x}
  assumes h1: r[l1x] \in s1x
  assumes h2: r[l2x] \in s2x
  shows r \in [l1x : s1x, l2x : s2x]
proof -
  let ?doms = <<l1x, l2x>>
  let ?domset = {l1x, l2x}
  let ?domsetrev = {l2x, l1x}
  let ?rngs = <<s1x, s2x>>
  let ?n0n = 0
  let ?n1n = Succ[?n0n]
  let ?n2n = Succ[?n1n]
  let ?indices = {?n2n, ?n1n}
  have hdomx : ?domsetrev = DOMAIN r
  by (rule zenon-set-rev-1, (rule zenon-set-rev-2)+, rule zenon-set-rev-3,
      rule hdom)
  show r \in EnumFuncSet (?doms, ?rngs)
  proof (rule EnumFuncSetI [of r, OF hfcn])
    have hx1: isASeq (?doms) & ?domsetrev = Range (?doms)
    by ((rule zenon-range-2)+, rule zenon-range-1)
    have hx2: ?domsetrev = Range (?doms)
    by (rule conjD2 [OF hx1])
    show DOMAIN r = Range (?doms)
    by (rule subst [where P = %x. x = Range (?doms), OF hdomx hx2])
  next
    show DOMAIN ?rngs = DOMAIN ?doms
    by (rule zenon-tuple-dom-3, (rule zenon-tuple-dom-2)+,
        rule zenon-tuple-dom-1)
  next
    have hdomseq: isASeq (?doms) by auto
    have hindx: isASeq (?doms) & ?n2n = Len (?doms)
                & ?indices = DOMAIN ?doms
    by (simp only: DomainSeqLen [OF hdomseq], (rule zenon-dom-app-2)+,
        rule zenon-dom-app-1)
   have hind: ?indices = DOMAIN ?doms by (rule conjE [OF hindx], elim conjE)
    show ALL i in DOMAIN ?doms : r[?doms[i]] \in ?rngs[i]
    proof (rule subst [OF hind], (rule zenon-all-rec-2)+, rule zenon-all-rec-1)

      have hn: l1x = ?doms[?n1n]
      by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
            rule zenon-zero-lt,
            simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
```

```
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
    have hs: s1x = ?rngs[?n1n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
    show r[?doms[?n1n]] \in ?rngs[?n1n]
    by (rule subst[OF hs], rule subst[OF hn], rule h1)
  next
    have hn: l2x = ?doms[?n2n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
    have hs: s2x = ?rngs[?n2n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
    show r[?doms[?n2n]] \in ?rngs[?n2n]
    by (rule subst[OF hs], rule subst[OF hn], rule h2)

  qed
 qed
qed

lemma zenon-inrecordsetI3 :
 fixes r l1x s1x l2x s2x l3x s3x
 assumes hfcn: isAFcn (r)
 assumes hdom: DOMAIN r = {l1x, l2x, l3x}
 assumes h1: r[l1x] \in s1x
 assumes h2: r[l2x] \in s2x
 assumes h3: r[l3x] \in s3x
 shows r \in [l1x : s1x, l2x : s2x, l3x : s3x]
proof −
 let ?doms = <<l1x, l2x, l3x>>
 let ?domset = {l1x, l2x, l3x}
 let ?domsetrev = {l3x, l2x, l1x}
 let ?rngs = <<s1x, s2x, s3x>>
 let ?n0n = 0
 let ?n1n = Succ[?n0n]
```

```
let ?n2n = Succ[?n1n]
let ?n3n = Succ[?n2n]
let ?indices = {?n3n, ?n2n, ?n1n}
have hdomx : ?domsetrev = DOMAIN r
by (rule zenon-set-rev-1, (rule zenon-set-rev-2)+, rule zenon-set-rev-3,
    rule hdom)
show r \in EnumFuncSet (?doms, ?rngs)
proof (rule EnumFuncSetI [of r, OF hfcn])
  have hx1: isASeq (?doms) & ?domsetrev = Range (?doms)
  by ((rule zenon-range-2)+, rule zenon-range-1)
  have hx2: ?domsetrev = Range (?doms)
  by (rule conjD2 [OF hx1])
  show DOMAIN r = Range (?doms)
  by (rule subst [where P = %x. x = Range (?doms), OF hdomx hx2])
next
  show DOMAIN ?rngs = DOMAIN ?doms
  by (rule zenon-tuple-dom-3, (rule zenon-tuple-dom-2)+,
      rule zenon-tuple-dom-1)
next
  have hdomseq: isASeq (?doms) by auto
  have hindx: isASeq (?doms) & ?n3n = Len (?doms)
              & ?indices = DOMAIN ?doms
  by (simp only: DomainSeqLen [OF hdomseq], (rule zenon-dom-app-2)+,
      rule zenon-dom-app-1)
 have hind: ?indices = DOMAIN ?doms by (rule conjE [OF hindx], elim conjE)
  show ALL i in DOMAIN ?doms : r[?doms[i]] \in ?rngs[i]
  proof (rule subst [OF hind], (rule zenon-all-rec-2)+, rule zenon-all-rec-1)

    have hn: l1x = ?doms[?n1n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
          rule zenon-zero-lt,
          simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
          ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
         )+)?,
        rule zenon-tuple-acc-1, auto)
    have hs: s1x = ?rngs[?n1n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
          rule zenon-zero-lt,
          simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
          ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
         )+)?,
        rule zenon-tuple-acc-1, auto)
    show r[?doms[?n1n]] \in ?rngs[?n1n]
    by (rule subst[OF hs], rule subst[OF hn], rule h1)
  next
    have hn: l2x = ?doms[?n2n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
          rule zenon-zero-lt,
          simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
```

```
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  have hs: s2x = ?rngs[?n2n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  show r[?doms[?n2n]] \in ?rngs[?n2n]
  by (rule subst[OF hs], rule subst[OF hn], rule h2)
 next
  have hn: l3x = ?doms[?n3n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  have hs: s3x = ?rngs[?n3n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  show r[?doms[?n3n]] \in ?rngs[?n3n]
  by (rule subst[OF hs], rule subst[OF hn], rule h3)

 qed
qed
qed

lemma zenon-inrecordsetI4 :
 fixes r l1x s1x l2x s2x l3x s3x l4x s4x
 assumes hfcn: isAFcn (r)
 assumes hdom: DOMAIN r = {l1x, l2x, l3x, l4x}
 assumes h1: r[l1x] \in s1x
 assumes h2: r[l2x] \in s2x
 assumes h3: r[l3x] \in s3x
 assumes h4: r[l4x] \in s4x
 shows r \in [l1x : s1x, l2x : s2x, l3x : s3x, l4x : s4x]
proof -
 let ?doms = <<l1x, l2x, l3x, l4x>>
 let ?domset = {l1x, l2x, l3x, l4x}
 let ?domsetrev = {l4x, l3x, l2x, l1x}
 let ?rngs = <<s1x, s2x, s3x, s4x>>
 let ?n0n = 0
```

```
let ?n1n = Succ[?n0n]
let ?n2n = Succ[?n1n]
let ?n3n = Succ[?n2n]
let ?n4n = Succ[?n3n]
let ?indices = {?n4n, ?n3n, ?n2n, ?n1n}
have hdomx : ?domsetrev = DOMAIN r
by (rule zenon-set-rev-1, (rule zenon-set-rev-2)+, rule zenon-set-rev-3,
    rule hdom)
show r \in EnumFuncSet (?doms, ?rngs)
proof (rule EnumFuncSetI [of r, OF hfcn])
  have hx1: isASeq (?doms) & ?domsetrev = Range (?doms)
  by ((rule zenon-range-2)+, rule zenon-range-1)
  have hx2: ?domsetrev = Range (?doms)
  by (rule conjD2 [OF hx1])
  show DOMAIN r = Range (?doms)
  by (rule subst [where P = %x. x = Range (?doms), OF hdomx hx2])
next
  show DOMAIN ?rngs = DOMAIN ?doms
  by (rule zenon-tuple-dom-3, (rule zenon-tuple-dom-2)+,
      rule zenon-tuple-dom-1)
next
  have hdomseq: isASeq (?doms) by auto
  have hindx: isASeq (?doms) & ?n4n = Len (?doms)
              & ?indices = DOMAIN ?doms
  by (simp only: DomainSeqLen [OF hdomseq], (rule zenon-dom-app-2)+,
      rule zenon-dom-app-1)
 have hind: ?indices = DOMAIN ?doms by (rule conjE [OF hindx], elim conjE)
  show ALL i in DOMAIN ?doms : r[?doms[i]] \in ?rngs[i]
  proof (rule subst [OF hind], (rule zenon-all-rec-2)+, rule zenon-all-rec-1)

    have hn: l1x = ?doms[?n1n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
          rule zenon-zero-lt,
          simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
          ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
         )+)?,
        rule zenon-tuple-acc-1, auto)
    have hs: s1x = ?rngs[?n1n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
          rule zenon-zero-lt,
          simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
          ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
         )+)?,
        rule zenon-tuple-acc-1, auto)
    show r[?doms[?n1n]] \in ?rngs[?n1n]
    by (rule subst[OF hs], rule subst[OF hn], rule h1)
  next
    have hn: l2x = ?doms[?n2n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
```

```
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  have hs: s2x = ?rngs[?n2n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  show r[?doms[?n2n]] \in ?rngs[?n2n]
  by (rule subst[OF hs], rule subst[OF hn], rule h2)
next
  have hn: l3x = ?doms[?n3n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  have hs: s3x = ?rngs[?n3n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  show r[?doms[?n3n]] \in ?rngs[?n3n]
  by (rule subst[OF hs], rule subst[OF hn], rule h3)
next
  have hn: l4x = ?doms[?n4n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  have hs: s4x = ?rngs[?n4n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  show r[?doms[?n4n]] \in ?rngs[?n4n]
  by (rule subst[OF hs], rule subst[OF hn], rule h4)
```

```
    qed
  qed
qed

lemma zenon-inrecordsetI5 :
  fixes r l1x s1x l2x s2x l3x s3x l4x s4x l5x s5x
  assumes hfcn: isAFcn (r)
  assumes hdom: DOMAIN r = {l1x, l2x, l3x, l4x, l5x}
  assumes h1: r[l1x] \in s1x
  assumes h2: r[l2x] \in s2x
  assumes h3: r[l3x] \in s3x
  assumes h4: r[l4x] \in s4x
  assumes h5: r[l5x] \in s5x
  shows r \in [l1x : s1x, l2x : s2x, l3x : s3x, l4x : s4x, l5x : s5x]
proof -
  let ?doms = <<l1x, l2x, l3x, l4x, l5x>>
  let ?domset = {l1x, l2x, l3x, l4x, l5x}
  let ?domsetrev = {l5x, l4x, l3x, l2x, l1x}
  let ?rngs = <<s1x, s2x, s3x, s4x, s5x>>
  let ?n0n = 0
  let ?n1n = Succ[?n0n]
  let ?n2n = Succ[?n1n]
  let ?n3n = Succ[?n2n]
  let ?n4n = Succ[?n3n]
  let ?n5n = Succ[?n4n]
  let ?indices = {?n5n, ?n4n, ?n3n, ?n2n, ?n1n}
  have hdomx : ?domsetrev = DOMAIN r
  by (rule zenon-set-rev-1, (rule zenon-set-rev-2)+, rule zenon-set-rev-3,
      rule hdom)
  show r \in EnumFuncSet (?doms, ?rngs)
  proof (rule EnumFuncSetI [of r, OF hfcn])
    have hx1: isASeq (?doms) & ?domsetrev = Range (?doms)
    by ((rule zenon-range-2)+, rule zenon-range-1)
    have hx2: ?domsetrev = Range (?doms)
    by (rule conjD2 [OF hx1])
    show DOMAIN r = Range (?doms)
    by (rule subst [where P = %x. x = Range (?doms), OF hdomx hx2])
  next
    show DOMAIN ?rngs = DOMAIN ?doms
    by (rule zenon-tuple-dom-3, (rule zenon-tuple-dom-2)+,
        rule zenon-tuple-dom-1)
  next
    have hdomseq: isASeq (?doms) by auto
    have hindx: isASeq (?doms) & ?n5n = Len (?doms)
                & ?indices = DOMAIN ?doms
    by (simp only: DomainSeqLen [OF hdomseq], (rule zenon-dom-app-2)+,
        rule zenon-dom-app-1)
   have hind: ?indices = DOMAIN ?doms by (rule conjE [OF hindx], elim conjE)
    show ALL i in DOMAIN ?doms : r[?doms[i]] \in ?rngs[i]
```

proof (*rule subst* [*OF hind*], (*rule zenon-all-rec-2*)+, *rule zenon-all-rec-1*)

have *hn*: *l1x* = *?doms*[*?n1n*]
by (((*rule zenon-tuple-acc-2*, *safe*, *simp only*: *zenon-ss-1 zenon-ss-2*,
rule zenon-zero-lt,
simp only: *zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2*,
((*rule zenon-ss-le-sa-2*)+)?, *rule zenon-ss-le-sa-1*
)+)?,
rule zenon-tuple-acc-1, *auto*)
have *hs*: *s1x* = *?rngs*[*?n1n*]
by (((*rule zenon-tuple-acc-2*, *safe*, *simp only*: *zenon-ss-1 zenon-ss-2*,
rule zenon-zero-lt,
simp only: *zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2*,
((*rule zenon-ss-le-sa-2*)+)?, *rule zenon-ss-le-sa-1*
)+)?,
rule zenon-tuple-acc-1, *auto*)
show *r*[*?doms*[*?n1n*]] \in *?rngs*[*?n1n*]
by (*rule subst*[*OF hs*], *rule subst*[*OF hn*], *rule h1*)
next
have *hn*: *l2x* = *?doms*[*?n2n*]
by (((*rule zenon-tuple-acc-2*, *safe*, *simp only*: *zenon-ss-1 zenon-ss-2*,
rule zenon-zero-lt,
simp only: *zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2*,
((*rule zenon-ss-le-sa-2*)+)?, *rule zenon-ss-le-sa-1*
)+)?,
rule zenon-tuple-acc-1, *auto*)
have *hs*: *s2x* = *?rngs*[*?n2n*]
by (((*rule zenon-tuple-acc-2*, *safe*, *simp only*: *zenon-ss-1 zenon-ss-2*,
rule zenon-zero-lt,
simp only: *zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2*,
((*rule zenon-ss-le-sa-2*)+)?, *rule zenon-ss-le-sa-1*
)+)?,
rule zenon-tuple-acc-1, *auto*)
show *r*[*?doms*[*?n2n*]] \in *?rngs*[*?n2n*]
by (*rule subst*[*OF hs*], *rule subst*[*OF hn*], *rule h2*)
next
have *hn*: *l3x* = *?doms*[*?n3n*]
by (((*rule zenon-tuple-acc-2*, *safe*, *simp only*: *zenon-ss-1 zenon-ss-2*,
rule zenon-zero-lt,
simp only: *zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2*,
((*rule zenon-ss-le-sa-2*)+)?, *rule zenon-ss-le-sa-1*
)+)?,
rule zenon-tuple-acc-1, *auto*)
have *hs*: *s3x* = *?rngs*[*?n3n*]
by (((*rule zenon-tuple-acc-2*, *safe*, *simp only*: *zenon-ss-1 zenon-ss-2*,
rule zenon-zero-lt,
simp only: *zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2*,
((*rule zenon-ss-le-sa-2*)+)?, *rule zenon-ss-le-sa-1*
)+)?,

