The Supplemental Isabelle/HOL Library

October 1, 2005

Contents

1 Accessible-Part: The accessible part of a relation

theory *Accessible-Part*
imports *Main*
begin

1.1 Inductive definition

Inductive definition of the accessible part *acc r* of a relation; see also [4].

consts
  $acc :: ('a \times 'a)\ set \Rightarrow 'a\ set$
inductive *acc r*
  intros
    $accI$: $(!!y.\ (y, x) \in r \Longrightarrow y \in acc\ r) \Longrightarrow x \in acc\ r$

syntax
  $termi :: ('a \times 'a)\ set \Rightarrow 'a\ set$
translations
  $termi\ r == acc\ (r^{-1})$

1.2 Induction rules

theorem *acc-induct*:
  $a \in acc\ r \Longrightarrow$
    $(!!x.\ x \in acc\ r \Longrightarrow \forall y.\ (y, x) \in r \longrightarrow P\ y \Longrightarrow P\ x) \Longrightarrow P\ a$
proof −
  assume *major*: $a \in acc\ r$
  assume *hyp*: $!!x.\ x \in acc\ r \Longrightarrow \forall y.\ (y, x) \in r \longrightarrow P\ y \Longrightarrow P\ x$
  show *?thesis*
    apply (*rule major* [*THEN acc.induct*])
    apply (*rule hyp*)
     apply (*rule accI*)
     apply *fast*
    apply *fast*
    done
qed

theorems *acc-induct-rule* = *acc-induct* [*rule-format*, *induct set*: *acc*]

theorem *acc-downward*: $b \in acc\ r \Longrightarrow (a, b) \in r \Longrightarrow a \in acc\ r$
  apply (*erule acc.elims*)
  apply *fast*
  done

lemma *acc-downwards-aux*: $(b, a) \in r^* \Longrightarrow a \in acc\ r \longrightarrow b \in acc\ r$
  apply (*erule rtrancl-induct*)
   apply *blast*
  apply (*blast dest*: *acc-downward*)

done

theorem *acc-downwards*: $a \in acc\ r \Longrightarrow (b, a) \in r^* \Longrightarrow b \in acc\ r$
apply (*blast dest*: *acc-downwards-aux*)
done

theorem *acc-wfI*: $\forall x.\ x \in acc\ r \Longrightarrow wf\ r$
apply (*rule wfUNIVI*)
apply (*induct-tac P x rule*: *acc-induct*)
apply *blast*
apply *blast*
done

theorem *acc-wfD*: $wf\ r \Longrightarrow x \in acc\ r$
apply (*erule wf-induct*)
apply (*rule accI*)
apply *blast*
done

theorem *wf-acc-iff*: $wf\ r = (\forall x.\ x \in acc\ r)$
apply (*blast intro*: *acc-wfI dest*: *acc-wfD*)
done

end

2 SetsAndFunctions: Operations on sets and functions

theory *SetsAndFunctions*
imports *Main*
begin

This library lifts operations like addition and muliplication to sets and functions of appropriate types. It was designed to support asymptotic calculations. See the comments at the top of theory *BigO*.

2.1 Basic definitions

instance *set* :: (*plus*) *plus* **..**
instance *fun* :: (*type*, *plus*) *plus* **..**

defs (**overloaded**)
func-plus: $f + g == (\%x.\ f\ x + g\ x)$
set-plus: $A + B == \{c.\ EX\ a{:}A.\ EX\ b{:}B.\ c = a + b\}$

instance *set* :: (*times*) *times* **..**
instance *fun* :: (*type*, *times*) *times* **..**

defs (**overloaded**)
func-times: $f * g == (\%x.\ f\ x * g\ x)$
set-times:$A * B == \{c.\ EX\ a{:}A.\ EX\ b{:}B.\ c = a * b\}$

instance *fun* :: (*type*, *minus*) *minus* **..**

defs (**overloaded**)
func-minus: $-\ f == (\%x.\ -\ f\ x)$
func-diff: $f\ -\ g == \%x.\ f\ x\ -\ g\ x$

instance *fun* :: (*type*, *zero*) *zero* **..**
instance *set* :: (*zero*) *zero* **..**

defs (**overloaded**)
func-zero: $0{::}(('a{::}type) => ('b{::}zero)) == \%x.\ 0$
set-zero: $0{::}('a{::}zero)set == \{0\}$

instance *fun* :: (*type*, *one*) *one* **..**
instance *set* :: (*one*) *one* **..**

defs (**overloaded**)
func-one: $1{::}(('a{::}type) => ('b{::}one)) == \%x.\ 1$
set-one: $1{::}('a{::}one)set == \{1\}$

constdefs
elt-set-plus :: $'a{::}plus => 'a\ set => 'a\ set$ (**infixl** $+o\ 70$)
$a\ +o\ B == \{c.\ EX\ b{:}B.\ c = a + b\}$

elt-set-times :: $'a{::}times => 'a\ set => 'a\ set$ (**infixl** $*o\ 80$)
$a\ *o\ B == \{c.\ EX\ b{:}B.\ c = a * b\}$

syntax
elt-set-eq :: $'a => 'a\ set => bool$ (**infix** $=o\ 50$)

translations
$x =o A => x : A$

instance *fun* :: (*type*,*semigroup-add*)*semigroup-add*
apply *intro-classes*
apply (*auto simp add*: *func-plus add-assoc*)
done

instance *fun* :: (*type*,*comm-monoid-add*)*comm-monoid-add*
apply *intro-classes*
apply (*auto simp add*: *func-zero func-plus add-ac*)
done

instance *fun* :: (*type*,*ab-group-add*)*ab-group-add*

apply *intro-classes*
apply (*simp add*: *func-minus func-plus func-zero*)
apply (*simp add*: *func-minus func-plus func-diff diff-minus*)
done

instance *fun* :: (*type*,*semigroup-mult*)*semigroup-mult*
apply *intro-classes*
apply (*auto simp add*: *func-times mult-assoc*)
done

instance *fun* :: (*type*,*comm-monoid-mult*)*comm-monoid-mult*
apply *intro-classes*
apply (*auto simp add*: *func-one func-times mult-ac*)
done

instance *fun* :: (*type*,*comm-ring-1*)*comm-ring-1*
apply *intro-classes*
apply (*auto simp add*: *func-plus func-times func-minus func-diff ext*
func-one func-zero ring-eq-simps)
apply (*drule fun-cong*)
apply *simp*
done

instance *set* :: (*semigroup-add*)*semigroup-add*
apply *intro-classes*
apply (*unfold set-plus*)
apply (*force simp add*: *add-assoc*)
done

instance *set* :: (*semigroup-mult*)*semigroup-mult*
apply *intro-classes*
apply (*unfold set-times*)
apply (*force simp add*: *mult-assoc*)
done

instance *set* :: (*comm-monoid-add*)*comm-monoid-add*
apply *intro-classes*
apply (*unfold set-plus*)
apply (*force simp add*: *add-ac*)
apply (*unfold set-zero*)
apply *force*
done

instance *set* :: (*comm-monoid-mult*)*comm-monoid-mult*
apply *intro-classes*
apply (*unfold set-times*)
apply (*force simp add*: *mult-ac*)
apply (*unfold set-one*)
apply *force*

done

2.2 Basic properties

lemma *set-plus-intro* [*intro*]: $a : C ==> b : D ==> a + b : C + D$
by (*auto simp add*: *set-plus*)

lemma *set-plus-intro2* [*intro*]: $b : C ==> a + b : a +o C$
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-plus-rearrange*: $((a::'a::comm\text{-}monoid\text{-}add) +o C) +$
$(b +o D) = (a + b) +o (C + D)$
apply (*auto simp add*: *elt-set-plus-def set-plus add-ac*)
apply (*rule-tac* $x = ba + bb$ **in** *exI*)
apply (*auto simp add*: *add-ac*)
apply (*rule-tac* $x = aa + a$ **in** *exI*)
apply (*auto simp add*: *add-ac*)
done

lemma *set-plus-rearrange2*: $(a::'a::semigroup\text{-}add) +o (b +o C) =$
$(a + b) +o C$
by (*auto simp add*: *elt-set-plus-def add-assoc*)

lemma *set-plus-rearrange3*: $((a::'a::semigroup\text{-}add) +o B) + C =$
$a +o (B + C)$
apply (*auto simp add*: *elt-set-plus-def set-plus*)
apply (*blast intro*: *add-ac*)
apply (*rule-tac* $x = a + aa$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = aa$ **in** *bexI*)
apply *auto*
apply (*rule-tac* $x = ba$ **in** *bexI*)
apply (*auto simp add*: *add-ac*)
done

theorem *set-plus-rearrange4*: $C + ((a::'a::comm\text{-}monoid\text{-}add) +o D) =$
$a +o (C + D)$
apply (*auto intro*!: *subsetI simp add*: *elt-set-plus-def set-plus add-ac*)
apply (*rule-tac* $x = aa + ba$ **in** *exI*)
apply (*auto simp add*: *add-ac*)
done

theorems *set-plus-rearranges* = *set-plus-rearrange set-plus-rearrange2*
set-plus-rearrange3 set-plus-rearrange4

lemma *set-plus-mono* [*intro*!]: $C <= D ==> a +o C <= a +o D$
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-plus-mono2* [*intro*]: $(C::('a::plus)\ set) <= D ==> E <= F ==>$

```
    C + E <= D + F
by (auto simp add: set-plus)

lemma set-plus-mono3 [intro]: a : C ==> a +o D <= C + D
by (auto simp add: elt-set-plus-def set-plus)

lemma set-plus-mono4 [intro]: (a::'a::comm-monoid-add) : C ==>
  a +o D <= D + C
by (auto simp add: elt-set-plus-def set-plus add-ac)

lemma set-plus-mono5: a:C ==> B <= D ==> a +o B <= C + D
  apply (subgoal-tac a +o B <= a +o D)
  apply (erule order-trans)
  apply (erule set-plus-mono3)
  apply (erule set-plus-mono)
done

lemma set-plus-mono-b: C <= D ==> x : a +o C
    ==> x : a +o D
  apply (frule set-plus-mono)
  apply auto
done

lemma set-plus-mono2-b: C <= D ==> E <= F ==> x : C + E ==>
    x : D + F
  apply (frule set-plus-mono2)
  prefer 2
  apply force
  apply assumption
done

lemma set-plus-mono3-b: a : C ==> x : a +o D ==> x : C + D
  apply (frule set-plus-mono3)
  apply auto
done

lemma set-plus-mono4-b: (a::'a::comm-monoid-add) : C ==>
  x : a +o D ==> x : D + C
  apply (frule set-plus-mono4)
  apply auto
done

lemma set-zero-plus [simp]: (0::'a::comm-monoid-add) +o C = C
by (auto simp add: elt-set-plus-def)

lemma set-zero-plus2: (0::'a::comm-monoid-add) : A ==> B <= A + B
  apply (auto intro!: subsetI simp add: set-plus)
  apply (rule-tac x = 0 in bexI)
  apply (rule-tac x = x in bexI)
```