```
      rule zenon-tuple-acc-1, auto)
    show r[?doms[?n3n]] \in ?rngs[?n3n]
    by (rule subst[OF hs], rule subst[OF hn], rule h3)
  next
    have hn: l4x = ?doms[?n4n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
         rule zenon-zero-lt,
         simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
         ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
        )+)?,
      rule zenon-tuple-acc-1, auto)
    have hs: s4x = ?rngs[?n4n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
         rule zenon-zero-lt,
         simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
         ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
        )+)?,
      rule zenon-tuple-acc-1, auto)
    show r[?doms[?n4n]] \in ?rngs[?n4n]
    by (rule subst[OF hs], rule subst[OF hn], rule h4)
  next
    have hn: l5x = ?doms[?n5n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
         rule zenon-zero-lt,
         simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
         ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
        )+)?,
      rule zenon-tuple-acc-1, auto)
    have hs: s5x = ?rngs[?n5n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
         rule zenon-zero-lt,
         simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
         ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
        )+)?,
      rule zenon-tuple-acc-1, auto)
    show r[?doms[?n5n]] \in ?rngs[?n5n]
    by (rule subst[OF hs], rule subst[OF hn], rule h5)

  qed
 qed
qed

lemma zenon-inrecordsetI6 :
 fixes r l1x s1x l2x s2x l3x s3x l4x s4x l5x s5x l6x s6x
 assumes hfcn: isAFcn (r)
 assumes hdom: DOMAIN r = {l1x, l2x, l3x, l4x, l5x, l6x}
 assumes h1: r[l1x] \in s1x
 assumes h2: r[l2x] \in s2x
 assumes h3: r[l3x] \in s3x
```

```
  assumes h4: r[l4x] \in s4x
  assumes h5: r[l5x] \in s5x
  assumes h6: r[l6x] \in s6x
  shows r \in [l1x : s1x, l2x : s2x, l3x : s3x, l4x : s4x, l5x : s5x, l6x : s6x]
proof -
  let ?doms = <<l1x, l2x, l3x, l4x, l5x, l6x>>
  let ?domset = {l1x, l2x, l3x, l4x, l5x, l6x}
  let ?domsetrev = {l6x, l5x, l4x, l3x, l2x, l1x}
  let ?rngs = <<s1x, s2x, s3x, s4x, s5x, s6x>>
  let ?n0n = 0
  let ?n1n = Succ[?n0n]
  let ?n2n = Succ[?n1n]
  let ?n3n = Succ[?n2n]
  let ?n4n = Succ[?n3n]
  let ?n5n = Succ[?n4n]
  let ?n6n = Succ[?n5n]
  let ?indices = {?n6n, ?n5n, ?n4n, ?n3n, ?n2n, ?n1n}
  have hdomx : ?domsetrev = DOMAIN r
  by (rule zenon-set-rev-1, (rule zenon-set-rev-2)+, rule zenon-set-rev-3,
      rule hdom)
  show r \in EnumFuncSet (?doms, ?rngs)
  proof (rule EnumFuncSetI [of r, OF hfcn])
    have hx1: isASeq (?doms) & ?domsetrev = Range (?doms)
    by ((rule zenon-range-2)+, rule zenon-range-1)
    have hx2: ?domsetrev = Range (?doms)
    by (rule conjD2 [OF hx1])
    show DOMAIN r = Range (?doms)
    by (rule subst [where P = %x. x = Range (?doms), OF hdomx hx2])
  next
    show DOMAIN ?rngs = DOMAIN ?doms
    by (rule zenon-tuple-dom-3, (rule zenon-tuple-dom-2)+,
        rule zenon-tuple-dom-1)
  next
    have hdomseq: isASeq (?doms) by auto
    have hindx: isASeq (?doms) & ?n6n = Len (?doms)
                & ?indices = DOMAIN ?doms
    by (simp only: DomainSeqLen [OF hdomseq], (rule zenon-dom-app-2)+,
        rule zenon-dom-app-1)
   have hind: ?indices = DOMAIN ?doms by (rule conjE [OF hindx], elim conjE)
    show ALL i in DOMAIN ?doms : r[?doms[i]] \in ?rngs[i]
    proof (rule subst [OF hind], (rule zenon-all-rec-2)+, rule zenon-all-rec-1)

      have hn: l1x = ?doms[?n1n]
      by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
            rule zenon-zero-lt,
            simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
            ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
           )+)?,
          rule zenon-tuple-acc-1, auto)
```

```
  have hs: s1x = ?rngs[?n1n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  show r[?doms[?n1n]] \in ?rngs[?n1n]
  by (rule subst[OF hs], rule subst[OF hn], rule h1)
next
  have hn: l2x = ?doms[?n2n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  have hs: s2x = ?rngs[?n2n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  show r[?doms[?n2n]] \in ?rngs[?n2n]
  by (rule subst[OF hs], rule subst[OF hn], rule h2)
next
  have hn: l3x = ?doms[?n3n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  have hs: s3x = ?rngs[?n3n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  show r[?doms[?n3n]] \in ?rngs[?n3n]
  by (rule subst[OF hs], rule subst[OF hn], rule h3)
next
  have hn: l4x = ?doms[?n4n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
```

```
        )+)?,
       rule zenon-tuple-acc-1, auto)
    have hs: s4x = ?rngs[?n4n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
          rule zenon-zero-lt,
          simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
          ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
        )+)?,
       rule zenon-tuple-acc-1, auto)
    show r[?doms[?n4n]] \in ?rngs[?n4n]
    by (rule subst[OF hs], rule subst[OF hn], rule h4)
  next
    have hn: l5x = ?doms[?n5n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
          rule zenon-zero-lt,
          simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
          ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
        )+)?,
       rule zenon-tuple-acc-1, auto)
    have hs: s5x = ?rngs[?n5n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
          rule zenon-zero-lt,
          simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
          ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
        )+)?,
       rule zenon-tuple-acc-1, auto)
    show r[?doms[?n5n]] \in ?rngs[?n5n]
    by (rule subst[OF hs], rule subst[OF hn], rule h5)
  next
    have hn: l6x = ?doms[?n6n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
          rule zenon-zero-lt,
          simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
          ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
        )+)?,
       rule zenon-tuple-acc-1, auto)
    have hs: s6x = ?rngs[?n6n]
    by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
          rule zenon-zero-lt,
          simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
          ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
        )+)?,
       rule zenon-tuple-acc-1, auto)
    show r[?doms[?n6n]] \in ?rngs[?n6n]
    by (rule subst[OF hs], rule subst[OF hn], rule h6)

  qed
 qed
qed
```

lemma *zenon-inrecordsetI7* :
fixes *r l1x s1x l2x s2x l3x s3x l4x s4x l5x s5x l6x s6x l7x s7x*
assumes *hfcn*: *isAFcn* (*r*)
assumes *hdom*: *DOMAIN r* = {*l1x*, *l2x*, *l3x*, *l4x*, *l5x*, *l6x*, *l7x*}
assumes *h1*: *r*[*l1x*] \in *s1x*
assumes *h2*: *r*[*l2x*] \in *s2x*
assumes *h3*: *r*[*l3x*] \in *s3x*
assumes *h4*: *r*[*l4x*] \in *s4x*
assumes *h5*: *r*[*l5x*] \in *s5x*
assumes *h6*: *r*[*l6x*] \in *s6x*
assumes *h7*: *r*[*l7x*] \in *s7x*
shows *r* \in [*l1x* : *s1x*, *l2x* : *s2x*, *l3x* : *s3x*, *l4x* : *s4x*, *l5x* : *s5x*, *l6x* : *s6x*, *l7x* : *s7x*]
proof −
let *?doms* = <<*l1x*, *l2x*, *l3x*, *l4x*, *l5x*, *l6x*, *l7x*>>
let *?domset* = {*l1x*, *l2x*, *l3x*, *l4x*, *l5x*, *l6x*, *l7x*}
let *?domsetrev* = {*l7x*, *l6x*, *l5x*, *l4x*, *l3x*, *l2x*, *l1x*}
let *?rngs* = <<*s1x*, *s2x*, *s3x*, *s4x*, *s5x*, *s6x*, *s7x*>>
let *?n0n* = *0*
let *?n1n* = *Succ*[*?n0n*]
let *?n2n* = *Succ*[*?n1n*]
let *?n3n* = *Succ*[*?n2n*]
let *?n4n* = *Succ*[*?n3n*]
let *?n5n* = *Succ*[*?n4n*]
let *?n6n* = *Succ*[*?n5n*]
let *?n7n* = *Succ*[*?n6n*]
let *?indices* = {*?n7n*, *?n6n*, *?n5n*, *?n4n*, *?n3n*, *?n2n*, *?n1n*}
have *hdomx* : *?domsetrev* = *DOMAIN r*
by (*rule zenon-set-rev-1*, (*rule zenon-set-rev-2*)+, *rule zenon-set-rev-3*, *rule hdom*)
show *r* \in *EnumFuncSet* (*?doms*, *?rngs*)
proof (*rule EnumFuncSetI* [*of r*, *OF hfcn*])
have *hx1*: *isASeq* (*?doms*) & *?domsetrev* = *Range* (*?doms*)
by ((*rule zenon-range-2*)+, *rule zenon-range-1*)
have *hx2*: *?domsetrev* = *Range* (*?doms*)
by (*rule conjD2* [*OF hx1*])
show *DOMAIN r* = *Range* (*?doms*)
by (*rule subst* [**where** *P* = %*x*. *x* = *Range* (*?doms*), *OF hdomx hx2*])
next
show *DOMAIN ?rngs* = *DOMAIN ?doms*
by (*rule zenon-tuple-dom-3*, (*rule zenon-tuple-dom-2*)+, *rule zenon-tuple-dom-1*)
next
have *hdomseq*: *isASeq* (*?doms*) **by** *auto*
have *hindx*: *isASeq* (*?doms*) & *?n7n* = *Len* (*?doms*) & *?indices* = *DOMAIN ?doms*
by (*simp only*: *DomainSeqLen* [*OF hdomseq*], (*rule zenon-dom-app-2*)+, *rule zenon-dom-app-1*)

```
have hind: ?indices = DOMAIN ?doms by (rule conjE [OF hindx], elim conjE)
 show ALL i in DOMAIN ?doms : r[?doms[i]] \in ?rngs[i]
 proof (rule subst [OF hind], (rule zenon-all-rec-2)+, rule zenon-all-rec-1)

   have hn: l1x = ?doms[?n1n]
   by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
         rule zenon-zero-lt,
         simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
         ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
        )+)?,
       rule zenon-tuple-acc-1, auto)
   have hs: s1x = ?rngs[?n1n]
   by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
         rule zenon-zero-lt,
         simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
         ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
        )+)?,
       rule zenon-tuple-acc-1, auto)
   show r[?doms[?n1n]] \in ?rngs[?n1n]
   by (rule subst[OF hs], rule subst[OF hn], rule h1)
 next
   have hn: l2x = ?doms[?n2n]
   by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
         rule zenon-zero-lt,
         simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
         ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
        )+)?,
       rule zenon-tuple-acc-1, auto)
   have hs: s2x = ?rngs[?n2n]
   by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
         rule zenon-zero-lt,
         simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
         ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
        )+)?,
       rule zenon-tuple-acc-1, auto)
   show r[?doms[?n2n]] \in ?rngs[?n2n]
   by (rule subst[OF hs], rule subst[OF hn], rule h2)
 next
   have hn: l3x = ?doms[?n3n]
   by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
         rule zenon-zero-lt,
         simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
         ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
        )+)?,
       rule zenon-tuple-acc-1, auto)
   have hs: s3x = ?rngs[?n3n]
   by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
         rule zenon-zero-lt,
         simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
```

```
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  show r[?doms[?n3n]] \in ?rngs[?n3n]
  by (rule subst[OF hs], rule subst[OF hn], rule h3)
next
  have hn: l4x = ?doms[?n4n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  have hs: s4x = ?rngs[?n4n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  show r[?doms[?n4n]] \in ?rngs[?n4n]
  by (rule subst[OF hs], rule subst[OF hn], rule h4)
next
  have hn: l5x = ?doms[?n5n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  have hs: s5x = ?rngs[?n5n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  show r[?doms[?n5n]] \in ?rngs[?n5n]
  by (rule subst[OF hs], rule subst[OF hn], rule h5)
next
  have hn: l6x = ?doms[?n6n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  have hs: s6x = ?rngs[?n6n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
```

```
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  show r[?doms[?n6n]] \in ?rngs[?n6n]
  by (rule subst[OF hs], rule subst[OF hn], rule h6)
 next
  have hn: l7x = ?doms[?n7n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  have hs: s7x = ?rngs[?n7n]
  by (((rule zenon-tuple-acc-2, safe, simp only: zenon-ss-1 zenon-ss-2,
        rule zenon-zero-lt,
        simp only: zenon-ss-1 zenon-ss-2 zenon-sa-1 zenon-sa-2,
        ((rule zenon-ss-le-sa-2)+)?, rule zenon-ss-le-sa-1
       )+)?,
      rule zenon-tuple-acc-1, auto)
  show r[?doms[?n7n]] \in ?rngs[?n7n]
  by (rule subst[OF hs], rule subst[OF hn], rule h7)

 qed
qed
qed
```

```
lemma zenon-caseother0 :
 P (CASE OTHER -> e0) ==> (P (e0) ==> FALSE) ==> FALSE
proof -
 fix P e0
 assume h: P (CASE OTHER -> e0)
 def cas == CASE OTHER -> e0
 have hh: P (cas) using h by (fold cas-def)
 assume hoth: P (e0) ==> FALSE
 have hb: (∀ i ∈ DOMAIN <<>> : ~<<>>[i]) = TRUE by auto
 have hc: cas = e0 using hb by (unfold cas-def, auto)
 have hg: P (e0) using hh by (rule subst [OF hc])
 show FALSE
 by (rule hoth [OF hg])
qed
```

```
lemma zenon-disjE1 :
  assumes h1: A | B
  assumes h2: A => FALSE
  shows B
using h1 h2 by blast

definition zenon-seqify where
zenon-seqify (x) ≡ IF isASeq (x) THEN x ELSE <<>>

definition zenon-appseq where
zenon-appseq (xs, x) ≡ Append (zenon-seqify (xs), x)

lemma zenon-seqifyIsASeq :
  fixes x
  shows isASeq (zenon-seqify (x))
by (simp only: zenon-seqify-def, rule condI, auto)

lemma zenon-isASeqSeqify :
  fixes x
  assumes h: isASeq (x)
  shows zenon-seqify (x) = x
using h by (simp only: zenon-seqify-def, auto)

lemma zenon-seqify-appseq :
  fixes cs c
  shows zenon-seqify (zenon-appseq (cs, c)) = Append (zenon-seqify (cs), c)
by (simp only: zenon-appseq-def, rule zenon-isASeqSeqify, rule appendIsASeq,
    rule zenon-seqifyIsASeq)

lemma zenon-seqify-empty :
  shows zenon-seqify (<<>>) = <<>>
using zenon-seqify-def by auto

lemma zenon-case-seq-simpl :
  fixes cs c
  shows (∃ i ∈ DOMAIN zenon-seqify (zenon-appseq (cs, c))
          : zenon-seqify (zenon-appseq (cs, c))[i])
         = (c | (∃ i ∈ DOMAIN zenon-seqify (cs)
                 : zenon-seqify (cs)[i]))
proof (rule boolEqual, simp only: zenon-seqify-appseq, rule iffI)
  have h6: isASeq (zenon-seqify (cs)) using zenon-seqifyIsASeq by auto
  assume h1: ∃ i ∈ DOMAIN Append (zenon-seqify (cs), c)
             : Append (zenon-seqify (cs), c)[i]
  show c | (∃ i ∈ DOMAIN zenon-seqify (cs) : zenon-seqify (cs)[i])
  proof
    assume h4: ~ c
    with h1 h6 obtain i
      where i: i ∈ DOMAIN zenon-seqify(cs) Append(zenon-seqify(cs), c)[i]
```

```
      by auto
    with h6 have zenon-seqify(cs)[i] by (auto simp: leq-neq-iff-less[simplified])
    with i show ∃ i ∈ DOMAIN zenon-seqify (cs) : zenon-seqify (cs)[i] by blast
  qed
next
  have h0: isASeq (zenon-seqify (cs)) using zenon-seqifyIsASeq by auto
  assume h1: c | (∃ i ∈ DOMAIN zenon-seqify (cs)
                  : zenon-seqify (cs)[i])
  show ∃ i ∈ DOMAIN Append (zenon-seqify (cs), c)
        : Append (zenon-seqify (cs), c)[i] (is ?g)
  proof (rule disjE [OF h1])
    assume h2: c
    show ?g
    using h2 h0 by auto
  next
    assume h2: ∃ i ∈ DOMAIN zenon-seqify (cs)
              : zenon-seqify (cs)[i]
    show ?g
    proof (rule bExE [OF h2])
      fix i
      assume h3: i \in DOMAIN zenon-seqify (cs)
      have h4: i \in DOMAIN Append (zenon-seqify (cs), c)
        using h0 h3 by auto
      assume h5: zenon-seqify (cs)[i]
      have h6: i ~= Succ[Len (zenon-seqify (cs))]
        using h0 h3 by auto
      have h7: Append (zenon-seqify (cs), c)[i]
        using h6 h5 h3 h0 by force
      show ?g
        using h4 h7 by auto
    qed
  qed
qed (simp-all)

lemma zenon-case-seq-empty :
  assumes h0: ∃ i ∈ DOMAIN zenon-seqify (<<>>)
              : zenon-seqify (<<>>)[i]
  shows FALSE
using zenon-seqify-empty h0 by auto

lemma zenon-case-domain :
  fixes cs es
  assumes h0: ∃ i ∈ DOMAIN cs : cs[i]
  shows ∃ x : x ∈ UNION {CaseArm (cs[i], es[i]) : i \in DOMAIN cs}
        (is ?g)
proof (rule bExE [OF h0])
  fix i
  assume h1: i \in DOMAIN cs
  assume h2: cs[i]
```

```
  show ?g
    using h1 h2 by auto
qed

lemma zenon-case-append1 :
  fixes s x i
  assumes h1: isASeq (s)
  assumes h2: i \in DOMAIN s
  shows Append (s, x)[i] = s[i]
using assms by force

lemma zenon-case-len-domain :
  fixes cs es
  assumes h1: Len (zenon-seqify (cs)) = Len (zenon-seqify (es))
  shows DOMAIN zenon-seqify (cs) = DOMAIN zenon-seqify (es)
using h1 zenon-seqifyIsASeq DomainSeqLen by auto

lemma zenon-case-union-split :
  fixes cs c es e x
  assumes h1: x \in UNION {CaseArm (Append (zenon-seqify (cs), c)[i],
                                    Append (zenon-seqify (es), e)[i])
                            : i \in DOMAIN Append (zenon-seqify (cs), c)}
  assumes h2: Len (zenon-seqify (cs)) = Len (zenon-seqify (es))
  shows x \in CaseArm (c, e)
        | x \in UNION {CaseArm (zenon-seqify (cs)[i], zenon-seqify (es)[i])
                       : i \in DOMAIN zenon-seqify (cs)}
proof
  have h3c: isASeq (zenon-seqify (cs)) using zenon-seqifyIsASeq by auto
  have h3e: isASeq (zenon-seqify (es)) using zenon-seqifyIsASeq by auto
  assume h4: ~ (x \in CaseArm (c, e))
  have h5: ∃ i ∈ DOMAIN Append (zenon-seqify (cs), c)
           : x \in CaseArm (Append (zenon-seqify (cs), c)[i],
                            Append (zenon-seqify (es), e)[i])
    using h1 by auto
  show x \in UNION {CaseArm (zenon-seqify (cs)[i], zenon-seqify (es)[i])
                    : i \in DOMAIN zenon-seqify (cs)}
      (is ?g1)
  proof (rule bExE [OF h5])
    fix i
    assume h6: i \in DOMAIN Append (zenon-seqify(cs), c)
    have h7: i = Succ [Len (zenon-seqify (cs))]
             | i \in DOMAIN (zenon-seqify (cs))
      using h3c h6 by auto
    assume h8: x \in CaseArm (Append (zenon-seqify (cs), c)[i],
                              Append (zenon-seqify (es), e)[i])
    have h9: i \in DOMAIN (zenon-seqify (cs)) (is ?g)
    proof (rule disjE [OF h7])
      assume h10: i = Succ [Len (zenon-seqify (cs))]
      have h11: FALSE using h8 h10 h4 h3c h3e h2 by auto
```

```
      show ?g using h11 by auto
    next
      assume ?g thus ?g by auto
    qed
    have h10: i \in DOMAIN (zenon-seqify (es))
      using h9 h2 h3c h3e DomainSeqLen by auto
    have h11: x \in CaseArm (zenon-seqify (cs)[i], zenon-seqify (es)[i])
    using h8 h9 h3c h3e h10 zenon-case-append1 by auto
    show ?g1 using h11 h9 by auto
  qed
qed

lemma zenon-case-union-simpl :
  fixes cs c es e x
  shows (Len (zenon-seqify (zenon-appseq (cs, c)))
          = Len (zenon-seqify (zenon-appseq (es, e)))
          & x \in UNION {CaseArm (zenon-seqify (zenon-appseq (cs, c))[i],
                                  zenon-seqify (zenon-appseq (es, e))[i])
                          : i \in DOMAIN zenon-seqify (zenon-appseq (cs, c))})
        = (  Len (zenon-seqify (zenon-appseq (cs, c)))
             = Len (zenon-seqify (zenon-appseq (es, e)))
             & x \in CaseArm (c, e)
           | Len (zenon-seqify (cs)) = Len (zenon-seqify (es))
             & x \in UNION {CaseArm (zenon-seqify (cs)[i],
                                     zenon-seqify (es)[i])
                             : i \in DOMAIN zenon-seqify (cs)})
proof (rule boolEqual, simp only: zenon-seqify-appseq, rule iffI)
  assume h1: Len (Append (zenon-seqify (cs), c))
             = Len (Append (zenon-seqify (es), e))
             & x \in UNION {CaseArm (Append (zenon-seqify (cs), c)[i],
                                     Append (zenon-seqify (es), e)[i])
                             : i \in DOMAIN Append (zenon-seqify (cs), c)}
  have h1a: Len (Append (zenon-seqify (cs), c))
            = Len (Append (zenon-seqify (es), e))
    using h1 by blast
  have h1b: x \in UNION {CaseArm (Append (zenon-seqify (cs), c)[i],
                                  Append (zenon-seqify (es), e)[i])
                          : i \in DOMAIN Append (zenon-seqify (cs), c)}
    using h1 by blast
  show   Len (Append (zenon-seqify (cs), c))
          = Len (Append (zenon-seqify (es), e))
          & x \in CaseArm (c, e)
        | Len (zenon-seqify (cs)) = Len (zenon-seqify (es))
          & x \in UNION {CaseArm (zenon-seqify (cs)[i], zenon-seqify (es)[i])
                          : i \in DOMAIN zenon-seqify (cs)}
  proof
    assume h2: ~(Len (Append (zenon-seqify (cs), c))
                 = Len (Append (zenon-seqify (es), e))
                 & x \in CaseArm (c, e))
```

```
    have h3: Len (zenon-seqify (cs)) = Len (zenon-seqify (es)) (is ?g3)
      using h1a zenon-seqifyIsASeq by auto
    have h4: ~(x \in CaseArm (c, e))
      using h2 h1a by blast
    have h5: x \in UNION {CaseArm (zenon-seqify (cs)[i], zenon-seqify (es)[i])
                          : i \in DOMAIN zenon-seqify (cs)} (is ?g5)
      using zenon-case-union-split [OF h1b h3] h4 by auto
    show ?g3 & ?g5 using h3 h5 by blast
  qed
next
  assume h1:   Len (Append (zenon-seqify (cs), c))
               = Len (Append (zenon-seqify (es), e))
               & x \in CaseArm (c, e)
             | Len (zenon-seqify (cs)) = Len (zenon-seqify (es))
               & x \in UNION {CaseArm (zenon-seqify (cs)[i],
                                       zenon-seqify (es)[i])
                             : i \in DOMAIN zenon-seqify (cs)}
  show Len (Append (zenon-seqify (cs), c))
       = Len (Append (zenon-seqify (es), e))
       & x \in UNION {CaseArm (Append (zenon-seqify (cs), c)[i],
                               Append (zenon-seqify (es), e)[i])
                     : i \in DOMAIN Append (zenon-seqify (cs), c)}
       (is ?g)
  proof (rule disjE [OF h1])
    assume h2: Len (Append (zenon-seqify (cs), c))
               = Len (Append (zenon-seqify (es), e))
               & x \in CaseArm (c, e)
    have h4: Len (Append (zenon-seqify (cs), c))
             = Len (Append (zenon-seqify (es), e))
      using h2 by blast
    have h3: Len (zenon-seqify (cs)) = Len (zenon-seqify (es))
      using h4 zenon-seqifyIsASeq by auto
    have h5: x \in CaseArm (c, e)
      using h2 by blast
    have h6: x \in UNION {CaseArm (Append (zenon-seqify (cs), c)[i],
                                   Append (zenon-seqify (es), e)[i])
                          : i \in DOMAIN Append (zenon-seqify (cs), c)}
      using h5 zenon-seqifyIsASeq appendElt2 h3 by auto
    show ?g
      using h4 h6 by auto
  next
    assume h2: Len (zenon-seqify (cs)) = Len (zenon-seqify (es))
               & x \in UNION {CaseArm (zenon-seqify (cs)[i],
                                       zenon-seqify (es)[i])
                             : i \in DOMAIN zenon-seqify (cs)}
    have h3: Len (zenon-seqify (cs)) = Len (zenon-seqify (es))
      using h2 by blast
    have h4: Len (Append (zenon-seqify (cs), c))
             = Len (Append (zenon-seqify (es), e))
```