apply (*auto simp add*: *add-ac*)
done

lemma *set-plus-imp-minus*: (*a*::'*a*::*ab-group-add*) : *b* +*o* *C* ==> (*a* − *b*) : *C*
by (*auto simp add*: *elt-set-plus-def add-ac diff-minus*)

lemma *set-minus-imp-plus*: (*a*::'*a*::*ab-group-add*) − *b* : *C* ==> *a* : *b* +*o* *C*
apply (*auto simp add*: *elt-set-plus-def add-ac diff-minus*)
apply (*subgoal-tac* *a* = (*a* + − *b*) + *b*)
apply (*rule bexI*, *assumption*, *assumption*)
apply (*auto simp add*: *add-ac*)
done

lemma *set-minus-plus*: ((*a*::'*a*::*ab-group-add*) − *b* : *C*) = (*a* : *b* +*o* *C*)
by (*rule iffI*, *rule set-minus-imp-plus*, *assumption*, *rule set-plus-imp-minus*,
assumption)

lemma *set-times-intro* [*intro*]: *a* : *C* ==> *b* : *D* ==> *a* ∗ *b* : *C* ∗ *D*
by (*auto simp add*: *set-times*)

lemma *set-times-intro2* [*intro*!]: *b* : *C* ==> *a* ∗ *b* : *a* ∗*o* *C*
by (*auto simp add*: *elt-set-times-def*)

lemma *set-times-rearrange*: ((*a*::'*a*::*comm-monoid-mult*) ∗*o* *C*) ∗
(*b* ∗*o* *D*) = (*a* ∗ *b*) ∗*o* (*C* ∗ *D*)
apply (*auto simp add*: *elt-set-times-def set-times*)
apply (*rule-tac* *x* = *ba* ∗ *bb* **in** *exI*)
apply (*auto simp add*: *mult-ac*)
apply (*rule-tac* *x* = *aa* ∗ *a* **in** *exI*)
apply (*auto simp add*: *mult-ac*)
done

lemma *set-times-rearrange2*: (*a*::'*a*::*semigroup-mult*) ∗*o* (*b* ∗*o* *C*) =
(*a* ∗ *b*) ∗*o* *C*
by (*auto simp add*: *elt-set-times-def mult-assoc*)

lemma *set-times-rearrange3*: ((*a*::'*a*::*semigroup-mult*) ∗*o* *B*) ∗ *C* =
a ∗*o* (*B* ∗ *C*)
apply (*auto simp add*: *elt-set-times-def set-times*)
apply (*blast intro*: *mult-ac*)
apply (*rule-tac* *x* = *a* ∗ *aa* **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* *x* = *aa* **in** *bexI*)
apply *auto*
apply (*rule-tac* *x* = *ba* **in** *bexI*)
apply (*auto simp add*: *mult-ac*)
done

theorem *set-times-rearrange4*: *C* ∗ ((*a*::'*a*::*comm-monoid-mult*) ∗*o* *D*) =

```
    a *o (C * D)
  apply (auto intro!: subsetI simp add: elt-set-times-def set-times
    mult-ac)
  apply (rule-tac x = aa * ba in exI)
  apply (auto simp add: mult-ac)
done

theorems set-times-rearranges = set-times-rearrange set-times-rearrange2
  set-times-rearrange3 set-times-rearrange4

lemma set-times-mono [intro]: C <= D ==> a *o C <= a *o D
by (auto simp add: elt-set-times-def)

lemma set-times-mono2 [intro]: (C::('a::times) set) <= D ==> E <= F ==>
    C * E <= D * F
by (auto simp add: set-times)

lemma set-times-mono3 [intro]: a : C ==> a *o D <= C * D
by (auto simp add: elt-set-times-def set-times)

lemma set-times-mono4 [intro]: (a::'a::comm-monoid-mult) : C ==>
  a *o D <= D * C
by (auto simp add: elt-set-times-def set-times mult-ac)

lemma set-times-mono5: a:C ==> B <= D ==> a *o B <= C * D
  apply (subgoal-tac a *o B <= a *o D)
  apply (erule order-trans)
  apply (erule set-times-mono3)
  apply (erule set-times-mono)
done

lemma set-times-mono-b: C <= D ==> x : a *o C
    ==> x : a *o D
  apply (frule set-times-mono)
  apply auto
done

lemma set-times-mono2-b: C <= D ==> E <= F ==> x : C * E ==>
    x : D * F
  apply (frule set-times-mono2)
  prefer 2
  apply force
  apply assumption
done

lemma set-times-mono3-b: a : C ==> x : a *o D ==> x : C * D
  apply (frule set-times-mono3)
  apply auto
done
```

lemma *set-times-mono4-b*: (*a*::'*a*::*comm-monoid-mult*) : *C* ==>
x : *a* *o *D* ==> *x* : *D* * *C*
apply (*frule set-times-mono4*)
apply *auto*
done

lemma *set-one-times* [*simp*]: (*1*::'*a*::*comm-monoid-mult*) *o *C* = *C*
by (*auto simp add*: *elt-set-times-def*)

lemma *set-times-plus-distrib*: (*a*::'*a*::*semiring*) *o (*b* +o *C*)=
(*a* * *b*) +o (*a* *o *C*)
by (*auto simp add*: *elt-set-plus-def elt-set-times-def ring-distrib*)

lemma *set-times-plus-distrib2*: (*a*::'*a*::*semiring*) *o (*B* + *C*) =
(*a* *o *B*) + (*a* *o *C*)
apply (*auto simp add*: *set-plus elt-set-times-def ring-distrib*)
apply *blast*
apply (*rule-tac x* = *b* + *bb* **in** *exI*)
apply (*auto simp add*: *ring-distrib*)
done

lemma *set-times-plus-distrib3*: ((*a*::'*a*::*semiring*) +o *C*) * *D* <=
a *o *D* + *C* * *D*
apply (*auto intro*!: *subsetI simp add*:
elt-set-plus-def elt-set-times-def set-times
set-plus ring-distrib)
apply *auto*
done

theorems *set-times-plus-distribs* = *set-times-plus-distrib*
set-times-plus-distrib2

lemma *set-neg-intro*: (*a*::'*a*::*ring-1*) : (− *1*) *o *C* ==>
− *a* : *C*
by (*auto simp add*: *elt-set-times-def*)

lemma *set-neg-intro2*: (*a*::'*a*::*ring-1*) : *C* ==>
− *a* : (− *1*) *o *C*
by (*auto simp add*: *elt-set-times-def*)

end

3 BigO: Big O notation

theory *BigO*
imports *SetsAndFunctions*
begin

This library is designed to support asymptotic "big O" calculations, i.e. reasoning with expressions of the form $f = O(g)$ and $f = g + O(h)$. An earlier version of this library is described in detail in [2].
The main changes in this version are as follows:

- We have eliminated the O operator on sets. (Most uses of this seem to be inessential.)
- We no longer use + as output syntax for $+o$
- Lemmas involving *sumr* have been replaced by more general lemmas involving '*setsum*.
- The library has been expanded, with e.g. support for expressions of the form $f < g + O(h)$.

See `Complex/ex/BigO_Complex.thy` for additional lemmas that require the `HOL-Complex` logic image.
Note also since the Big O library includes rules that demonstrate set inclusion, to use the automated reasoners effectively with the library one should redeclare the theorem *subsetI* as an intro rule, rather than as an *intro*! rule, for example, using **declare** *subsetI* [*del*, *intro*].

3.1 Definitions

constdefs

bigo :: ('*a* => '*b*::*ordered-idom*) => ('*a* => '*b*) *set* ((*1O'*(-')))
O(*f*::('*a* => '*b*)) ==
  {*h*. *EX c*. *ALL x*. *abs* (*h x*) <= *c* * *abs* (*f x*)}

lemma *bigo-pos-const*: (*EX* (*c*::'*a*::*ordered-idom*).
  ALL x. (*abs* (*h x*)) <= (*c* * (*abs* (*f x*))))
  = (*EX c*. *0* < *c* & (*ALL x*. (*abs*(*h x*)) <= (*c* * (*abs* (*f x*))))))
 apply *auto*
 apply (*case-tac c* = *0*)
 apply *simp*
 apply (*rule-tac x* = *1* **in** *exI*)
 apply *simp*
 apply (*rule-tac x* = *abs c* **in** *exI*)
 apply *auto*
 apply (*subgoal-tac c* * *abs*(*f x*) <= *abs c* * *abs* (*f x*))
 apply (*erule-tac x* = *x* **in** *allE*)
 apply *force*
 apply (*rule mult-right-mono*)
 apply (*rule abs-ge-self*)
 apply (*rule abs-ge-zero*)
done

lemma *bigo-alt-def*: $O(f) =$
$\{h.\ EX\ c.\ (0 < c\ \&\ (ALL\ x.\ abs\ (h\ x) <= c * abs\ (f\ x)))\}$
by (*auto simp add*: *bigo-def bigo-pos-const*)

lemma *bigo-elt-subset* [*intro*]: $f : O(g) ==> O(f) <= O(g)$
apply (*auto simp add*: *bigo-alt-def*)
apply (*rule-tac* $x = ca * c$ **in** *exI*)
apply (*rule conjI*)
apply (*rule mult-pos-pos*)
apply (*assumption*)+
apply (*rule allI*)
apply (*drule-tac* $x = xa$ **in** *spec*)+
apply (*subgoal-tac* $ca * abs(f\ xa) <= ca * (c * abs(g\ xa))$)
apply (*erule order-trans*)
apply (*simp add*: *mult-ac*)
apply (*rule mult-left-mono*, *assumption*)
apply (*rule order-less-imp-le*, *assumption*)
done