```
      using h3 zenon-seqifyIsASeq by auto
    have h5: x \in UNION {CaseArm (zenon-seqify (cs)[i],
                                    zenon-seqify (es)[i])
                          : i \in DOMAIN zenon-seqify (cs)}
      using h2 by blast
    have h6: ∃ i ∈ DOMAIN zenon-seqify (cs)
             : x \in CaseArm (zenon-seqify (cs)[i], zenon-seqify (es)[i])
    using h5 by auto
    have h7: x \in UNION {CaseArm (Append (zenon-seqify (cs), c)[i],
                                    Append (zenon-seqify (es), e)[i])
                          : i \in DOMAIN Append (zenon-seqify (cs), c)}
           (is ?g7)
    proof (rule bExE [OF h6])
      fix i
      assume h8: i \in DOMAIN zenon-seqify (cs)
      have h9: i \in DOMAIN zenon-seqify (es)
        using h8 zenon-case-len-domain [OF h3] by auto
      have h10: i \in DOMAIN Append (zenon-seqify (cs), c)
      using h8 zenon-seqifyIsASeq by auto
      assume h11: x \in CaseArm (zenon-seqify (cs)[i], zenon-seqify (es)[i])
      have h12: x \in CaseArm (Append (zenon-seqify (cs), c)[i],
                                Append (zenon-seqify (es), e)[i])
             (is ?g12)
        using h11 zenon-case-append1 [OF - h8] zenon-case-append1 [OF - h9]
              zenon-seqifyIsASeq
        by auto
      show ?g7
      proof
        show ?g12 by (rule h12)
      next
        show CaseArm (Append (zenon-seqify (cs), c) [i],
                      Append (zenon-seqify (es), e) [i])
             \in {CaseArm (Append (zenon-seqify (cs), c)[i],
                           Append (zenon-seqify (es), e)[i])
                  : i \in DOMAIN Append (zenon-seqify (cs), c)}
          using h10 by auto
      qed
    qed
    show ?g using h4 h7 by blast
  qed
qed (simp-all)

lemma zenon-case-len-simpl :
  fixes cs c es e
  shows (Len (zenon-seqify (zenon-appseq (cs, c)))
         = Len (zenon-seqify (zenon-appseq (es, e))))
        = (Len (zenon-seqify (cs)) = Len (zenon-seqify (es)))
proof (rule boolEqual, simp only: zenon-seqify-appseq, rule iffI)
  assume h1: Len (Append (zenon-seqify (cs), c))
```

```
            = Len (Append (zenon-seqify (es), e))
  show Len (zenon-seqify (cs)) = Len (zenon-seqify (es))
    using h1 zenon-seqifyIsASeq by auto
next
  assume h1: Len (zenon-seqify (cs)) = Len (zenon-seqify (es))
  show Len (Append (zenon-seqify (cs), c))
        = Len (Append (zenon-seqify (es), e))
    using h1 zenon-seqifyIsASeq by auto
qed (simp-all)

lemma zenon-case-empty-union :
  fixes x
  assumes h: x \in UNION {CaseArm (<<>>[i], <<>>[i]) : i \in DOMAIN
<<>>}
  shows FALSE
using h by auto

lemma zenon-case-oth-simpl-l1 :
  fixes cs c
  assumes g0 : ~c
  shows (∀ i ∈ DOMAIN Append(zenon-seqify(cs), c)
          : ~ Append(zenon-seqify(cs), c)[i])
         = (∀ i ∈ DOMAIN zenon-seqify(cs)
            : ~ zenon-seqify(cs)[i])
        (is ?f1 = ?f2)
proof (rule boolEqual, rule iffI)
  assume h6: ?f1
  show ?f2
  proof
    fix i
    assume h7: i \in DOMAIN zenon-seqify (cs)
    have h8: i \in DOMAIN Append (zenon-seqify (cs), c)
      using h7 zenon-seqifyIsASeq by auto
    have h9: zenon-seqify (cs) [i] = Append (zenon-seqify (cs), c)[i]
      using zenon-case-append1 zenon-seqifyIsASeq h7 by auto
    show ~ zenon-seqify(cs)[i]
      using h9 h8 h6 by auto
  qed
next
  assume h6: ?f2
  show ?f1
  proof
    fix i
    assume h7: i \in DOMAIN Append (zenon-seqify (cs), c)
    show ~Append (zenon-seqify (cs), c)[i]
      using g0 h7 h6 zenon-seqifyIsASeq by (unfold Append-def, auto)
  qed
qed (simp-all)
```

```
lemma zenon-case-oth-simpl-l2 :
  fixes cs c es e
  assumes g0: ~c
  assumes g2: Len(zenon-seqify(cs)) = Len(zenon-seqify(es))
  shows UNION {CaseArm
          (Append(zenon-seqify(cs), c)[i],
           Append
            (zenon-seqify(es),
             e)[i]) : i \in DOMAIN Append(zenon-seqify(cs), c)}
          = UNION {CaseArm
              (zenon-seqify(cs)[i],
               zenon-seqify(es)[i]) : i \in DOMAIN zenon-seqify(cs)}
         (is ?u1 = ?u2)
proof
  fix x
  assume h6: x \in ?u1
  show x \in ?u2
  proof (rule UNIONE [OF h6])
    fix B
    assume h7: x \in B
    assume h8: B \in {CaseArm
          (Append(zenon-seqify(cs), c)[i],
           Append
            (zenon-seqify(es),
             e)[i]) : i \in DOMAIN Append(zenon-seqify(cs), c)}
    show x \in ?u2
    proof (rule setOfAllE [OF h8])
      fix i
      assume h9: i \in DOMAIN Append (zenon-seqify (cs), c)
      assume h10: CaseArm (Append (zenon-seqify (cs), c)[i],
                           Append (zenon-seqify (es), e)[i]) = B
      have h11: i = Succ[Len (zenon-seqify (cs))] ==> FALSE
      proof -
        assume h12: i = Succ[Len (zenon-seqify (cs))]
        have h13: B = CaseArm (c, e)
          using h10 h12 g0 appendElt2 zenon-seqifyIsASeq by auto
        show FALSE
          using h7 h13 g0 by auto
      qed
      have h12: Append (zenon-seqify (cs), c)[i] = zenon-seqify (cs)[i]
        using h11 zenon-seqifyIsASeq[of cs] h9 by force
      have h13: Append (zenon-seqify (es), e)[i] = zenon-seqify (es)[i]
        using h11 zenon-seqifyIsASeq[of es] h9 g2 by force
      show x \in ?u2
      proof
        show x \in B by (rule h7)
      next
        have h14: i \in DOMAIN zenon-seqify (cs)
          using h9 zenon-seqifyIsASeq h11 by auto
```

```
        show B \in {CaseArm (zenon-seqify(cs)[i],
                            zenon-seqify(es)[i]) :
                   i \in DOMAIN zenon-seqify(cs)}
          using h10 h12 h13 h14 by auto
      qed
    qed
  qed
next
  fix x
  assume h6: x \in ?u2
  show x \in ?u1
  proof (rule UNIONE [OF h6])
    fix B
    assume h7: x \in B
    assume h8: B \in {CaseArm (zenon-seqify (cs)[i],
                              zenon-seqify (es)[i])
                      : i \in DOMAIN zenon-seqify (cs)}
    show x \in ?u1
    proof (rule setOfAllE [OF h8])
      fix i
      assume h9: i \in DOMAIN zenon-seqify (cs)
      assume h10: CaseArm (zenon-seqify (cs)[i], zenon-seqify (es)[i]) = B
      show x \in ?u1
      proof
        show x \in B by (rule h7)
      next
        show B \in {CaseArm (Append(zenon-seqify(cs), c)[i],
                            Append(zenon-seqify(es), e)[i])
                   : i \in DOMAIN Append(zenon-seqify(cs), c)}
        proof
          show i \in DOMAIN Append (zenon-seqify (cs), c)
            using h9 zenon-seqifyIsASeq by auto
        next
          have h11: Append (zenon-seqify (cs), c)[i] = zenon-seqify (cs)[i]
            using zenon-seqifyIsASeq zenon-case-append1 [OF - h9] by auto
          have h12: i \in DOMAIN zenon-seqify (es)
            using h9 zenon-case-len-domain [OF g2] by auto
          have h13: Append (zenon-seqify (es), e)[i] = zenon-seqify (es)[i]
            using zenon-seqifyIsASeq zenon-case-append1 [OF - h12] g2
                  zenon-case-len-domain
            by auto
          show B = CaseArm (Append (zenon-seqify (cs), c)[i],
                           Append (zenon-seqify (es), e)[i])
            using h11 h13 h10 by auto
        qed
      qed
    qed
  qed
qed
```

```
lemma zenon-case-oth-simpl :
  fixes cs c es e x dcs oth
  shows (~ (c | dcs)
          & Len (zenon-seqify (zenon-appseq (cs, c)))
            = Len (zenon-seqify (zenon-appseq (es, e)))
          & x = UNION {CaseArm (zenon-seqify (zenon-appseq (cs, c))[i],
                               zenon-seqify (zenon-appseq (es, e))[i])
                     : i \in DOMAIN zenon-seqify (zenon-appseq (cs, c))}
               \cup CaseArm (∀ i ∈ DOMAIN zenon-seqify
                                                 (zenon-appseq (cs, c))
                             : ~zenon-seqify (zenon-appseq (cs, c))[i],
                             oth))
         = (~c
            & (~dcs
               & Len (zenon-seqify (cs)) = Len (zenon-seqify (es))
               & x = UNION {CaseArm (zenon-seqify (cs)[i], zenon-seqify (es)[i])
                           : i \in DOMAIN zenon-seqify (cs)}
                    \cup CaseArm (∀ i ∈ DOMAIN zenon-seqify (cs)
                                  : ~zenon-seqify(cs)[i],
                                  oth)))
proof (rule boolEqual, simp only: zenon-seqify-appseq, rule iffI)
  assume h0: ~ (c | dcs)
             & Len(Append(zenon-seqify(cs), c))
               = Len(Append(zenon-seqify(es), e))
             & x = UNION {CaseArm (Append (zenon-seqify (cs), c)[i],
                                  Append (zenon-seqify (es), e)[i])
                         : i \in DOMAIN Append(zenon-seqify(cs), c)}
                  \cup CaseArm (∀ i ∈ DOMAIN Append (zenon-seqify
                                                        (cs),
                                                     c)
                                : ~ Append(zenon-seqify(cs), c)[i],
                                oth)
           (is ?h1 & ?h2 & ?h3)
  have h1: ?h1 using h0 by blast
  have h2: ?h2 using h0 by blast
  have h3: ?h3 using h0 by blast
  have g0: ~c (is ?g0) using h1 by blast
  have g1: ~dcs (is ?g1) using h1 by blast
  have g2: Len(zenon-seqify(cs)) = Len(zenon-seqify(es)) (is ?g2)
    using h2 zenon-seqifyIsASeq by auto
  have g3: x = UNION {CaseArm (zenon-seqify(cs)[i],
                               zenon-seqify(es)[i])
                     : i \in DOMAIN zenon-seqify(cs)}
               \cup CaseArm (∀ i ∈ DOMAIN zenon-seqify(cs)
                             : ~ zenon-seqify(cs)[i],
                             oth)
         (is ?g3)
    using h3 zenon-case-oth-simpl-l1 [OF g0] zenon-case-oth-simpl-l2 [OF g0 g2]
```

```
    by auto
  show ?g0 & ?g1 & ?g2 & ?g3
    using g0 g1 g2 g3 by blast
next
  assume h: ~c
           & ~dcs
           & Len (zenon-seqify (cs)) = Len (zenon-seqify (es))
           & x = UNION {CaseArm (zenon-seqify (cs)[i], zenon-seqify (es)[i])
                      : i \in DOMAIN zenon-seqify (cs)}
                 \cup CaseArm (∀ i ∈ DOMAIN zenon-seqify (cs)
                              : ~zenon-seqify (cs)[i],
                              oth)
          (is ?h0 & ?h1 & ?h2 & ?h3)
  have h0: ?h0 using h by blast
  have h1: ?h1 using h by blast
  have h2: ?h2 using h by blast
  have h3: ?h3 using h by blast
  have g1: ~(c | dcs) (is ?g1) using h0 h1 by blast
  have g2: Len (Append (zenon-seqify (cs), c))
          = Len (Append (zenon-seqify (es), e))
         (is ?g2)
    using h2 zenon-seqifyIsASeq by auto
  have g3: x = UNION {CaseArm (Append (zenon-seqify (cs), c)[i],
                              Append (zenon-seqify (es), e)[i])
                     : i \in DOMAIN Append(zenon-seqify(cs), c)}
          \cup CaseArm (∀ i ∈ DOMAIN Append(zenon-seqify(cs), c)
                       : ~ Append(zenon-seqify(cs), c)[i],
                       oth)
         (is ?g3)
   using h3 zenon-case-oth-simpl-l1 [OF h0] zenon-case-oth-simpl-l2 [OF h0 h2]
   by auto
  show ?g1 & ?g2 & ?g3
    using g1 g2 g3 by blast
qed (simp-all)

lemma zenon-case-oth-empty :
  fixes x
  shows (x = UNION {CaseArm (zenon-seqify (<<>>)[i], zenon-seqify (<<>>)[i])
                   : i \in DOMAIN zenon-seqify (<<>>)}
             \cup CaseArm (∀ i ∈ DOMAIN zenon-seqify (<<>>)
                          : ~zenon-seqify(<<>>)[i],
                          oth))
         = (x = {oth})
by (rule boolEqual, simp only: zenon-seqify-empty, rule iffI, auto)
```

```
lemma zenon-case1 :
  fixes P c1x e1x
  assumes h: P (CASE c1x -> e1x)
           (is P (?cas))
  assumes h1: c1x ==> P (e1x) ==> FALSE

  assumes hoth: ~c1x & TRUE ==> FALSE
  shows FALSE
proof -
  def cs == <<c1x>> (is ?cs)
  def es == <<e1x>> (is ?es)
  def arms == UNION {CaseArm (?cs[i], ?es[i]) : i \in DOMAIN ?cs}
            (is ?arms)
  def cas == ?cas
  have h0: P (cas) using h by (fold cas-def)
  def dcs == c1x (is ?dcs)
  show FALSE
  proof (rule zenon-em [of ?dcs])
    assume ha: ~(?dcs)
    have hh1: ~c1x using ha by blast

    show FALSE
      using hoth hh1 by blast
  next
    assume ha: ?dcs
    def scs == zenon-seqify (zenon-appseq (
                             <<>>, c1x))
             (is ?scs)
    def ses == zenon-seqify (zenon-appseq (
                             <<>>, e1x))
             (is ?ses)
    have ha1: ∃ i ∈ DOMAIN ?scs : ?scs[i]
      using ha zenon-case-seq-empty
      by (simp only: zenon-case-seq-simpl zenon-seqify-empty, blast)
    have ha2: ∃ i ∈ DOMAIN ?cs : ?cs[i]
      using ha1 by (simp only: zenon-seqify-appseq zenon-seqify-empty)
    have hb: \E x : x \in arms
      using zenon-case-domain [OF ha2, where es = ?es]
      by (unfold arms-def, blast)
    have hc: (CHOOSE x : x \in arms) \in arms
      using hb by (unfold Ex-def, auto)
    have hf0: ?cas \in arms
      using hc by (unfold arms-def, fold Case-def)
    have hf3: cas \in UNION {CaseArm (?scs[i], ?ses[i])
                             : i \in DOMAIN ?scs}
           (is ?hf3)
      using hf0 by (fold cas-def, unfold arms-def,
                   simp only: zenon-seqify-appseq zenon-seqify-empty)
    have hf4: Len (?scs) = Len (?ses) (is ?hf4)
```

```
    by (simp only: zenon-case-len-simpl)
  have hf5: ?hf4 & ?hf3
    by (rule conjI [OF hf4 hf3])
  have hf:
           cas \in CaseArm (c1x, e1x)
         | cas \in UNION {CaseArm (zenon-seqify (<<>>)[i],
                                  zenon-seqify (<<>>)[i])
                         : i \in DOMAIN zenon-seqify (<<>>)}
          (is - | ?gxx)
    using hf5 by (simp only: zenon-case-union-simpl, blast)
  have hg1x: cas \in CaseArm (c1x, e1x) => FALSE
    using h0 h1 by auto

  from hf
  have hh0: ?gxx
    by (rule zenon-disjE1 [OF - hg1x])
  have hi: cas \in UNION {CaseArm (<<>>[i], <<>>[i])
                         : i \in DOMAIN <<>>}
    using hh0 by (simp only: zenon-seqify-empty)
  show FALSE
    by (rule zenon-case-empty-union [OF hi])
 qed
qed

lemma zenon-caseother1 :
 fixes P oth c1x e1x
 assumes h: P (CASE c1x -> e1x
                [] OTHER -> oth)
          (is P (?cas))
 assumes h1: c1x ==> P (e1x) ==> FALSE

 assumes hoth: ~c1x & TRUE ==> P (oth) ==> FALSE
 shows FALSE
proof -
 def cs == <<c1x>> (is ?cs)
 def es == <<e1x>> (is ?es)
 def arms == UNION {CaseArm (?cs[i], ?es[i]) : i \in DOMAIN ?cs}
          (is ?arms)
 def cas == ?cas
 have h0: P (cas) using h by (fold cas-def)
 def dcs == c1x | FALSE (is ?dcs)
 def scs == zenon-seqify (zenon-appseq (
                      <<>>, c1x))
          (is ?scs)
 have hscs : ?cs = ?scs
   by (simp only: zenon-seqify-appseq zenon-seqify-empty)
 def ses == zenon-seqify (zenon-appseq (
                      <<>>, e1x))
          (is ?ses)
```

have *hses* : *?es* = *?ses*
  by (*simp only*: *zenon-seqify-appseq zenon-seqify-empty*)
have *hlen*: *Len* (*?scs*) = *Len* (*?ses*) (**is** *?hlen*)
  by (*simp only*: *zenon-case-len-simpl*)
def *armoth* == *CaseArm* (∀ *i* ∈ *DOMAIN ?cs* : ~ *?cs*[*i*], *oth*)
      (**is** *?armoth*)
show *FALSE*
proof (*rule zenon-em* [*of ?dcs*])
  assume *ha*: ~(*?dcs*)
  have *hb*: *P* (*CHOOSE x* : *x* \in *arms* ∪ *armoth*)
    using *h* **by** (*unfold CaseOther-def*, *fold arms-def armoth-def*)
  have *hc*: *arms* \cup *armoth*
      = *UNION* {*CaseArm* (*?scs*[*i*], *?ses*[*i*]) : *i* \in *DOMAIN ?scs*}
       \cup *CaseArm* (∀ *i* ∈ *DOMAIN ?scs* : ~ *?scs*[*i*],
            oth)
     (**is** - = *?sarmsoth*)
    using *hscs hses* **by** (*unfold arms-def armoth-def*, *auto*)
  have *hd*: ~(*?dcs*) & *?hlen* & *arms* ∪ *armoth* = *?sarmsoth*
    using *ha hlen hc* **by** *blast*
  have *he*: *arms* ∪ *armoth* = {*oth*}
    using *hd* **by** (*simp only*: *zenon-case-oth-simpl zenon-case-oth-empty*)
  have *hf*: (*CHOOSE x* : *x* \in *arms* \cup *armoth*) = *oth*
    using *he* **by** *auto*
  have *hg*: *P* (*oth*)
    using *hb hf* **by** *auto*
  have *hh1*: ~ *c1x* **using** *ha* **by** *blast*

  show *FALSE*
    using *hg hoth hh1* **by** *blast*
next
  assume *ha*: *?dcs*
  have *ha1*: ∃ *i* ∈ *DOMAIN ?scs* : *?scs*[*i*]
    using *ha zenon-case-seq-empty*
    by (*simp only*: *zenon-case-seq-simpl zenon-seqify-empty*, *blast*)
  have *ha2*: ∃ *i* ∈ *DOMAIN ?cs* : *?cs*[*i*]
    using *ha1 hscs* **by** *auto*
  have *ha3*: ~ (∀ *i* ∈ *DOMAIN ?cs* : ~ *?cs*[*i*])
    using *ha2* **by** *blast*
  have *ha4*: *armoth* = {}
    using *ha3 condElse* [*OF ha3*, **where** *t* = {*oth*} **and** *e* = {}]
    by (*unfold armoth-def CaseArm-def*, *blast*)
  have *hb*: \E *x* : *x* \in *arms* \cup *armoth*
    using *zenon-case-domain* [*OF ha2*, **where** *es* = *?es*]
    by (*unfold arms-def*, *blast*)
  have *hc*: (*CHOOSE x* : *x* \in *arms* \cup *armoth*)
      \in *arms* \cup *armoth*
    using *hb* **by** (*unfold Ex-def*, *auto*)
  have *hf0*: *?cas* \in *arms* \cup *armoth*
    using *hc* **by** (*unfold arms-def armoth-def*, *fold CaseOther-def*)

```
    have hf1: cas \in arms \cup armoth
      using hf0 by (fold cas-def)
    have hf2: cas \in arms
      using hf1 ha4 by auto
    have hf3: cas \in UNION {CaseArm (?scs[i], ?ses[i])
                                 : i \in DOMAIN ?scs}
          (is ?hf3)
      using hf2 by (unfold arms-def,
                   simp only: zenon-seqify-appseq zenon-seqify-empty)
    have hf5: ?hlen & ?hf3
      by (rule conjI [OF hlen hf3])
    have hf:
            cas \in CaseArm (c1x, e1x)
          | cas \in UNION {CaseArm (zenon-seqify (<<>>)[i],
                                    zenon-seqify (<<>>)[i])
                          : i \in DOMAIN zenon-seqify (<<>>)}
          (is - | ?gxx)
      using hf5 by (simp only: zenon-case-union-simpl, blast)
    have hg1x: cas \in CaseArm (c1x, e1x) => FALSE
      using h0 h1 by auto

    from hf
    have hh0: ?gxx
      by (rule zenon-disjE1 [OF - hg1x])
    have hi: cas \in UNION {CaseArm (<<>>[i], <<>>[i])
                          : i \in DOMAIN <<>>}
      using hh0 by (simp only: zenon-seqify-empty)
    show FALSE
      by (rule zenon-case-empty-union [OF hi])
  qed
qed

lemma zenon-case2 :
  fixes P c1x e1x c2x e2x
  assumes h: P (CASE c1x -> e1x [] c2x -> e2x)
          (is P (?cas))
  assumes h1: c1x ==> P (e1x) ==> FALSE
  assumes h2: c2x ==> P (e2x) ==> FALSE
  assumes hoth: ~c2x & ~c1x & TRUE ==> FALSE
  shows FALSE
proof -
  def cs == <<c1x, c2x>> (is ?cs)
  def es == <<e1x, e2x>> (is ?es)
  def arms == UNION {CaseArm (?cs[i], ?es[i]) : i \in DOMAIN ?cs}
          (is ?arms)
  def cas == ?cas
  have h0: P (cas) using h by (fold cas-def)
  def dcs == c2x | c1x (is ?dcs)
  show FALSE
```

```
proof (rule zenon-em [of ?dcs])
  assume ha: ~(?dcs)
  have hh1: ~c1x using ha by blast
  have hh2: ~c2x using ha by blast
  show FALSE
    using hoth hh1 hh2 by blast
next
  assume ha: ?dcs
  def scs == zenon-seqify (zenon-appseq (zenon-appseq (
                          <<>>, c1x), c2x))
            (is ?scs)
  def ses == zenon-seqify (zenon-appseq (zenon-appseq (
                          <<>>, e1x), e2x))
            (is ?ses)
  have ha1: ∃ i ∈ DOMAIN ?scs : ?scs[i]
    using ha zenon-case-seq-empty
    by (simp only: zenon-case-seq-simpl zenon-seqify-empty, blast)
  have ha2: ∃ i ∈ DOMAIN ?cs : ?cs[i]
    using ha1 by (simp only: zenon-seqify-appseq zenon-seqify-empty)
  have hb: \E x : x \in arms
    using zenon-case-domain [OF ha2, where es = ?es]
    by (unfold arms-def, blast)
  have hc: (CHOOSE x : x \in arms) \in arms
    using hb by (unfold Ex-def, auto)
  have hf0: ?cas \in arms
    using hc by (unfold arms-def, fold Case-def)
  have hf3: cas \in UNION {CaseArm (?scs[i], ?ses[i])
                             : i \in DOMAIN ?scs}
          (is ?hf3)
    using hf0 by (fold cas-def, unfold arms-def,
                 simp only: zenon-seqify-appseq zenon-seqify-empty)
  have hf4: Len (?scs) = Len (?ses) (is ?hf4)
    by (simp only: zenon-case-len-simpl)
  have hf5: ?hf4 & ?hf3
    by (rule conjI [OF hf4 hf3])
  have hf:
            cas \in CaseArm (c2x, e2x)
          | cas \in CaseArm (c1x, e1x)
          | cas \in UNION {CaseArm (zenon-seqify (<<>>)[i],
                                   zenon-seqify (<<>>)[i])
                          : i \in DOMAIN zenon-seqify (<<>>)}
          (is - | ?gxx)
    using hf5 by (simp only: zenon-case-union-simpl, blast)
  have hg1x: cas \in CaseArm (c1x, e1x) => FALSE
    using h0 h1 by auto
  have hg2x: cas \in CaseArm (c2x, e2x) => FALSE
    using h0 h2 by auto
  from hf
  have hh0: ?gxx (is - | ?g0)
```