lemma *bigo-refl* [*intro*]: $f : O(f)$
apply(*auto simp add*: *bigo-def*)
apply(*rule-tac* $x = 1$ **in** *exI*)
apply *simp*
done

lemma *bigo-zero*: $0 : O(g)$
apply (*auto simp add*: *bigo-def func-zero*)
apply (*rule-tac* $x = 0$ **in** *exI*)
apply *auto*
done

lemma *bigo-zero2*: $O(\%x.0) = \{\%x.0\}$
apply (*auto simp add*: *bigo-def*)
apply (*rule ext*)
apply *auto*
done

lemma *bigo-plus-self-subset* [*intro*]:
$O(f) + O(f) <= O(f)$
apply (*auto simp add*: *bigo-alt-def set-plus*)
apply (*rule-tac* $x = c + ca$ **in** *exI*)
apply *auto*
apply (*simp add*: *ring-distrib func-plus*)
apply (*rule order-trans*)
apply (*rule abs-triangle-ineq*)
apply (*rule add-mono*)
apply *force*
apply *force*

done

lemma *bigo-plus-idemp* [*simp*]: $O(f) + O(f) = O(f)$
apply (*rule equalityI*)
apply (*rule bigo-plus-self-subset*)
apply (*rule set-zero-plus2*)
apply (*rule bigo-zero*)
done

lemma *bigo-plus-subset* [*intro*]: $O(f + g) <= O(f) + O(g)$
apply (*rule subsetI*)
apply (*auto simp add*: *bigo-def bigo-pos-const func-plus set-plus*)
apply (*subst bigo-pos-const* [*symmetric*])+
apply (*rule-tac x* =
$\%n.$ *if abs* (*g n*) <= (*abs* (*f n*)) *then x n else 0* **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac x* = *c* + *c* **in** *exI*)
apply (*clarsimp*)
apply (*auto*)
apply (*subgoal-tac c* * *abs* (*f xa* + *g xa*) <= (*c* + *c*) * *abs* (*f xa*))
apply (*erule-tac x* = *xa* **in** *allE*)
apply (*erule order-trans*)
apply (*simp*)
apply (*subgoal-tac c* * *abs* (*f xa* + *g xa*) <= *c* * (*abs* (*f xa*) + *abs* (*g xa*)))
apply (*erule order-trans*)
apply (*simp add*: *ring-distrib*)
apply (*rule mult-left-mono*)
apply *assumption*
apply (*simp add*: *order-less-le*)
apply (*rule mult-left-mono*)
apply (*simp add*: *abs-triangle-ineq*)
apply (*simp add*: *order-less-le*)
apply (*rule mult-nonneg-nonneg*)
apply (*rule add-nonneg-nonneg*)
apply *auto*
apply (*rule-tac x* = $\%n.$ *if* (*abs* (*f n*)) < *abs* (*g n*) *then x n else 0*
in *exI*)
apply (*rule conjI*)
apply (*rule-tac x* = *c* + *c* **in** *exI*)
apply *auto*
apply (*subgoal-tac c* * *abs* (*f xa* + *g xa*) <= (*c* + *c*) * *abs* (*g xa*))
apply (*erule-tac x* = *xa* **in** *allE*)
apply (*erule order-trans*)
apply (*simp*)
apply (*subgoal-tac c* * *abs* (*f xa* + *g xa*) <= *c* * (*abs* (*f xa*) + *abs* (*g xa*)))
apply (*erule order-trans*)
apply (*simp add*: *ring-distrib*)
apply (*rule mult-left-mono*)
apply (*simp add*: *order-less-le*)

apply (*simp add*: *order-less-le*)
apply (*rule mult-left-mono*)
apply (*rule abs-triangle-ineq*)
apply (*simp add*: *order-less-le*)
apply (*rule mult-nonneg-nonneg*)
apply (*rule add-nonneg-nonneg*)
apply (*erule order-less-imp-le*)+
apply *simp*
apply (*rule ext*)
apply (*auto simp add*: *if-splits linorder-not-le*)
done

lemma *bigo-plus-subset2* [*intro*]: $A <= O(f) ==> B <= O(f) ==> A + B <= O(f)$
apply (*subgoal-tac* $A + B <= O(f) + O(f)$)
apply (*erule order-trans*)
apply *simp*
apply (*auto del*: *subsetI simp del*: *bigo-plus-idemp*)
done

lemma *bigo-plus-eq*: *ALL x. 0 <= f x ==> ALL x. 0 <= g x ==>*
$O(f + g) = O(f) + O(g)$
apply (*rule equalityI*)
apply (*rule bigo-plus-subset*)
apply (*simp add*: *bigo-alt-def set-plus func-plus*)
apply *clarify*
apply (*rule-tac x = max c ca* **in** *exI*)
apply (*rule conjI*)
apply (*subgoal-tac c <= max c ca*)
apply (*erule order-less-le-trans*)
apply *assumption*
apply (*rule le-maxI1*)
apply *clarify*
apply (*drule-tac x = xa* **in** *spec*)+
apply (*subgoal-tac 0 <= f xa + g xa*)
apply (*simp add*: *ring-distrib*)
apply (*subgoal-tac abs(a xa + b xa) <= abs(a xa) + abs(b xa)*)
apply (*subgoal-tac abs(a xa) + abs(b xa) <=*
*max c ca * f xa + max c ca * g xa*)
apply (*force*)
apply (*rule add-mono*)
apply (*subgoal-tac c * f xa <= max c ca * f xa*)
apply (*force*)
apply (*rule mult-right-mono*)
apply (*rule le-maxI1*)
apply *assumption*
apply (*subgoal-tac ca * g xa <= max c ca * g xa*)
apply (*force*)
apply (*rule mult-right-mono*)

apply (*rule le-maxI2*)
apply *assumption*
apply (*rule abs-triangle-ineq*)
apply (*rule add-nonneg-nonneg*)
apply *assumption*+
done

lemma *bigo-bounded-alt*: *ALL x. 0 <= f x ==> ALL x. f x <= c * g x ==>*
f : O(g)
apply (*auto simp add*: *bigo-def*)
apply (*rule-tac x = abs c* **in** *exI*)
apply *auto*
apply (*drule-tac x = x* **in** *spec*)+
apply (*simp add*: *abs-mult* [*symmetric*])
done

lemma *bigo-bounded*: *ALL x. 0 <= f x ==> ALL x. f x <= g x ==>*
f : O(g)
apply (*erule bigo-bounded-alt* [*of f 1 g*])
apply *simp*
done

lemma *bigo-bounded2*: *ALL x. lb x <= f x ==> ALL x. f x <= lb x + g x ==>*
f : lb +o O(g)
apply (*rule set-minus-imp-plus*)
apply (*rule bigo-bounded*)
apply (*auto simp add*: *diff-minus func-minus func-plus*)
apply (*drule-tac x = x* **in** *spec*)+
apply *force*
apply (*drule-tac x = x* **in** *spec*)+
apply *force*
done

lemma *bigo-abs*: *(%x. abs(f x)) =o O(f)*
apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac x = 1* **in** *exI*)
apply *auto*
done

lemma *bigo-abs2*: *f =o O(%x. abs(f x))*
apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac x = 1* **in** *exI*)
apply *auto*
done

lemma *bigo-abs3*: *O(f) = O(%x. abs(f x))*
apply (*rule equalityI*)

```
  apply (rule bigo-elt-subset)
  apply (rule bigo-abs2)
  apply (rule bigo-elt-subset)
  apply (rule bigo-abs)
done

lemma bigo-abs4: f =o g +o O(h) ==>
    (%x. abs (f x)) =o (%x. abs (g x)) +o O(h)
  apply (drule set-plus-imp-minus)
  apply (rule set-minus-imp-plus)
  apply (subst func-diff)
proof -
  assume a: f - g : O(h)
  have (%x. abs (f x) - abs (g x)) =o O(%x. abs(abs (f x) - abs (g x)))
    by (rule bigo-abs2)
  also have ... <= O(%x. abs (f x - g x))
    apply (rule bigo-elt-subset)
    apply (rule bigo-bounded)
    apply force
    apply (rule allI)
    apply (rule abs-triangle-ineq3)
    done
  also have ... <= O(f - g)
    apply (rule bigo-elt-subset)
    apply (subst func-diff)
    apply (rule bigo-abs)
    done
  also have ... <= O(h)
    by (rule bigo-elt-subset)
  finally show (%x. abs (f x) - abs (g x)) : O(h).
qed

lemma bigo-abs5: f =o O(g) ==> (%x. abs(f x)) =o O(g)
by (unfold bigo-def, auto)

lemma bigo-elt-subset2 [intro]: f : g +o O(h) ==> O(f) <= O(g) + O(h)
proof -
  assume f : g +o O(h)
  also have ... <= O(g) + O(h)
    by (auto del: subsetI)
  also have ... = O(%x. abs(g x)) + O(%x. abs(h x))
    apply (subst bigo-abs3 [symmetric])+
    apply (rule refl)
    done
  also have ... = O((%x. abs(g x)) + (%x. abs(h x)))
    by (rule bigo-plus-eq [symmetric], auto)
  finally have f : ....
  then have O(f) <= ...
    by (elim bigo-elt-subset)
```

also have ... = $O(\%x.\ abs(g\ x)) + O(\%x.\ abs(h\ x))$
by (*rule bigo-plus-eq*, *auto*)
finally show *?thesis*
by (*simp add*: *bigo-abs3* [*symmetric*])
qed

lemma *bigo-mult* [*intro*]: $O(f)*O(g) <= O(f * g)$
apply (*rule subsetI*)
apply (*subst bigo-def*)
apply (*auto simp add*: *bigo-alt-def set-times func-times*)
apply (*rule-tac* $x = c * ca$ **in** *exI*)
apply(*rule allI*)
apply(*erule-tac* $x = x$ **in** *allE*)+
apply(*subgoal-tac* $c * ca * abs(f\ x * g\ x) =$
$(c * abs(f\ x)) * (ca * abs(g\ x)))$
apply(*erule ssubst*)
apply (*subst abs-mult*)
apply (*rule mult-mono*)
apply *assumption*+
apply (*rule mult-nonneg-nonneg*)
apply *auto*
apply (*simp add*: *mult-ac abs-mult*)
done

lemma *bigo-mult2* [*intro*]: $f *o O(g) <= O(f * g)$
apply (*auto simp add*: *bigo-def elt-set-times-def func-times abs-mult*)
apply (*rule-tac* $x = c$ **in** *exI*)
apply *auto*
apply (*drule-tac* $x = x$ **in** *spec*)
apply (*subgoal-tac* $abs(f\ x) * abs(b\ x) <= abs(f\ x) * (c * abs(g\ x)))$
apply (*force simp add*: *mult-ac*)
apply (*rule mult-left-mono*, *assumption*)
apply (*rule abs-ge-zero*)
done

lemma *bigo-mult3*: $f : O(h) ==> g : O(j) ==> f * g : O(h * j)$
apply (*rule subsetD*)
apply (*rule bigo-mult*)
apply (*erule set-times-intro*, *assumption*)
done

lemma *bigo-mult4* [*intro*]:$f : k +o O(h) ==> g * f : (g * k) +o O(g * h)$
apply (*drule set-plus-imp-minus*)
apply (*rule set-minus-imp-plus*)
apply (*drule bigo-mult3* [**where** $g = g$ **and** $j = g$])
apply (*auto simp add*: *ring-eq-simps mult-ac*)
done

lemma *bigo-mult5*: *ALL x*. $f\ x \sim= 0 ==>$

```
    O(f * g) <= (f::'a => ('b::ordered-field)) *o O(g)
proof -
  assume ALL x. f x ~= 0
  show O(f * g) <= f *o O(g)
  proof
    fix h
    assume h : O(f * g)
    then have (%x. 1 / (f x)) * h : (%x. 1 / f x) *o O(f * g)
      by auto
    also have ... <= O((%x. 1 / f x) * (f * g))
      by (rule bigo-mult2)
    also have (%x. 1 / f x) * (f * g) = g
      apply (simp add: func-times)
      apply (rule ext)
      apply (simp add: prems nonzero-divide-eq-eq mult-ac)
      done
    finally have (%x. (1::'b) / f x) * h : O(g).
    then have f * ((%x. (1::'b) / f x) * h) : f *o O(g)
      by auto
    also have f * ((%x. (1::'b) / f x) * h) = h
      apply (simp add: func-times)
      apply (rule ext)
      apply (simp add: prems nonzero-divide-eq-eq mult-ac)
      done
    finally show h : f *o O(g).
  qed
qed

lemma bigo-mult6: ALL x. f x ~= 0 ==>
    O(f * g) = (f::'a => ('b::ordered-field)) *o O(g)
  apply (rule equalityI)
  apply (erule bigo-mult5)
  apply (rule bigo-mult2)
done

lemma bigo-mult7: ALL x. f x ~= 0 ==>
    O(f * g) <= O(f::'a => ('b::ordered-field)) * O(g)
  apply (subst bigo-mult6)
  apply assumption
  apply (rule set-times-mono3)
  apply (rule bigo-refl)
done

lemma bigo-mult8: ALL x. f x ~= 0 ==>
    O(f * g) = O(f::'a => ('b::ordered-field)) * O(g)
  apply (rule equalityI)
  apply (erule bigo-mult7)
  apply (rule bigo-mult)
done
```

lemma *bigo-minus* [*intro*]: $f : O(g) ==> -f : O(g)$
by (*auto simp add*: *bigo-def func-minus*)

lemma *bigo-minus2*: $f : g +o O(h) ==> -f : -g +o O(h)$
apply (*rule set-minus-imp-plus*)
apply (*drule set-plus-imp-minus*)
apply (*drule bigo-minus*)
apply (*simp add*: *diff-minus*)
done

lemma *bigo-minus3*: $O(-f) = O(f)$
by (*auto simp add*: *bigo-def func-minus abs-minus-cancel*)

lemma *bigo-plus-absorb-lemma1*: $f : O(g) ==> f +o O(g) <= O(g)$
proof −
assume *a*: $f : O(g)$
show $f +o O(g) <= O(g)$
proof −
have $f : O(f)$ **by** *auto*
then have $f +o O(g) <= O(f) + O(g)$
by (*auto del*: *subsetI*)
also have $... <= O(g) + O(g)$
proof −
from *a* **have** $O(f) <= O(g)$ **by** (*auto del*: *subsetI*)
thus *?thesis* **by** (*auto del*: *subsetI*)
qed
also have $... <= O(g)$ **by** (*simp add*: *bigo-plus-idemp*)
finally show *?thesis* **.**
qed
qed

lemma *bigo-plus-absorb-lemma2*: $f : O(g) ==> O(g) <= f +o O(g)$
proof −
assume *a*: $f : O(g)$
show $O(g) <= f +o O(g)$
proof −
from *a* **have** $-f : O(g)$ **by** *auto*
then have $-f +o O(g) <= O(g)$ **by** (*elim bigo-plus-absorb-lemma1*)
then have $f +o (-f +o O(g)) <= f +o O(g)$ **by** *auto*
also have $f +o (-f +o O(g)) = O(g)$
by (*simp add*: *set-plus-rearranges*)
finally show *?thesis* **.**
qed
qed

lemma *bigo-plus-absorb* [*simp*]: $f : O(g) ==> f +o O(g) = O(g)$
apply (*rule equalityI*)
apply (*erule bigo-plus-absorb-lemma1*)

apply (*erule bigo-plus-absorb-lemma2*)
done

lemma *bigo-plus-absorb2* [*intro*]: $f : O(g) ==> A <= O(g) ==> f +o A <= O(g)$
apply (*subgoal-tac* $f +o A <= f +o O(g)$)
apply *force+*
done

lemma *bigo-add-commute-imp*: $f : g +o O(h) ==> g : f +o O(h)$
apply (*subst set-minus-plus* [*symmetric*])
apply (*subgoal-tac* $g - f = - (f - g)$)
apply (*erule ssubst*)
apply (*rule bigo-minus*)
apply (*subst set-minus-plus*)
apply *assumption*
apply (*simp add*: *diff-minus add-ac*)
done

lemma *bigo-add-commute*: $(f : g +o O(h)) = (g : f +o O(h))$
apply (*rule iffI*)
apply (*erule bigo-add-commute-imp*)+
done

lemma *bigo-const1*: $(\%x.\ c) : O(\%x.\ 1)$
by (*auto simp add*: *bigo-def mult-ac*)

lemma *bigo-const2* [*intro*]: $O(\%x.\ c) <= O(\%x.\ 1)$
apply (*rule bigo-elt-subset*)
apply (*rule bigo-const1*)
done

lemma *bigo-const3*: $(c::'a::ordered\text{-}field) \sim= 0 ==> (\%x.\ 1) : O(\%x.\ c)$
apply (*simp add*: *bigo-def*)
apply (*rule-tac* $x = abs(inverse\ c)$ **in** *exI*)
apply (*simp add*: *abs-mult* [*symmetric*])
done

lemma *bigo-const4*: $(c::'a::ordered\text{-}field) \sim= 0 ==> O(\%x.\ 1) <= O(\%x.\ c)$
by (*rule bigo-elt-subset*, *rule bigo-const3*, *assumption*)

lemma *bigo-const* [*simp*]: $(c::'a::ordered\text{-}field) \sim= 0 ==>$
$O(\%x.\ c) = O(\%x.\ 1)$
by (*rule equalityI*, *rule bigo-const2*, *rule bigo-const4*, *assumption*)

lemma *bigo-const-mult1*: $(\%x.\ c * f\ x) : O(f)$
apply (*simp add*: *bigo-def*)
apply (*rule-tac* $x = abs(c)$ **in** *exI*)
apply (*auto simp add*: *abs-mult* [*symmetric*])

done

lemma *bigo-const-mult2*: $O(\%x.\ c * f\ x) <= O(f)$
by (*rule bigo-elt-subset*, *rule bigo-const-mult1*)

lemma *bigo-const-mult3*: $(c::'a::ordered\text{-}field)\ \sim= 0 ==> f : O(\%x.\ c * f\ x)$
apply (*simp add*: *bigo-def*)
apply (*rule-tac* $x = abs(inverse\ c)$ **in** *exI*)
apply (*simp add*: *abs-mult* [*symmetric*] *mult-assoc* [*symmetric*])
done

lemma *bigo-const-mult4*: $(c::'a::ordered\text{-}field)\ \sim= 0 ==>$
$O(f) <= O(\%x.\ c * f\ x)$
by (*rule bigo-elt-subset*, *rule bigo-const-mult3*, *assumption*)

lemma *bigo-const-mult* [*simp*]: $(c::'a::ordered\text{-}field)\ \sim= 0 ==>$
$O(\%x.\ c * f\ x) = O(f)$
by (*rule equalityI*, *rule bigo-const-mult2*, *erule bigo-const-mult4*)

lemma *bigo-const-mult5* [*simp*]: $(c::'a::ordered\text{-}field)\ \sim= 0 ==>$
$(\%x.\ c) *o O(f) = O(f)$
apply (*auto del*: *subsetI*)
apply (*rule order-trans*)
apply (*rule bigo-mult2*)
apply (*simp add*: *func-times*)
apply (*auto intro*!: *subsetI simp add*: *bigo-def elt-set-times-def func-times*)
apply (*rule-tac* $x = \%y.\ inverse\ c * x\ y$ **in** *exI*)
apply (*simp add*: *mult-assoc* [*symmetric*] *abs-mult*)
apply (*rule-tac* $x = abs\ (inverse\ c) * ca$ **in** *exI*)
apply (*rule allI*)
apply (*subst mult-assoc*)
apply (*rule mult-left-mono*)
apply (*erule spec*)
apply *force*
done

lemma *bigo-const-mult6* [*intro*]: $(\%x.\ c) *o O(f) <= O(f)$
apply (*auto intro*!: *subsetI*
simp add: *bigo-def elt-set-times-def func-times*)
apply (*rule-tac* $x = ca * (abs\ c)$ **in** *exI*)
apply (*rule allI*)
apply (*subgoal-tac* $ca * abs(c) * abs(f\ x) = abs(c) * (ca * abs(f\ x))$)
apply (*erule ssubst*)
apply (*subst abs-mult*)
apply (*rule mult-left-mono*)
apply (*erule spec*)
apply *simp*
apply(*simp add*: *mult-ac*)
done

lemma *bigo-const-mult7* [*intro*]: $f =o O(g) ==> (\%x.\ c * f\ x) =o O(g)$
proof −
assume $f =o O(g)$
then have $(\%x.\ c) * f =o (\%x.\ c) *o O(g)$
by *auto*
also have $(\%x.\ c) * f = (\%x.\ c * f\ x)$
by (*simp add*: *func-times*)
also have $(\%x.\ c) *o O(g) <= O(g)$
by (*auto del*: *subsetI*)
finally show *?thesis* **.**
qed

lemma *bigo-compose1*: $f =o O(g) ==> (\%x.\ f(k\ x)) =o O(\%x.\ g(k\ x))$
by (*unfold bigo-def*, *auto*)

lemma *bigo-compose2*: $f =o g +o O(h) ==> (\%x.\ f(k\ x)) =o (\%x.\ g(k\ x)) +o O(\%x.\ h(k\ x))$
apply (*simp only*: *set-minus-plus* [*symmetric*] *diff-minus func-minus func-plus*)
apply (*erule bigo-compose1*)
done