```
    by (rule zenon-disjE1 [OF - hg2x])
   then have hh0: ?g0
    by (rule zenon-disjE1 [OF - hg1x])
   have hi: cas \in UNION {CaseArm (<<>>[i], <<>>[i])
                              : i \in DOMAIN <<>>}
    using hh0 by (simp only: zenon-seqify-empty)
   show FALSE
    by (rule zenon-case-empty-union [OF hi])
  qed
qed

lemma zenon-caseother2 :
  fixes P oth c1x e1x c2x e2x
  assumes h: P (CASE c1x -> e1x [] c2x -> e2x
                  [] OTHER -> oth)
            (is P (?cas))
  assumes h1: c1x ==> P (e1x) ==> FALSE
  assumes h2: c2x ==> P (e2x) ==> FALSE
  assumes hoth: ~c2x & ~c1x & TRUE ==> P (oth) ==> FALSE
  shows FALSE
proof -
  def cs == <<c1x, c2x>> (is ?cs)
  def es == <<e1x, e2x>> (is ?es)
  def arms == UNION {CaseArm (?cs[i], ?es[i]) : i \in DOMAIN ?cs}
             (is ?arms)
  def cas == ?cas
  have h0: P (cas) using h by (fold cas-def)
  def dcs == c2x | c1x | FALSE (is ?dcs)
  def scs == zenon-seqify (zenon-appseq (zenon-appseq (
                           <<>>, c1x), c2x))
            (is ?scs)
  have hscs : ?cs = ?scs
   by (simp only: zenon-seqify-appseq zenon-seqify-empty)
  def ses == zenon-seqify (zenon-appseq (zenon-appseq (
                           <<>>, e1x), e2x))
            (is ?ses)
  have hses : ?es = ?ses
   by (simp only: zenon-seqify-appseq zenon-seqify-empty)
  have hlen: Len (?scs) = Len (?ses) (is ?hlen)
   by (simp only: zenon-case-len-simpl)
  def armoth == CaseArm (∀ i ∈ DOMAIN ?cs : ~?cs[i], oth)
              (is ?armoth)
  show FALSE
  proof (rule zenon-em [of ?dcs])
   assume ha: ~(?dcs)
   have hb: P (CHOOSE x : x \in arms ∪ armoth)
    using h by (unfold CaseOther-def, fold arms-def armoth-def)
   have hc: arms \cup armoth
           = UNION {CaseArm (?scs[i], ?ses[i]) : i \in DOMAIN ?scs}
```

\cup *CaseArm* (∀ *i* ∈ *DOMAIN ?scs* : ~ *?scs*[*i*],
oth)
(**is** - = *?sarmsoth*)
using *hscs hses* **by** (*unfold arms-def armoth-def*, *auto*)
have *hd*: ~(*?dcs*) & *?hlen* & *arms* ∪ *armoth* = *?sarmsoth*
using *ha hlen hc* **by** *blast*
have *he*: *arms* ∪ *armoth* = {*oth*}
using *hd* **by** (*simp only*: *zenon-case-oth-simpl zenon-case-oth-empty*)
have *hf*: (*CHOOSE x* : *x* \in *arms* \cup *armoth*) = *oth*
using *he* **by** *auto*
have *hg*: *P* (*oth*)
using *hb hf* **by** *auto*
have *hh1*: ~ *c1x* **using** *ha* **by** *blast*
have *hh2*: ~ *c2x* **using** *ha* **by** *blast*
show *FALSE*
using *hg hoth hh1 hh2* **by** *blast*
next
assume *ha*: *?dcs*
have *ha1*: ∃ *i* ∈ *DOMAIN ?scs* : *?scs*[*i*]
using *ha zenon-case-seq-empty*
by (*simp only*: *zenon-case-seq-simpl zenon-seqify-empty*, *blast*)
have *ha2*: ∃ *i* ∈ *DOMAIN ?cs* : *?cs*[*i*]
using *ha1 hscs* **by** *auto*
have *ha3*: ~ (∀ *i* ∈ *DOMAIN ?cs* : ~ *?cs*[*i*])
using *ha2* **by** *blast*
have *ha4*: *armoth* = {}
using *ha3 condElse* [*OF ha3*, **where** *t* = {*oth*} **and** *e* = {}]
by (*unfold armoth-def CaseArm-def*, *blast*)
have *hb*: \E *x* : *x* \in *arms* \cup *armoth*
using *zenon-case-domain* [*OF ha2*, **where** *es* = *?es*]
by (*unfold arms-def*, *blast*)
have *hc*: (*CHOOSE x* : *x* \in *arms* \cup *armoth*)
\in *arms* \cup *armoth*
using *hb* **by** (*unfold Ex-def*, *auto*)
have *hf0*: *?cas* \in *arms* \cup *armoth*
using *hc* **by** (*unfold arms-def armoth-def*, *fold CaseOther-def*)
have *hf1*: *cas* \in *arms* \cup *armoth*
using *hf0* **by** (*fold cas-def*)
have *hf2*: *cas* \in *arms*
using *hf1 ha4* **by** *auto*
have *hf3*: *cas* \in *UNION* {*CaseArm* (*?scs*[*i*], *?ses*[*i*])
: *i* \in *DOMAIN ?scs*}
(**is** *?hf3*)
using *hf2* **by** (*unfold arms-def*,
simp only: *zenon-seqify-appseq zenon-seqify-empty*)
have *hf5*: *?hlen* & *?hf3*
by (*rule conjI* [*OF hlen hf3*])
have *hf*:
cas \in *CaseArm* (*c2x*, *e2x*)

```
            | cas \in CaseArm (c1x, e1x)
            | cas \in UNION {CaseArm (zenon-seqify (<<>>)[i],
                                     zenon-seqify (<<>>)[i])
                             : i \in DOMAIN zenon-seqify (<<>>)}
            (is - | ?gxx)
      using hf5 by (simp only: zenon-case-union-simpl, blast)
    have hg1x: cas \in CaseArm (c1x, e1x) => FALSE
      using h0 h1 by auto
    have hg2x: cas \in CaseArm (c2x, e2x) => FALSE
      using h0 h2 by auto
    from hf
    have hh0: ?gxx (is - | ?g0)
      by (rule zenon-disjE1 [OF - hg2x])
    then have hh0: ?g0
      by (rule zenon-disjE1 [OF - hg1x])
    have hi: cas \in UNION {CaseArm (<<>>[i], <<>>[i])
                             : i \in DOMAIN <<>>}
      using hh0 by (simp only: zenon-seqify-empty)
    show FALSE
      by (rule zenon-case-empty-union [OF hi])
  qed
qed

lemma zenon-case3 :
  fixes P c1x e1x c2x e2x c3x e3x
  assumes h: P (CASE c1x -> e1x [] c2x -> e2x [] c3x -> e3x)
           (is P (?cas))
  assumes h1: c1x ==> P (e1x) ==> FALSE
  assumes h2: c2x ==> P (e2x) ==> FALSE
  assumes h3: c3x ==> P (e3x) ==> FALSE
  assumes hoth: ~c3x & ~c2x & ~c1x & TRUE ==> FALSE
  shows FALSE
proof -
  def cs == <<c1x, c2x, c3x>> (is ?cs)
  def es == <<e1x, e2x, e3x>> (is ?es)
  def arms == UNION {CaseArm (?cs[i], ?es[i]) : i \in DOMAIN ?cs}
            (is ?arms)
  def cas == ?cas
  have h0: P (cas) using h by (fold cas-def)
  def dcs == c3x | c2x | c1x (is ?dcs)
  show FALSE
  proof (rule zenon-em [of ?dcs])
    assume ha: ~(?dcs)
    have hh1: ~c1x using ha by blast
    have hh2: ~c2x using ha by blast
    have hh3: ~c3x using ha by blast
    show FALSE
      using hoth hh1 hh2 hh3 by blast
  next
```

```
assume ha: ?dcs
def scs == zenon-seqify (zenon-appseq (zenon-appseq (zenon-appseq (
                      <<>>, c1x), c2x), c3x))
        (is ?scs)
def ses == zenon-seqify (zenon-appseq (zenon-appseq (zenon-appseq (
                      <<>>, e1x), e2x), e3x))
        (is ?ses)
have ha1: ∃ i ∈ DOMAIN ?scs : ?scs[i]
  using ha zenon-case-seq-empty
  by (simp only: zenon-case-seq-simpl zenon-seqify-empty, blast)
have ha2: ∃ i ∈ DOMAIN ?cs : ?cs[i]
  using ha1 by (simp only: zenon-seqify-appseq zenon-seqify-empty)
have hb: \E x : x \in arms
  using zenon-case-domain [OF ha2, where es = ?es]
  by (unfold arms-def, blast)
have hc: (CHOOSE x : x \in arms) \in arms
  using hb by (unfold Ex-def, auto)
have hf0: ?cas \in arms
  using hc by (unfold arms-def, fold Case-def)
have hf3: cas \in UNION {CaseArm (?scs[i], ?ses[i])
                        : i \in DOMAIN ?scs}
       (is ?hf3)
  using hf0 by (fold cas-def, unfold arms-def,
           simp only: zenon-seqify-appseq zenon-seqify-empty)
have hf4: Len (?scs) = Len (?ses) (is ?hf4)
  by (simp only: zenon-case-len-simpl)
have hf5: ?hf4 & ?hf3
  by (rule conjI [OF hf4 hf3])
have hf:
        cas \in CaseArm (c3x, e3x)
      | cas \in CaseArm (c2x, e2x)
      | cas \in CaseArm (c1x, e1x)
      | cas \in UNION {CaseArm (zenon-seqify (<<>>)[i],
                               zenon-seqify (<<>>)[i])
                        : i \in DOMAIN zenon-seqify (<<>>)}
       (is - | ?gxx)
  using hf5 by (simp only: zenon-case-union-simpl, blast)
have hg1x: cas \in CaseArm (c1x, e1x) => FALSE
  using h0 h1 by auto
have hg2x: cas \in CaseArm (c2x, e2x) => FALSE
  using h0 h2 by auto
have hg3x: cas \in CaseArm (c3x, e3x) => FALSE
  using h0 h3 by auto
from hf
have hh0: ?gxx (is - | ?g1)
  by (rule zenon-disjE1 [OF - hg3x])
then have hh1: ?g1 (is - | ?g0)
  by (rule zenon-disjE1 [OF - hg2x])
then have hh0: ?g0
```

```
      by (rule zenon-disjE1 [OF - hg1x])
    have hi: cas \in UNION {CaseArm (<<>>[i], <<>>[i])
                                : i \in DOMAIN <<>>}
      using hh0 by (simp only: zenon-seqify-empty)
    show FALSE
      by (rule zenon-case-empty-union [OF hi])
  qed
qed

lemma zenon-caseother3 :
  fixes P oth c1x e1x c2x e2x c3x e3x
  assumes h: P (CASE c1x −> e1x [] c2x −> e2x [] c3x −> e3x
                [] OTHER −> oth)
             (is P (?cas))
  assumes h1: c1x ==> P (e1x) ==> FALSE
  assumes h2: c2x ==> P (e2x) ==> FALSE
  assumes h3: c3x ==> P (e3x) ==> FALSE
  assumes hoth: ~c3x & ~c2x & ~c1x & TRUE ==> P (oth) ==> FALSE
  shows FALSE
proof −
  def cs == <<c1x, c2x, c3x>> (is ?cs)
  def es == <<e1x, e2x, e3x>> (is ?es)
  def arms == UNION {CaseArm (?cs[i], ?es[i]) : i \in DOMAIN ?cs}
              (is ?arms)
  def cas == ?cas
  have h0: P (cas) using h by (fold cas-def)
  def dcs == c3x | c2x | c1x | FALSE (is ?dcs)
  def scs == zenon-seqify (zenon-appseq (zenon-appseq (zenon-appseq (
                          <<>>, c1x), c2x), c3x))
             (is ?scs)
  have hscs : ?cs = ?scs
    by (simp only: zenon-seqify-appseq zenon-seqify-empty)
  def ses == zenon-seqify (zenon-appseq (zenon-appseq (zenon-appseq (
                          <<>>, e1x), e2x), e3x))
             (is ?ses)
  have hses : ?es = ?ses
    by (simp only: zenon-seqify-appseq zenon-seqify-empty)
  have hlen: Len (?scs) = Len (?ses) (is ?hlen)
    by (simp only: zenon-case-len-simpl)
  def armoth == CaseArm (∀ i ∈ DOMAIN ?cs : ~?cs[i], oth)
                (is ?armoth)
  show FALSE
  proof (rule zenon-em [of ?dcs])
    assume ha: ~(?dcs)
    have hb: P (CHOOSE x : x \in arms ∪ armoth)
      using h by (unfold CaseOther-def, fold arms-def armoth-def)
    have hc: arms \cup armoth
               = UNION {CaseArm (?scs[i], ?ses[i]) : i \in DOMAIN ?scs}
                 \cup CaseArm (∀ i ∈ DOMAIN ?scs : ~?scs[i],
```

```
                              oth)
          (is - = ?sarmsoth)
    using hscs hses by (unfold arms-def armoth-def, auto)
  have hd: ~(?dcs) & ?hlen & arms ∪ armoth = ?sarmsoth
    using ha hlen hc by blast
  have he: arms ∪ armoth = {oth}
    using hd by (simp only: zenon-case-oth-simpl zenon-case-oth-empty)
  have hf: (CHOOSE x : x \in arms \cup armoth) = oth
    using he by auto
  have hg: P (oth)
    using hb hf by auto
  have hh1: ~c1x using ha by blast
  have hh2: ~c2x using ha by blast
  have hh3: ~c3x using ha by blast
  show FALSE
    using hg hoth hh1 hh2 hh3 by blast
next
  assume ha: ?dcs
  have ha1: ∃ i ∈ DOMAIN ?scs : ?scs[i]
    using ha zenon-case-seq-empty
    by (simp only: zenon-case-seq-simpl zenon-seqify-empty, blast)
  have ha2: ∃ i ∈ DOMAIN ?cs : ?cs[i]
    using ha1 hscs by auto
  have ha3: ~ (∀ i ∈ DOMAIN ?cs : ~?cs[i])
    using ha2 by blast
  have ha4: armoth = {}
    using ha3 condElse [OF ha3, where t = {oth} and e = {}]
    by (unfold armoth-def CaseArm-def, blast)
  have hb: \E x : x \in arms \cup armoth
    using zenon-case-domain [OF ha2, where es = ?es]
    by (unfold arms-def, blast)
  have hc: (CHOOSE x : x \in arms \cup armoth)
           \in arms \cup armoth
    using hb by (unfold Ex-def, auto)
  have hf0: ?cas \in arms \cup armoth
    using hc by (unfold arms-def armoth-def, fold CaseOther-def)
  have hf1: cas \in arms \cup armoth
    using hf0 by (fold cas-def)
  have hf2: cas \in arms
    using hf1 ha4 by auto
  have hf3: cas \in UNION {CaseArm (?scs[i], ?ses[i])
                              : i \in DOMAIN ?scs}
         (is ?hf3)
    using hf2 by (unfold arms-def,
               simp only: zenon-seqify-appseq zenon-seqify-empty)
  have hf5: ?hlen & ?hf3
    by (rule conjI [OF hlen hf3])
  have hf:
           cas \in CaseArm (c3x, e3x)
```