3.2 Setsum

lemma *bigo-setsum-main*: $ALL\ x.\ ALL\ y : A\ x.\ 0 <= h\ x\ y ==> EX\ c.\ ALL\ x.\ ALL\ y : A\ x.\ abs(f\ x\ y) <= c * (h\ x\ y) ==> (\%x.\ SUM\ y : A\ x.\ f\ x\ y) =o O(\%x.\ SUM\ y : A\ x.\ h\ x\ y)$
apply (*auto simp add*: *bigo-def*)
apply (*rule-tac* $x = abs\ c$ **in** *exI*)
apply (*subst abs-of-nonneg*) **back back**
apply (*rule setsum-nonneg*)
apply *force*
apply (*subst setsum-mult*)
apply (*rule allI*)
apply (*rule order-trans*)
apply (*rule setsum-abs*)
apply (*rule setsum-mono*)
apply (*rule order-trans*)
apply (*drule spec*)+
apply (*drule bspec*)+
apply *assumption*+
apply (*drule bspec*)
apply *assumption*+
apply (*rule mult-right-mono*)
apply (*rule abs-ge-self*)
apply *force*
done

lemma *bigo-setsum1*: *ALL x y. 0 <= h x y ==>*
*EX c. ALL x y. abs(f x y) <= c * (h x y) ==>*
(%x. SUM y : A x. f x y) =o O(%x. SUM y : A x. h x y)
apply (*rule bigo-setsum-main*)
apply *force*
apply *clarsimp*
apply (*rule-tac x = c* **in** *exI*)
apply *force*
done

lemma *bigo-setsum2*: *ALL y. 0 <= h y ==>*
*EX c. ALL y. abs(f y) <= c * (h y) ==>*
(%x. SUM y : A x. f y) =o O(%x. SUM y : A x. h y)
by (*rule bigo-setsum1*, *auto*)

lemma *bigo-setsum3*: *f =o O(h) ==>*
*(%x. SUM y : A x. (l x y) * f(k x y)) =o*
*O(%x. SUM y : A x. abs(l x y * h(k x y)))*
apply (*rule bigo-setsum1*)
apply (*rule allI*)+
apply (*rule abs-ge-zero*)
apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac x = c* **in** *exI*)
apply (*rule allI*)+
apply (*subst abs-mult*)+
apply (*subst mult-left-commute*)
apply (*rule mult-left-mono*)
apply (*erule spec*)
apply (*rule abs-ge-zero*)
done

lemma *bigo-setsum4*: *f =o g +o O(h) ==>*
*(%x. SUM y : A x. l x y * f(k x y)) =o*
*(%x. SUM y : A x. l x y * g(k x y)) +o*
*O(%x. SUM y : A x. abs(l x y * h(k x y)))*
apply (*rule set-minus-imp-plus*)
apply (*subst func-diff*)
apply (*subst setsum-subtractf* [*symmetric*])
apply (*subst right-diff-distrib* [*symmetric*])
apply (*rule bigo-setsum3*)
apply (*subst func-diff* [*symmetric*])
apply (*erule set-plus-imp-minus*)
done

lemma *bigo-setsum5*: *f =o O(h) ==> ALL x y. 0 <= l x y ==>*
ALL x. 0 <= h x ==>
*(%x. SUM y : A x. (l x y) * f(k x y)) =o*
*O(%x. SUM y : A x. (l x y) * h(k x y))*

apply (*subgoal-tac* (%*x*. *SUM y* : *A x*. (*l x y*) ∗ *h*(*k x y*)) =
(%*x*. *SUM y* : *A x*. *abs*((*l x y*) ∗ *h*(*k x y*))))
apply (*erule ssubst*)
apply (*erule bigo-setsum3*)
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst abs-of-nonneg*)
apply (*rule mult-nonneg-nonneg*)
apply *auto*
done

lemma *bigo-setsum6*: *f* =*o* *g* +*o* *O*(*h*) ==> *ALL x y*. 0 <= *l x y* ==>
ALL x. 0 <= *h x* ==>
(%*x*. *SUM y* : *A x*. (*l x y*) ∗ *f*(*k x y*)) =*o*
(%*x*. *SUM y* : *A x*. (*l x y*) ∗ *g*(*k x y*)) +*o*
O(%*x*. *SUM y* : *A x*. (*l x y*) ∗ *h*(*k x y*))
apply (*rule set-minus-imp-plus*)
apply (*subst func-diff*)
apply (*subst setsum-subtractf* [*symmetric*])
apply (*subst right-diff-distrib* [*symmetric*])
apply (*rule bigo-setsum5*)
apply (*subst func-diff* [*symmetric*])
apply (*drule set-plus-imp-minus*)
apply *auto*
done

3.3 Misc useful stuff

lemma *bigo-useful-intro*: *A* <= *O*(*f*) ==> *B* <= *O*(*f*) ==>
A + *B* <= *O*(*f*)
apply (*subst bigo-plus-idemp* [*symmetric*])
apply (*rule set-plus-mono2*)
apply *assumption*+
done

lemma *bigo-useful-add*: *f* =*o* *O*(*h*) ==> *g* =*o* *O*(*h*) ==> *f* + *g* =*o* *O*(*h*)
apply (*subst bigo-plus-idemp* [*symmetric*])
apply (*rule set-plus-intro*)
apply *assumption*+
done

lemma *bigo-useful-const-mult*: (*c*::'*a*::*ordered-field*) ~= 0 ==>
(%*x*. *c*) ∗ *f* =*o* *O*(*h*) ==> *f* =*o* *O*(*h*)
apply (*rule subsetD*)
apply (*subgoal-tac* (%*x*. 1 / *c*) ∗*o* *O*(*h*) <= *O*(*h*))
apply *assumption*
apply (*rule bigo-const-mult6*)
apply (*subgoal-tac f* = (%*x*. 1 / *c*) ∗ ((%*x*. *c*) ∗ *f*))
apply (*erule ssubst*)

apply (*erule set-times-intro2*)
apply (*simp add*: *func-times*)
apply (*rule ext*)
apply (*subst times-divide-eq-left* [*symmetric*])
apply (*subst divide-self*)
apply (*assumption*, *simp*)
done

lemma *bigo-fix*: (%*x*. *f* ((*x*::*nat*) + *1*)) =*o* *O*(%*x*. *h*(*x* + *1*)) ==> *f* *0* = *0* ==>
f =*o* *O*(*h*)
apply (*simp add*: *bigo-alt-def*)
apply *auto*
apply (*rule-tac x* = *c* **in** *exI*)
apply *auto*
apply (*case-tac x* = *0*)
apply *simp*
apply (*rule mult-nonneg-nonneg*)
apply *force*
apply *force*
apply (*subgoal-tac x* = *Suc* (*x* − *1*))
apply (*erule ssubst*) **back**
apply (*erule spec*)
apply *simp*
done

lemma *bigo-fix2*:
(%*x*. *f* ((*x*::*nat*) + *1*)) =*o* (%*x*. *g*(*x* + *1*)) +*o* *O*(%*x*. *h*(*x* + *1*)) ==>
f *0* = *g* *0* ==> *f* =*o* *g* +*o* *O*(*h*)
apply (*rule set-minus-imp-plus*)
apply (*rule bigo-fix*)
apply (*subst func-diff*)
apply (*subst func-diff* [*symmetric*])
apply (*rule set-plus-imp-minus*)
apply *simp*
apply (*simp add*: *func-diff*)
done

3.4 Less than or equal to

constdefs
lesso :: ('*a* => '*b*::*ordered-idom*) => ('*a* => '*b*) => ('*a* => '*b*)
(**infixl** <*o* *70*)
f <*o* *g* == (%*x*. *max* (*f* *x* − *g* *x*) *0*)

lemma *bigo-lesseq1*: *f* =*o* *O*(*h*) ==> *ALL* *x*. *abs* (*g* *x*) <= *abs* (*f* *x*) ==>
g =*o* *O*(*h*)
apply (*unfold bigo-def*)
apply *clarsimp*
apply (*rule-tac x* = *c* **in** *exI*)

apply (*rule allI*)
apply (*rule order-trans*)
apply (*erule spec*)+
done

lemma *bigo-lesseq2*: *f =o O(h) ==> ALL x. abs (g x) <= f x ==>*
g =o O(h)
apply (*erule bigo-lesseq1*)
apply (*rule allI*)
apply (*drule-tac x = x* **in** *spec*)
apply (*rule order-trans*)
apply *assumption*
apply (*rule abs-ge-self*)
done

lemma *bigo-lesseq3*: *f =o O(h) ==> ALL x. 0 <= g x ==> ALL x. g x <= f x ==>*
g =o O(h)
apply (*erule bigo-lesseq2*)
apply (*rule allI*)
apply (*subst abs-of-nonneg*)
apply (*erule spec*)+
done

lemma *bigo-lesseq4*: *f =o O(h) ==>*
ALL x. 0 <= g x ==> ALL x. g x <= abs (f x) ==>
g =o O(h)
apply (*erule bigo-lesseq1*)
apply (*rule allI*)
apply (*subst abs-of-nonneg*)
apply (*erule spec*)+
done

lemma *bigo-lesso1*: *ALL x. f x <= g x ==> f <o g =o O(h)*
apply (*unfold lesso-def*)
apply (*subgoal-tac* (%*x. max* (*f x − g x*) *0*) *= 0*)
apply (*erule ssubst*)
apply (*rule bigo-zero*)
apply (*unfold func-zero*)
apply (*rule ext*)
apply (*simp split*: *split-max*)
done

lemma *bigo-lesso2*: *f =o g +o O(h) ==>*
ALL x. 0 <= k x ==> ALL x. k x <= f x ==>
k <o g =o O(h)
apply (*unfold lesso-def*)
apply (*rule bigo-lesseq4*)
apply (*erule set-plus-imp-minus*)

apply (*rule allI*)
apply (*rule le-maxI2*)
apply (*rule allI*)
apply (*subst func-diff*)
apply (*case-tac 0 <= k x − g x*)
apply *simp*
apply (*subst abs-of-nonneg*)
apply (*drule-tac x = x* **in** *spec*) **back**
apply (*simp add*: *compare-rls*)
apply (*subst diff-minus*)+
apply (*rule add-right-mono*)
apply (*erule spec*)
apply (*rule order-trans*)
prefer *2*
apply (*rule abs-ge-zero*)
apply (*simp add*: *compare-rls*)
done

lemma *bigo-lesso3*: *f =o g +o O(h) ==>*
ALL x. 0 <= k x ==> ALL x. g x <= k x ==>
f <o k =o O(h)
apply (*unfold lesso-def*)
apply (*rule bigo-lesseq4*)
apply (*erule set-plus-imp-minus*)
apply (*rule allI*)
apply (*rule le-maxI2*)
apply (*rule allI*)
apply (*subst func-diff*)
apply (*case-tac 0 <= f x − k x*)
apply *simp*
apply (*subst abs-of-nonneg*)
apply (*drule-tac x = x* **in** *spec*) **back**
apply (*simp add*: *compare-rls*)
apply (*subst diff-minus*)+
apply (*rule add-left-mono*)
apply (*rule le-imp-neg-le*)
apply (*erule spec*)
apply (*rule order-trans*)
prefer *2*
apply (*rule abs-ge-zero*)
apply (*simp add*: *compare-rls*)
done

lemma *bigo-lesso4*: *f <o g =o O(k::'a=>'b::ordered-field) ==>*
g =o h +o O(k) ==> f <o h =o O(k)
apply (*unfold lesso-def*)
apply (*drule set-plus-imp-minus*)
apply (*drule bigo-abs5*) **back**
apply (*simp add*: *func-diff*)

apply (*drule bigo-useful-add*)
apply *assumption*
apply (*erule bigo-lesseq2*) **back**
apply (*rule allI*)
apply (*auto simp add*: *func-plus func-diff compare-rls*
split: *split-max abs-split*)
done

lemma *bigo-lesso5*: $f <o g =o O(h) ==>$
$EX\ C.\ ALL\ x.\ f\ x <= g\ x + C * abs(h\ x)$
apply (*simp only*: *lesso-def bigo-alt-def*)
apply *clarsimp*
apply (*rule-tac* $x = c$ **in** *exI*)
apply (*rule allI*)
apply (*drule-tac* $x = x$ **in** *spec*)
apply (*subgoal-tac* $abs(max\ (f\ x - g\ x)\ 0) = max\ (f\ x - g\ x)\ 0$)
apply (*clarsimp simp add*: *compare-rls add-ac*)
apply (*rule abs-of-nonneg*)
apply (*rule le-maxI2*)
done

lemma *lesso-add*: $f <o g =o O(h) ==>$
$k <o l =o O(h) ==> (f + k) <o (g + l) =o O(h)$
apply (*unfold lesso-def*)
apply (*rule bigo-lesseq3*)
apply (*erule bigo-useful-add*)
apply *assumption*
apply (*force split*: *split-max*)
apply (*auto split*: *split-max simp add*: *func-plus*)
done

end

4 Continuity: Continuity and iterations (of set transformers)

theory *Continuity*
imports *Main*
begin

4.1 Chains

constdefs
up-chain :: $(nat => 'a\ set) => bool$
up-chain $F == \forall i.\ F\ i \subseteq F\ (Suc\ i)$

lemma *up-chainI*: $(!!i.\ F\ i \subseteq F\ (Suc\ i)) ==>$ *up-chain* F

by (*simp add*: *up-chain-def*)

lemma *up-chainD*: *up-chain F* ==> *F i* ⊆ *F* (*Suc i*)
by (*simp add*: *up-chain-def*)

lemma *up-chain-less-mono* [*rule-format*]:
up-chain F ==> *x* < *y* --> *F x* ⊆ *F y*
apply (*induct-tac y*)
apply (*blast dest*: *up-chainD elim*: *less-SucE*)+
done

lemma *up-chain-mono*: *up-chain F* ==> *x* ≤ *y* ==> *F x* ⊆ *F y*
apply (*drule le-imp-less-or-eq*)
apply (*blast dest*: *up-chain-less-mono*)
done

constdefs
down-chain :: (*nat* => *'a set*) => *bool*
down-chain F == ∀ *i*. *F* (*Suc i*) ⊆ *F i*

lemma *down-chainI*: (!!*i*. *F* (*Suc i*) ⊆ *F i*) ==> *down-chain F*
by (*simp add*: *down-chain-def*)

lemma *down-chainD*: *down-chain F* ==> *F* (*Suc i*) ⊆ *F i*
by (*simp add*: *down-chain-def*)

lemma *down-chain-less-mono* [*rule-format*]:
down-chain F ==> *x* < *y* --> *F y* ⊆ *F x*
apply (*induct-tac y*)
apply (*blast dest*: *down-chainD elim*: *less-SucE*)+
done

lemma *down-chain-mono*: *down-chain F* ==> *x* ≤ *y* ==> *F y* ⊆ *F x*
apply (*drule le-imp-less-or-eq*)
apply (*blast dest*: *down-chain-less-mono*)
done

4.2 Continuity

constdefs
up-cont :: (*'a set* => *'a set*) => *bool*
up-cont f == ∀ *F*. *up-chain F* --> *f* (⋃(*range F*)) = ⋃(*f* ' *range F*)

lemma *up-contI*:
(!!*F*. *up-chain F* ==> *f* (⋃(*range F*)) = ⋃(*f* ' *range F*)) ==> *up-cont f*
apply (*unfold up-cont-def*)
apply *blast*
done

lemma *up-contD*:
up-cont f ==> up-chain F ==> f $(\bigcup$ *(range F)) =* $\bigcup$ *(f ' range F)*
apply (*unfold up-cont-def*)
apply *auto*
done

lemma *up-cont-mono*: *up-cont f ==> mono f*
apply (*rule monoI*)
apply (*drule-tac F =* λ*i. if i = 0 then A else B* **in** *up-contD*)
apply (*rule up-chainI*)
apply *simp+*
apply (*drule Un-absorb1*)
apply (*auto simp add*: *nat-not-singleton*)
done

constdefs
down-cont :: (*'a set => 'a set*) *=> bool*
down-cont f ==
$\forall$ *F. down-chain F --> f (Inter (range F)) = Inter (f ' range F)*

lemma *down-contI*:
(!!*F. down-chain F ==> f (Inter (range F)) = Inter (f ' range F)) ==>*
down-cont f
apply (*unfold down-cont-def*)
apply *blast*
done

lemma *down-contD*: *down-cont f ==> down-chain F ==>*
f (Inter (range F)) = Inter (f ' range F)
apply (*unfold down-cont-def*)
apply *auto*
done

lemma *down-cont-mono*: *down-cont f ==> mono f*
apply (*rule monoI*)
apply (*drule-tac F =* λ*i. if i = 0 then B else A* **in** *down-contD*)
apply (*rule down-chainI*)
apply *simp+*
apply (*drule Int-absorb1*)
apply (*auto simp add*: *nat-not-singleton*)
done

4.3 Iteration

constdefs
up-iterate :: (*'a set => 'a set*) *=> nat => 'a set*

up-iterate f n == (f^n) {}

lemma *up-iterate-0* [*simp*]: *up-iterate f 0 = {}*
by (*simp add*: *up-iterate-def*)

lemma *up-iterate-Suc* [*simp*]: *up-iterate f (Suc i) = f (up-iterate f i)*
by (*simp add*: *up-iterate-def*)

lemma *up-iterate-chain*: *mono F ==> up-chain (up-iterate F)*
apply (*rule up-chainI*)
apply (*induct-tac i*)
apply *simp+*
apply (*erule (1) monoD*)
done

lemma *UNION-up-iterate-is-fp*:
up-cont F ==>
F (UNION UNIV (up-iterate F)) = UNION UNIV (up-iterate F)
apply (*frule up-cont-mono* [*THEN up-iterate-chain*])
apply (*drule (1) up-contD*)
apply *simp*
apply (*auto simp del*: *up-iterate-Suc simp add*: *up-iterate-Suc* [*symmetric*])
apply (*case-tac xa*)
apply *auto*
done

lemma *UNION-up-iterate-lowerbound*:
mono F ==> F P = P ==> UNION UNIV (up-iterate F) ⊆ P
apply (*subgoal-tac* (!!*i. up-iterate F i ⊆ P*))
apply *fast*
apply (*induct-tac i*)
prefer *2* **apply** (*drule (1) monoD*)
apply *auto*
done

lemma *UNION-up-iterate-is-lfp*:
up-cont F ==> lfp F = UNION UNIV (up-iterate F)
apply (*rule set-eq-subset* [*THEN iffD2*])
apply (*rule conjI*)
prefer *2*
apply (*drule up-cont-mono*)
apply (*rule UNION-up-iterate-lowerbound*)
apply *assumption*
apply (*erule lfp-unfold* [*symmetric*])
apply (*rule lfp-lowerbound*)
apply (*rule set-eq-subset* [*THEN iffD1*, *THEN conjunct2*])
apply (*erule UNION-up-iterate-is-fp* [*symmetric*])
done