```
           | cas \in CaseArm (c2x, e2x)
           | cas \in CaseArm (c1x, e1x)
           | cas \in UNION {CaseArm (zenon-seqify (<<>>)[i],
                                     zenon-seqify (<<>>)[i])
                          : i \in DOMAIN zenon-seqify (<<>>)}
           (is - | ?gxx)
      using hf5 by (simp only: zenon-case-union-simpl, blast)
    have hg1x: cas \in CaseArm (c1x, e1x) => FALSE
      using h0 h1 by auto
    have hg2x: cas \in CaseArm (c2x, e2x) => FALSE
      using h0 h2 by auto
    have hg3x: cas \in CaseArm (c3x, e3x) => FALSE
      using h0 h3 by auto
    from hf
    have hh0: ?gxx (is - | ?g1)
      by (rule zenon-disjE1 [OF - hg3x])
    then have hh1: ?g1 (is - | ?g0)
      by (rule zenon-disjE1 [OF - hg2x])
    then have hh0: ?g0
      by (rule zenon-disjE1 [OF - hg1x])
    have hi: cas \in UNION {CaseArm (<<>>[i], <<>>[i])
                          : i \in DOMAIN <<>>}
      using hh0 by (simp only: zenon-seqify-empty)
    show FALSE
      by (rule zenon-case-empty-union [OF hi])
  qed
qed

lemma zenon-case4 :
  fixes P c1x e1x c2x e2x c3x e3x c4x e4x
  assumes h: P (CASE c1x -> e1x [] c2x -> e2x [] c3x -> e3x [] c4x -> e4x)
           (is P (?cas))
  assumes h1: c1x ==> P (e1x) ==> FALSE
  assumes h2: c2x ==> P (e2x) ==> FALSE
  assumes h3: c3x ==> P (e3x) ==> FALSE
  assumes h4: c4x ==> P (e4x) ==> FALSE
  assumes hoth: ~c4x & ~c3x & ~c2x & ~c1x & TRUE ==> FALSE
  shows FALSE
proof -
  def cs == <<c1x, c2x, c3x, c4x>> (is ?cs)
  def es == <<e1x, e2x, e3x, e4x>> (is ?es)
  def arms == UNION {CaseArm (?cs[i], ?es[i]) : i \in DOMAIN ?cs}
           (is ?arms)
  def cas == ?cas
  have h0: P (cas) using h by (fold cas-def)
  def dcs == c4x | c3x | c2x | c1x (is ?dcs)
  show FALSE
  proof (rule zenon-em [of ?dcs])
    assume ha: ~(?dcs)
```

have *hh1*: ~*c1x* **using** *ha* **by** *blast*
have *hh2*: ~*c2x* **using** *ha* **by** *blast*
have *hh3*: ~*c3x* **using** *ha* **by** *blast*
have *hh4*: ~*c4x* **using** *ha* **by** *blast*
show *FALSE*
using *hoth hh1 hh2 hh3 hh4* **by** *blast*
next
assume *ha*: *?dcs*
def *scs* == *zenon-seqify* (*zenon-appseq* (*zenon-appseq* (*zenon-appseq* (*zenon-appseq*
(
<<>>, *c1x*), *c2x*), *c3x*), *c4x*))
(**is** *?scs*)
def *ses* == *zenon-seqify* (*zenon-appseq* (*zenon-appseq* (*zenon-appseq* (*zenon-appseq*
(
<<>>, *e1x*), *e2x*), *e3x*), *e4x*))
(**is** *?ses*)
have *ha1*: ∃ *i* ∈ *DOMAIN ?scs* : *?scs*[*i*]
using *ha zenon-case-seq-empty*
by (*simp only*: *zenon-case-seq-simpl zenon-seqify-empty*, *blast*)
have *ha2*: ∃ *i* ∈ *DOMAIN ?cs* : *?cs*[*i*]
using *ha1* **by** (*simp only*: *zenon-seqify-appseq zenon-seqify-empty*)
have *hb*: \E *x* : *x* \in *arms*
using *zenon-case-domain* [*OF ha2*, **where** *es* = *?es*]
by (*unfold arms-def*, *blast*)
have *hc*: (*CHOOSE x* : *x* \in *arms*) \in *arms*
using *hb* **by** (*unfold Ex-def*, *auto*)
have *hf0*: *?cas* \in *arms*
using *hc* **by** (*unfold arms-def*, *fold Case-def*)
have *hf3*: *cas* \in *UNION* {*CaseArm* (*?scs*[*i*], *?ses*[*i*])
: *i* \in *DOMAIN ?scs*}
(**is** *?hf3*)
using *hf0* **by** (*fold cas-def*, *unfold arms-def*,
simp only: *zenon-seqify-appseq zenon-seqify-empty*)
have *hf4*: *Len* (*?scs*) = *Len* (*?ses*) (**is** *?hf4*)
by (*simp only*: *zenon-case-len-simpl*)
have *hf5*: *?hf4* & *?hf3*
by (*rule conjI* [*OF hf4 hf3*])
have *hf*:
cas \in *CaseArm* (*c4x*, *e4x*)
| *cas* \in *CaseArm* (*c3x*, *e3x*)
| *cas* \in *CaseArm* (*c2x*, *e2x*)
| *cas* \in *CaseArm* (*c1x*, *e1x*)
| *cas* \in *UNION* {*CaseArm* (*zenon-seqify* (<<>>)[*i*],
zenon-seqify (<<>>)[*i*])
: *i* \in *DOMAIN zenon-seqify* (<<>>)}
(**is** - | *?gxx*)
using *hf5* **by** (*simp only*: *zenon-case-union-simpl*, *blast*)
have *hg1x*: *cas* \in *CaseArm* (*c1x*, *e1x*) => *FALSE*
using *h0 h1* **by** *auto*

```
    have hg2x: cas \in CaseArm (c2x, e2x) => FALSE
      using h0 h2 by auto
    have hg3x: cas \in CaseArm (c3x, e3x) => FALSE
      using h0 h3 by auto
    have hg4x: cas \in CaseArm (c4x, e4x) => FALSE
      using h0 h4 by auto
    from hf
    have hh0: ?gxx (is - | ?g2)
      by (rule zenon-disjE1 [OF - hg4x])
    then have hh2: ?g2 (is - | ?g1)
      by (rule zenon-disjE1 [OF - hg3x])
    then have hh1: ?g1 (is - | ?g0)
      by (rule zenon-disjE1 [OF - hg2x])
    then have hh0: ?g0
      by (rule zenon-disjE1 [OF - hg1x])
    have hi: cas \in UNION {CaseArm (<<>>[i], <<>>[i])
                              : i \in DOMAIN <<>>}
      using hh0 by (simp only: zenon-seqify-empty)
    show FALSE
      by (rule zenon-case-empty-union [OF hi])
  qed
qed

lemma zenon-caseother4 :
  fixes P oth c1x e1x c2x e2x c3x e3x c4x e4x
  assumes h: P (CASE c1x -> e1x [] c2x -> e2x [] c3x -> e3x [] c4x -> e4x
                 [] OTHER -> oth)
             (is P (?cas))
  assumes h1: c1x ==> P (e1x) ==> FALSE
  assumes h2: c2x ==> P (e2x) ==> FALSE
  assumes h3: c3x ==> P (e3x) ==> FALSE
  assumes h4: c4x ==> P (e4x) ==> FALSE
  assumes hoth: ~c4x & ~c3x & ~c2x & ~c1x & TRUE ==> P (oth) ==>
FALSE
  shows FALSE
proof -
  def cs == <<c1x, c2x, c3x, c4x>> (is ?cs)
  def es == <<e1x, e2x, e3x, e4x>> (is ?es)
  def arms == UNION {CaseArm (?cs[i], ?es[i]) : i \in DOMAIN ?cs}
             (is ?arms)
  def cas == ?cas
  have h0: P (cas) using h by (fold cas-def)
  def dcs == c4x | c3x | c2x | c1x | FALSE (is ?dcs)
  def scs == zenon-seqify (zenon-appseq (zenon-appseq (zenon-appseq (zenon-appseq
(
                            <<>>, c1x), c2x), c3x), c4x))
             (is ?scs)
  have hscs : ?cs = ?scs
    by (simp only: zenon-seqify-appseq zenon-seqify-empty)
```

def *ses* == *zenon-seqify* (*zenon-appseq* (*zenon-appseq* (*zenon-appseq* (*zenon-appseq*
(
<<>>, *e1x*), *e2x*), *e3x*), *e4x*))
(**is** *?ses*)
have *hses* : *?es* = *?ses*
by (*simp only*: *zenon-seqify-appseq zenon-seqify-empty*)
have *hlen*: *Len* (*?scs*) = *Len* (*?ses*) (**is** *?hlen*)
by (*simp only*: *zenon-case-len-simpl*)
def *armoth* == *CaseArm* (∀ *i* ∈ *DOMAIN ?cs* : ~*?cs*[*i*], *oth*)
(**is** *?armoth*)
show *FALSE*
proof (*rule zenon-em* [*of ?dcs*])
assume *ha*: ~(*?dcs*)
have *hb*: *P* (*CHOOSE x* : *x* \in *arms* ∪ *armoth*)
using *h* **by** (*unfold CaseOther-def*, *fold arms-def armoth-def*)
have *hc*: *arms* \cup *armoth*
= *UNION* {*CaseArm* (*?scs*[*i*], *?ses*[*i*]) : *i* \in *DOMAIN ?scs*}
\cup *CaseArm* (∀ *i* ∈ *DOMAIN ?scs* : ~*?scs*[*i*],
oth)
(**is** - = *?sarmsoth*)
using *hscs hses* **by** (*unfold arms-def armoth-def*, *auto*)
have *hd*: ~(*?dcs*) & *?hlen* & *arms* ∪ *armoth* = *?sarmsoth*
using *ha hlen hc* **by** *blast*
have *he*: *arms* ∪ *armoth* = {*oth*}
using *hd* **by** (*simp only*: *zenon-case-oth-simpl zenon-case-oth-empty*)
have *hf*: (*CHOOSE x* : *x* \in *arms* \cup *armoth*) = *oth*
using *he* **by** *auto*
have *hg*: *P* (*oth*)
using *hb hf* **by** *auto*
have *hh1*: ~*c1x* **using** *ha* **by** *blast*
have *hh2*: ~*c2x* **using** *ha* **by** *blast*
have *hh3*: ~*c3x* **using** *ha* **by** *blast*
have *hh4*: ~*c4x* **using** *ha* **by** *blast*
show *FALSE*
using *hg hoth hh1 hh2 hh3 hh4* **by** *blast*
next
assume *ha*: *?dcs*
have *ha1*: ∃ *i* ∈ *DOMAIN ?scs* : *?scs*[*i*]
using *ha zenon-case-seq-empty*
by (*simp only*: *zenon-case-seq-simpl zenon-seqify-empty*, *blast*)
have *ha2*: ∃ *i* ∈ *DOMAIN ?cs* : *?cs*[*i*]
using *ha1 hscs* **by** *auto*
have *ha3*: ~ (∀ *i* ∈ *DOMAIN ?cs* : ~*?cs*[*i*])
using *ha2* **by** *blast*
have *ha4*: *armoth* = {}
using *ha3 condElse* [*OF ha3*, **where** *t* = {*oth*} **and** *e* = {}]
by (*unfold armoth-def CaseArm-def*, *blast*)
have *hb*: \E *x* : *x* \in *arms* \cup *armoth*
using *zenon-case-domain* [*OF ha2*, **where** *es* = *?es*]

```
  by (unfold arms-def, blast)
have hc: (CHOOSE x : x \in arms \cup armoth)
           \in arms \cup armoth
  using hb by (unfold Ex-def, auto)
have hf0: ?cas \in arms \cup armoth
  using hc by (unfold arms-def armoth-def, fold CaseOther-def)
have hf1: cas \in arms \cup armoth
  using hf0 by (fold cas-def)
have hf2: cas \in arms
  using hf1 ha4 by auto
have hf3: cas \in UNION {CaseArm (?scs[i], ?ses[i])
                              : i \in DOMAIN ?scs}
          (is ?hf3)
  using hf2 by (unfold arms-def,
              simp only: zenon-seqify-appseq zenon-seqify-empty)
have hf5: ?hlen & ?hf3
  by (rule conjI [OF hlen hf3])
have hf:
           cas \in CaseArm (c4x, e4x)
         | cas \in CaseArm (c3x, e3x)
         | cas \in CaseArm (c2x, e2x)
         | cas \in CaseArm (c1x, e1x)
         | cas \in UNION {CaseArm (zenon-seqify (<<>>)[i],
                                   zenon-seqify (<<>>)[i])
                             : i \in DOMAIN zenon-seqify (<<>>)}
          (is - | ?gxx)
  using hf5 by (simp only: zenon-case-union-simpl, blast)
have hg1x: cas \in CaseArm (c1x, e1x) => FALSE
  using h0 h1 by auto
have hg2x: cas \in CaseArm (c2x, e2x) => FALSE
  using h0 h2 by auto
have hg3x: cas \in CaseArm (c3x, e3x) => FALSE
  using h0 h3 by auto
have hg4x: cas \in CaseArm (c4x, e4x) => FALSE
  using h0 h4 by auto
from hf
have hh0: ?gxx (is - | ?g2)
  by (rule zenon-disjE1 [OF - hg4x])
then have hh2: ?g2 (is - | ?g1)
  by (rule zenon-disjE1 [OF - hg3x])
then have hh1: ?g1 (is - | ?g0)
  by (rule zenon-disjE1 [OF - hg2x])
then have hh0: ?g0
  by (rule zenon-disjE1 [OF - hg1x])
have hi: cas \in UNION {CaseArm (<<>>[i], <<>>[i])
                             : i \in DOMAIN <<>>}
  using hh0 by (simp only: zenon-seqify-empty)
show FALSE
  by (rule zenon-case-empty-union [OF hi])
```