constdefs
down-iterate :: (*'a set* => *'a set*) => *nat* => *'a set*
down-iterate f n == (*f^n*) *UNIV*

lemma *down-iterate-0* [*simp*]: *down-iterate f 0* = *UNIV*
by (*simp add*: *down-iterate-def*)

lemma *down-iterate-Suc* [*simp*]:
down-iterate f (*Suc i*) = *f* (*down-iterate f i*)
by (*simp add*: *down-iterate-def*)

lemma *down-iterate-chain*: *mono F* ==> *down-chain* (*down-iterate F*)
apply (*rule down-chainI*)
apply (*induct-tac i*)
apply *simp+*
apply (*erule* (*1*) *monoD*)
done

lemma *INTER-down-iterate-is-fp*:
down-cont F ==>
F (*INTER UNIV* (*down-iterate F*)) = *INTER UNIV* (*down-iterate F*)
apply (*frule down-cont-mono* [*THEN down-iterate-chain*])
apply (*drule* (*1*) *down-contD*)
apply *simp*
apply (*auto simp del*: *down-iterate-Suc simp add*: *down-iterate-Suc* [*symmetric*])
apply (*case-tac xa*)
apply *auto*
done

lemma *INTER-down-iterate-upperbound*:
mono F ==> *F P* = *P* ==> *P* ⊆ *INTER UNIV* (*down-iterate F*)
apply (*subgoal-tac* (!!*i*. *P* ⊆ *down-iterate F i*))
apply *fast*
apply (*induct-tac i*)
prefer *2* **apply** (*drule* (*1*) *monoD*)
apply *auto*
done

lemma *INTER-down-iterate-is-gfp*:
down-cont F ==> *gfp F* = *INTER UNIV* (*down-iterate F*)
apply (*rule set-eq-subset* [*THEN iffD2*])
apply (*rule conjI*)
apply (*drule down-cont-mono*)
apply (*rule INTER-down-iterate-upperbound*)
apply *assumption*
apply (*erule gfp-unfold* [*symmetric*])
apply (*rule gfp-upperbound*)
apply (*rule set-eq-subset* [*THEN iffD1*, *THEN conjunct2*])

apply (*erule INTER-down-iterate-is-fp*)
done

end

5 EfficientNat: Implementation of natural numbers by integers

theory *EfficientNat*
imports *Main*
begin

When generating code for functions on natural numbers, the canonical representation using *0* and *Suc* is unsuitable for computations involving large numbers. The efficiency of the generated code can be improved drastically by implementing natural numbers by integers. To do this, just include this theory.

5.1 Basic functions

The implementation of *0* and *Suc* using the ML integers is straightforward. Since natural numbers are implemented using integers, the coercion function *int* of type *nat* ⇒ *int* is simply implemented by the identity function. For the *nat* function for converting an integer to a natural number, we give a specific implementation using an ML function that returns its input value, provided that it is non-negative, and otherwise returns *0*.

types-code
nat (*int*)
attach (*term-of*) ⟪
fun term-of-nat 0 = Const (*0*, *HOLogic.natT*)
| *term-of-nat 1 = Const* (*1*, *HOLogic.natT*)
| *term-of-nat i = HOLogic.number-of-const HOLogic.natT* $
HOLogic.mk-bin (*IntInf.fromInt i*);
⟫
attach (*test*) ⟪
fun gen-nat i = random-range 0 i;
⟫

consts-code
0 :: *nat* (*0*)
Suc ((- + *1*))
nat (⟨**module**⟩*nat*)
attach ⟪
fun nat i = if i < 0 then 0 else i;
⟫

int ((-))

Case analysis on natural numbers is rephrased using a conditional expression:

lemma [*code unfold*]: *nat-case* ≡ (λ*f g n*. *if n = 0 then f else g* (*n − 1*))
apply (*rule eq-reflection ext*)+
apply (*case-tac n*)
apply *simp-all*
done

Most standard arithmetic functions on natural numbers are implemented using their counterparts on the integers:

lemma [*code*]: *m − n = nat* (*int m − int n*) **by** *arith*
lemma [*code*]: *m + n = nat* (*int m + int n*) **by** *arith*
lemma [*code*]: *m ∗ n = nat* (*int m ∗ int n*)
by (*simp add*: *zmult-int*)
lemma [*code*]: *m div n = nat* (*int m div int n*)
by (*simp add*: *zdiv-int* [*symmetric*])
lemma [*code*]: *m mod n = nat* (*int m mod int n*)
by (*simp add*: *zmod-int* [*symmetric*])
lemma [*code*]: (*m < n*) = (*int m < int n*)
by *simp*

5.2 Preprocessors

In contrast to *Suc n*, the term *n + 1* is no longer a constructor term. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a recursion equation or in the arguments of an inductive relation in an introduction rule) must be eliminated. This can be accomplished by applying the following transformation rules:

theorem *Suc-if-eq*: (⋀*n*. *f* (*Suc n*) = *h n*) ⟹ *f 0* = *g* ⟹
f n = (*if n = 0 then g else h* (*n − 1*))
by (*case-tac n*) *simp-all*

theorem *Suc-clause*: (⋀*n*. *P n* (*Suc n*)) ⟹ *n* ≠ *0* ⟹ *P* (*n − 1*) *n*
by (*case-tac n*) *simp-all*

The rules above are built into a preprocessor that is plugged into the code generator. Since the preprocessor for introduction rules does not know anything about modes, some of the modes that worked for the canonical representation of natural numbers may no longer work.

end

6 ExecutableSet: Implementation of finite sets by lists

theory *ExecutableSet*
imports *Main*
begin

lemma [*code target*: *Set*]: $(A = B) = (A \subseteq B \wedge B \subseteq A)$
by *blast*

declare *bex-triv-one-point1* [*symmetric*, *standard*, *code*]

types-code
set (- *list*)
attach (*term-of*) ⟪
fun term-of-set f T [] = *Const* ({}, *Type* (*set*, [*T*]))
| *term-of-set f T* (*x* :: *xs*) = *Const* (*insert*,
T −−> *Type* (*set*, [*T*]) −−> *Type* (*set*, [*T*])) $ *f x* $ *term-of-set f T xs*;
⟫
attach (*test*) ⟪
fun gen-set′ aG i j = *frequency*
[(*i*, *fn* () => *aG j* :: *gen-set′ aG* (*i*−*1*) *j*), (*1*, *fn* () => [])] ()
and gen-set aG i = *gen-set′ aG i i*;
⟫

consts-code
{} ([])
insert ((- *ins* -))
op Un ((- *union* -))
op Int ((- *inter* -))
op − :: *′a set* ⇒ *′a set* ⇒ *′a set* ((- \\ -))
image (⟨**module**⟩*image*)
attach ⟪
fun image f S = *distinct* (*map f S*);
⟫
UNION (⟨**module**⟩*UNION*)
attach ⟪
fun UNION S f = *Library.foldr Library.union* (*map f S*, []);
⟫
INTER (⟨**module**⟩*INTER*)
attach ⟪
fun INTER S f = *Library.foldr1 Library.inter* (*map f S*);
⟫
Bex (⟨**module**⟩*Bex*)
attach ⟪
fun Bex S P = *Library.exists P S*;
⟫
Ball (⟨**module**⟩*Ball*)
attach ⟪

fun Ball S P = Library.forall P S;
⟫

end

7 FuncSet: Pi and Function Sets

theory *FuncSet*
imports *Main*
begin

constdefs
Pi :: [*'a set*, *'a => 'b set*] => (*'a => 'b*) *set*
Pi A B == {*f*. ∀*x*. *x* ∈ *A* −−> *f x* ∈ *B x*}

extensional :: *'a set* => (*'a => 'b*) *set*
extensional A == {*f*. ∀*x*. *x*~:*A* −−> *f x* = *arbitrary*}

restrict :: [*'a => 'b*, *'a set*] => (*'a => 'b*)
restrict f A == (%*x*. *if x* ∈ *A then f x else arbitrary*)

syntax
@*Pi* :: [*pttrn*, *'a set*, *'b set*] => (*'a => 'b*) *set* ((*3PI* -:-./ -) *10*)
funcset :: [*'a set*, *'b set*] => (*'a => 'b*) *set* (**infixr** −> *60*)
@*lam* :: [*pttrn*, *'a set*, *'a => 'b*] => (*'a=>'b*) ((*3*%-:-./ -) [*0*,*0*,*3*] *3*)

syntax (*xsymbols*)
@*Pi* :: [*pttrn*, *'a set*, *'b set*] => (*'a => 'b*) *set* ((*3*Π -∈-./ -) *10*)
funcset :: [*'a set*, *'b set*] => (*'a => 'b*) *set* (**infixr** → *60*)
@*lam* :: [*pttrn*, *'a set*, *'a => 'b*] => (*'a=>'b*) ((*3*λ-∈-./ -) [*0*,*0*,*3*] *3*)

syntax (*HTML* **output**)
@*Pi* :: [*pttrn*, *'a set*, *'b set*] => (*'a => 'b*) *set* ((*3*Π -∈-./ -) *10*)
@*lam* :: [*pttrn*, *'a set*, *'a => 'b*] => (*'a=>'b*) ((*3*λ-∈-./ -) [*0*,*0*,*3*] *3*)

translations
PI x:*A*. *B* => *Pi A* (%*x*. *B*)
A −> *B* => *Pi A* (-*K B*)
%*x*:*A*. *f* == *restrict* (%*x*. *f*) *A*

constdefs
compose :: [*'a set*, *'b => 'c*, *'a => 'b*] => (*'a => 'c*)
compose A g f == λ*x*∈*A*. *g* (*f x*)

print-translation ⟪ [(*Pi*, *dependent-tr'* (@*Pi*, *funcset*))] ⟫

7.1 Basic Properties of *Pi*

lemma *Pi-I*: (!!*x*. *x* ∈ *A* ==> *f x* ∈ *B x*) ==> *f* ∈ *Pi A B*
by (*simp add*: *Pi-def*)

lemma *funcsetI*: (!!*x*. *x* ∈ *A* ==> *f x* ∈ *B*) ==> *f* ∈ *A* -> *B*
by (*simp add*: *Pi-def*)

lemma *Pi-mem*: [|*f*: *Pi A B*; *x* ∈ *A*|] ==> *f x* ∈ *B x*
by (*simp add*: *Pi-def*)

lemma *funcset-mem*: [|*f* ∈ *A* -> *B*; *x* ∈ *A*|] ==> *f x* ∈ *B*
by (*simp add*: *Pi-def*)

lemma *funcset-image*: *f* ∈ *A*→*B* ==> *f* ‘ *A* ⊆ *B*
by (*auto simp add*: *Pi-def*)

lemma *Pi-eq-empty*: ((*PI x*: *A*. *B x*) = {}) = (∃ *x*∈*A*. *B*(*x*) = {})
apply (*simp add*: *Pi-def*, *auto*)