```
  qed
qed

lemma zenon-case5 :
  fixes P c1x e1x c2x e2x c3x e3x c4x e4x c5x e5x
  assumes h: P (CASE c1x -> e1x [] c2x -> e2x [] c3x -> e3x [] c4x -> e4x
[] c5x -> e5x)
            (is P (?cas))
  assumes h1: c1x ==> P (e1x) ==> FALSE
  assumes h2: c2x ==> P (e2x) ==> FALSE
  assumes h3: c3x ==> P (e3x) ==> FALSE
  assumes h4: c4x ==> P (e4x) ==> FALSE
  assumes h5: c5x ==> P (e5x) ==> FALSE
  assumes hoth: ~c5x & ~c4x & ~c3x & ~c2x & ~c1x & TRUE ==> FALSE
  shows FALSE
proof -
  def cs == <<c1x, c2x, c3x, c4x, c5x>> (is ?cs)
  def es == <<e1x, e2x, e3x, e4x, e5x>> (is ?es)
  def arms == UNION {CaseArm (?cs[i], ?es[i]) : i \in DOMAIN ?cs}
             (is ?arms)
  def cas == ?cas
  have h0: P (cas) using h by (fold cas-def)
  def dcs == c5x | c4x | c3x | c2x | c1x (is ?dcs)
  show FALSE
  proof (rule zenon-em [of ?dcs])
    assume ha: ~(?dcs)
    have hh1: ~c1x using ha by blast
    have hh2: ~c2x using ha by blast
    have hh3: ~c3x using ha by blast
    have hh4: ~c4x using ha by blast
    have hh5: ~c5x using ha by blast
    show FALSE
      using hoth hh1 hh2 hh3 hh4 hh5 by blast
  next
    assume ha: ?dcs
   def scs == zenon-seqify (zenon-appseq (zenon-appseq (zenon-appseq (zenon-appseq
(zenon-appseq (
                      <<>>, c1x), c2x), c3x), c4x), c5x))
              (is ?scs)
   def ses == zenon-seqify (zenon-appseq (zenon-appseq (zenon-appseq (zenon-appseq
(zenon-appseq (
                      <<>>, e1x), e2x), e3x), e4x), e5x))
              (is ?ses)
    have ha1: ∃ i ∈ DOMAIN ?scs : ?scs[i]
      using ha zenon-case-seq-empty
      by (simp only: zenon-case-seq-simpl zenon-seqify-empty, blast)
    have ha2: ∃ i ∈ DOMAIN ?cs : ?cs[i]
      using ha1 by (simp only: zenon-seqify-appseq zenon-seqify-empty)
    have hb: \E x : x \in arms
```

```
  using zenon-case-domain [OF ha2, where es = ?es]
  by (unfold arms-def, blast)
have hc: (CHOOSE x : x \in arms) \in arms
  using hb by (unfold Ex-def, auto)
have hf0: ?cas \in arms
  using hc by (unfold arms-def, fold Case-def)
have hf3: cas \in UNION {CaseArm (?scs[i], ?ses[i])
                          : i \in DOMAIN ?scs}
        (is ?hf3)
  using hf0 by (fold cas-def, unfold arms-def,
              simp only: zenon-seqify-appseq zenon-seqify-empty)
have hf4: Len (?scs) = Len (?ses) (is ?hf4)
  by (simp only: zenon-case-len-simpl)
have hf5: ?hf4 & ?hf3
  by (rule conjI [OF hf4 hf3])
have hf:
          cas \in CaseArm (c5x, e5x)
        | cas \in CaseArm (c4x, e4x)
        | cas \in CaseArm (c3x, e3x)
        | cas \in CaseArm (c2x, e2x)
        | cas \in CaseArm (c1x, e1x)
        | cas \in UNION {CaseArm (zenon-seqify (<<>>)[i],
                                 zenon-seqify (<<>>)[i])
                          : i \in DOMAIN zenon-seqify (<<>>)}
        (is - | ?gxx)
  using hf5 by (simp only: zenon-case-union-simpl, blast)
have hg1x: cas \in CaseArm (c1x, e1x) => FALSE
  using h0 h1 by auto
have hg2x: cas \in CaseArm (c2x, e2x) => FALSE
  using h0 h2 by auto
have hg3x: cas \in CaseArm (c3x, e3x) => FALSE
  using h0 h3 by auto
have hg4x: cas \in CaseArm (c4x, e4x) => FALSE
  using h0 h4 by auto
have hg5x: cas \in CaseArm (c5x, e5x) => FALSE
  using h0 h5 by auto
from hf
have hh0: ?gxx (is - | ?g3)
  by (rule zenon-disjE1 [OF - hg5x])
then have hh3: ?g3 (is - | ?g2)
  by (rule zenon-disjE1 [OF - hg4x])
then have hh2: ?g2 (is - | ?g1)
  by (rule zenon-disjE1 [OF - hg3x])
then have hh1: ?g1 (is - | ?g0)
  by (rule zenon-disjE1 [OF - hg2x])
then have hh0: ?g0
  by (rule zenon-disjE1 [OF - hg1x])
have hi: cas \in UNION {CaseArm (<<>>[i], <<>>[i])
                        : i \in DOMAIN <<>>}
```

```
    using hh0 by (simp only: zenon-seqify-empty)
  show FALSE
    by (rule zenon-case-empty-union [OF hi])
 qed
qed

lemma zenon-caseother5 :
 fixes P oth c1x e1x c2x e2x c3x e3x c4x e4x c5x e5x
 assumes h: P (CASE c1x -> e1x [] c2x -> e2x [] c3x -> e3x [] c4x -> e4x
[] c5x -> e5x
          [] OTHER -> oth)
       (is P (?cas))
 assumes h1: c1x ==> P (e1x) ==> FALSE
 assumes h2: c2x ==> P (e2x) ==> FALSE
 assumes h3: c3x ==> P (e3x) ==> FALSE
 assumes h4: c4x ==> P (e4x) ==> FALSE
 assumes h5: c5x ==> P (e5x) ==> FALSE
 assumes hoth: ~c5x & ~c4x & ~c3x & ~c2x & ~c1x & TRUE ==> P (oth)
==> FALSE
 shows FALSE
proof -
 def cs == <<c1x, c2x, c3x, c4x, c5x>> (is ?cs)
 def es == <<e1x, e2x, e3x, e4x, e5x>> (is ?es)
 def arms == UNION {CaseArm (?cs[i], ?es[i]) : i \in DOMAIN ?cs}
        (is ?arms)
 def cas == ?cas
 have h0: P (cas) using h by (fold cas-def)
 def dcs == c5x | c4x | c3x | c2x | c1x | FALSE (is ?dcs)
 def scs == zenon-seqify (zenon-appseq (zenon-appseq (zenon-appseq (zenon-appseq
(zenon-appseq (
             <<>>, c1x), c2x), c3x), c4x), c5x))
       (is ?scs)
 have hscs : ?cs = ?scs
  by (simp only: zenon-seqify-appseq zenon-seqify-empty)
 def ses == zenon-seqify (zenon-appseq (zenon-appseq (zenon-appseq (zenon-appseq
(zenon-appseq (
             <<>>, e1x), e2x), e3x), e4x), e5x))
       (is ?ses)
 have hses : ?es = ?ses
  by (simp only: zenon-seqify-appseq zenon-seqify-empty)
 have hlen: Len (?scs) = Len (?ses) (is ?hlen)
  by (simp only: zenon-case-len-simpl)
 def armoth == CaseArm (∀ i ∈ DOMAIN ?cs : ~?cs[i], oth)
         (is ?armoth)
 show FALSE
 proof (rule zenon-em [of ?dcs])
  assume ha: ~(?dcs)
  have hb: P (CHOOSE x : x \in arms ∪ armoth)
    using h by (unfold CaseOther-def, fold arms-def armoth-def)
```

have *hc*: *arms* \cup *armoth*
= *UNION* {*CaseArm* (*?scs*[*i*], *?ses*[*i*]) : *i* \in *DOMAIN* *?scs*}
\cup *CaseArm* (∀ *i* ∈ *DOMAIN* *?scs* : ~*?scs*[*i*],
oth)
(**is** - = *?sarmsoth*)
using *hscs hses* **by** (*unfold arms-def armoth-def*, *auto*)
have *hd*: ~(*?dcs*) & *?hlen* & *arms* ∪ *armoth* = *?sarmsoth*
using *ha hlen hc* **by** *blast*
have *he*: *arms* ∪ *armoth* = {*oth*}
using *hd* **by** (*simp only*: *zenon-case-oth-simpl zenon-case-oth-empty*)
have *hf*: (*CHOOSE* *x* : *x* \in *arms* \cup *armoth*) = *oth*
using *he* **by** *auto*
have *hg*: *P* (*oth*)
using *hb hf* **by** *auto*
have *hh1*: ~*c1x* **using** *ha* **by** *blast*
have *hh2*: ~*c2x* **using** *ha* **by** *blast*
have *hh3*: ~*c3x* **using** *ha* **by** *blast*
have *hh4*: ~*c4x* **using** *ha* **by** *blast*
have *hh5*: ~*c5x* **using** *ha* **by** *blast*
show *FALSE*
using *hg hoth hh1 hh2 hh3 hh4 hh5* **by** *blast*
next
assume *ha*: *?dcs*
have *ha1*: ∃ *i* ∈ *DOMAIN* *?scs* : *?scs*[*i*]
using *ha zenon-case-seq-empty*
by (*simp only*: *zenon-case-seq-simpl zenon-seqify-empty*, *blast*)
have *ha2*: ∃ *i* ∈ *DOMAIN* *?cs* : *?cs*[*i*]
using *ha1 hscs* **by** *auto*
have *ha3*: ~ (∀ *i* ∈ *DOMAIN* *?cs* : ~*?cs*[*i*])
using *ha2* **by** *blast*
have *ha4*: *armoth* = {}
using *ha3 condElse* [*OF ha3*, **where** *t* = {*oth*} **and** *e* = {}]
by (*unfold armoth-def CaseArm-def*, *blast*)
have *hb*: \E *x* : *x* \in *arms* \cup *armoth*
using *zenon-case-domain* [*OF ha2*, **where** *es* = *?es*]
by (*unfold arms-def*, *blast*)
have *hc*: (*CHOOSE* *x* : *x* \in *arms* \cup *armoth*)
\in *arms* \cup *armoth*
using *hb* **by** (*unfold Ex-def*, *auto*)
have *hf0*: *?cas* \in *arms* \cup *armoth*
using *hc* **by** (*unfold arms-def armoth-def*, *fold CaseOther-def*)
have *hf1*: *cas* \in *arms* \cup *armoth*
using *hf0* **by** (*fold cas-def*)
have *hf2*: *cas* \in *arms*
using *hf1 ha4* **by** *auto*
have *hf3*: *cas* \in *UNION* {*CaseArm* (*?scs*[*i*], *?ses*[*i*])
: *i* \in *DOMAIN* *?scs*}
(**is** *?hf3*)
using *hf2* **by** (*unfold arms-def*,

```
                simp only: zenon-seqify-appseq zenon-seqify-empty)
    have hf5: ?hlen & ?hf3
      by (rule conjI [OF hlen hf3])
    have hf:
              cas \in CaseArm (c5x, e5x)
            | cas \in CaseArm (c4x, e4x)
            | cas \in CaseArm (c3x, e3x)
            | cas \in CaseArm (c2x, e2x)
            | cas \in CaseArm (c1x, e1x)
            | cas \in UNION {CaseArm (zenon-seqify (<<>>)[i],
                                   zenon-seqify (<<>>)[i])
                            : i \in DOMAIN zenon-seqify (<<>>)}
           (is - | ?gxx)
      using hf5 by (simp only: zenon-case-union-simpl, blast)
    have hg1x: cas \in CaseArm (c1x, e1x) => FALSE
      using h0 h1 by auto
    have hg2x: cas \in CaseArm (c2x, e2x) => FALSE
      using h0 h2 by auto
    have hg3x: cas \in CaseArm (c3x, e3x) => FALSE
      using h0 h3 by auto
    have hg4x: cas \in CaseArm (c4x, e4x) => FALSE
      using h0 h4 by auto
    have hg5x: cas \in CaseArm (c5x, e5x) => FALSE
      using h0 h5 by auto
    from hf
    have hh0: ?gxx (is - | ?g3)
      by (rule zenon-disjE1 [OF - hg5x])
    then have hh3: ?g3 (is - | ?g2)
      by (rule zenon-disjE1 [OF - hg4x])
    then have hh2: ?g2 (is - | ?g1)
      by (rule zenon-disjE1 [OF - hg3x])
    then have hh1: ?g1 (is - | ?g0)
      by (rule zenon-disjE1 [OF - hg2x])
    then have hh0: ?g0
      by (rule zenon-disjE1 [OF - hg1x])
    have hi: cas \in UNION {CaseArm (<<>>[i], <<>>[i])
                            : i \in DOMAIN <<>>}
      using hh0 by (simp only: zenon-seqify-empty)
    show FALSE
      by (rule zenon-case-empty-union [OF hi])
  qed
qed


end
```