Converse direction requires Axiom of Choice to exhibit a function picking an element from each non-empty *B x*

apply (*drule-tac x* = %*u*. *SOME y*. *y* ∈ *B u* **in** *spec*, *auto*)
apply (*cut-tac P*= %*y*. *y* ∈ *B x* **in** *some-eq-ex*, *auto*)
done

lemma *Pi-empty* [*simp*]: *Pi* {} *B* = *UNIV*
by (*simp add*: *Pi-def*)

lemma *Pi-UNIV* [*simp*]: *A* -> *UNIV* = *UNIV*
by (*simp add*: *Pi-def*)

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: (!!*x*. *x* ∈ *A* ==> *B x* <= *C x*) ==> *Pi A B* <= *Pi A C*
by (*simp add*: *Pi-def*, *blast*)

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: *A'* <= *A* ==> *Pi A B* <= *Pi A' B*
by (*simp add*: *Pi-def*, *blast*)

7.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*:
[| *f* ∈ *A* -> *B*; *g* ∈ *B* -> *C* |]==> *compose A g f* ∈ *A* -> *C*
by (*simp add*: *Pi-def compose-def restrict-def*)

lemma *compose-assoc*:
[| *f* ∈ *A* -> *B*; *g* ∈ *B* -> *C*; *h* ∈ *C* -> *D* |]
==> *compose A h* (*compose A g f*) = *compose A* (*compose B h g*) *f*

by (*simp add*: *expand-fun-eq Pi-def compose-def restrict-def*)

lemma *compose-eq*: $x \in A$ ==> *compose* $A\ g\ f\ x = g(f(x))$
by (*simp add*: *compose-def restrict-def*)

lemma *surj-compose*: [| f ‘ $A = B$; g ‘ $B = C$ |] ==> *compose* $A\ g\ f$ ‘ $A = C$
by (*auto simp add*: *image-def compose-eq*)

7.3 Bounded Abstraction: *restrict*

lemma *restrict-in-funcset*: (!!x. $x \in A$ ==> $f\ x \in B$) ==> ($\lambda x{\in}A.\ f\ x$) $\in A$ -> B
by (*simp add*: *Pi-def restrict-def*)

lemma *restrictI*: (!!x. $x \in A$ ==> $f\ x \in B\ x$) ==> ($\lambda x{\in}A.\ f\ x$) $\in$ *Pi* $A\ B$
by (*simp add*: *Pi-def restrict-def*)

lemma *restrict-apply* [*simp*]:
($\lambda y{\in}A.\ f\ y$) x = (*if* $x \in A$ *then* $f\ x$ *else arbitrary*)
by (*simp add*: *restrict-def*)

lemma *restrict-ext*:
(!!x. $x \in A$ ==> $f\ x = g\ x$) ==> ($\lambda x{\in}A.\ f\ x$) = ($\lambda x{\in}A.\ g\ x$)
by (*simp add*: *expand-fun-eq Pi-def Pi-def restrict-def*)

lemma *inj-on-restrict-eq* [*simp*]: *inj-on* (*restrict* $f\ A$) A = *inj-on* $f\ A$
by (*simp add*: *inj-on-def restrict-def*)

lemma *Id-compose*:
[|$f \in A$ -> B; $f \in$ *extensional* A|] ==> *compose* A ($\lambda y{\in}B.\ y$) $f = f$
by (*auto simp add*: *expand-fun-eq compose-def extensional-def Pi-def*)

lemma *compose-Id*:
[|$g \in A$ -> B; $g \in$ *extensional* A|] ==> *compose* $A\ g$ ($\lambda x{\in}A.\ x$) = g
by (*auto simp add*: *expand-fun-eq compose-def extensional-def Pi-def*)

lemma *image-restrict-eq* [*simp*]: (*restrict* $f\ A$) ‘ A = f ‘ A
by (*auto simp add*: *restrict-def*)

7.4 Bijections Between Sets

The basic definition could be moved to *Fun.thy*, but most of the theorems belong here, or need at least *Hilbert-Choice*.

constdefs
bij-betw :: ['*a* => '*b*, '*a set*, '*b set*] => *bool*
bij-betw $f\ A\ B$ == *inj-on* $f\ A$ & f ‘ $A = B$

lemma *bij-betw-imp-inj-on*: *bij-betw* $f\ A\ B \Longrightarrow$ *inj-on* $f\ A$
by (*simp add*: *bij-betw-def*)

lemma *bij-betw-imp-funcset*: *bij-betw f A B* $\Longrightarrow$ $f \in A \to B$
by (*auto simp add*: *bij-betw-def inj-on-Inv Pi-def*)

lemma *bij-betw-Inv*: *bij-betw f A B* $\Longrightarrow$ *bij-betw* (*Inv A f*) *B A*
apply (*auto simp add*: *bij-betw-def inj-on-Inv Inv-mem*)
apply (*simp add*: *image-compose* [*symmetric*] *o-def*)
apply (*simp add*: *image-def Inv-f-f*)
done

lemma *inj-on-compose*:
[| *bij-betw f A B*; *inj-on g B* |] ==> *inj-on* (*compose A g f*) *A*
by (*auto simp add*: *bij-betw-def inj-on-def compose-eq*)

lemma *bij-betw-compose*:
[| *bij-betw f A B*; *bij-betw g B C* |] ==> *bij-betw* (*compose A g f*) *A C*
apply (*simp add*: *bij-betw-def compose-eq inj-on-compose*)
apply (*auto simp add*: *compose-def image-def*)
done

lemma *bij-betw-restrict-eq* [*simp*]:
bij-betw (*restrict f A*) *A B* = *bij-betw f A B*
by (*simp add*: *bij-betw-def*)

7.5 Extensionality

lemma *extensional-arb*: [|$f \in$ *extensional A*; $x \notin A$|] ==> $f\ x$ = *arbitrary*
by (*simp add*: *extensional-def*)

lemma *restrict-extensional* [*simp*]: *restrict f A* $\in$ *extensional A*
by (*simp add*: *restrict-def extensional-def*)

lemma *compose-extensional* [*simp*]: *compose A f g* $\in$ *extensional A*
by (*simp add*: *compose-def*)

lemma *extensionalityI*:
[| $f \in$ *extensional A*; $g \in$ *extensional A*;
!!x. $x \in A$ ==> $f\ x = g\ x$ |] ==> $f = g$
by (*force simp add*: *expand-fun-eq extensional-def*)

lemma *Inv-funcset*: f ` $A = B$ ==> $(\lambda x \in B.$ *Inv A f* $x) : B \to A$
by (*unfold Inv-def*) (*fast intro*: *restrict-in-funcset someI2*)

lemma *compose-Inv-id*:
bij-betw f A B ==> *compose A* $(\lambda y \in B.$ *Inv A f y*) *f* = $(\lambda x \in A.\ x)$
apply (*simp add*: *bij-betw-def compose-def*)
apply (*rule restrict-ext*, *auto*)
apply (*erule subst*)
apply (*simp add*: *Inv-f-f*)

done

lemma *compose-id-Inv*:
f ' A = B ==> compose B f (λ*y*∈*B*. *Inv A f y*) = (λ*x*∈*B*. *x*)
apply (*simp add*: *compose-def*)
apply (*rule restrict-ext*)
apply (*simp add*: *f-Inv-f*)
done

7.6 Cardinality

lemma *card-inj*: [|*f* ∈ *A*→*B*; *inj-on f A*; *finite B*|] ==> *card*(*A*) ≤ *card*(*B*)
apply (*rule card-inj-on-le*)
apply (*auto simp add*: *Pi-def*)
done

lemma *card-bij*:
[|*f* ∈ *A*→*B*; *inj-on f A*;
g ∈ *B*→*A*; *inj-on g B*; *finite A*; *finite B*|] ==> *card*(*A*) = *card*(*B*)
by (*blast intro*: *card-inj order-antisym*)

end

8 Multiset: Multisets

theory *Multiset*
imports *Accessible-Part*
begin

8.1 The type of multisets

typedef *'a multiset* = {*f*::*'a* => *nat*. *finite* {*x* . *0* < *f x*}}
proof
show (λ*x*. *0*::*nat*) ∈ *?multiset* **by** *simp*
qed

lemmas *multiset-typedef* [*simp*] =
Abs-multiset-inverse Rep-multiset-inverse Rep-multiset
and [*simp*] = *Rep-multiset-inject* [*symmetric*]

constdefs
Mempty :: *'a multiset* ({#})
{#} == *Abs-multiset* (λ*a*. *0*)

single :: *'a* => *'a multiset* ({#-#})
{#*a*#} == *Abs-multiset* (λ*b*. *if b = a then 1 else 0*)

count :: *'a multiset* => *'a* => *nat*

count == *Rep-multiset*

MCollect :: *'a multiset* => (*'a* => *bool*) => *'a multiset*
MCollect M P == *Abs-multiset* (λ*x*. *if P x then Rep-multiset M x else 0*)

syntax
-Melem :: *'a* => *'a multiset* => *bool* ((-/ :# -) [*50*, *51*] *50*)
-MCollect :: *pttrn* => *'a multiset* => *bool* => *'a multiset* ((*1*{# - : -./ -#}))
translations
a :# *M* == *0* < *count M a*
{#*x*:*M*. *P*#} == *MCollect M* (λ*x*. *P*)

constdefs
set-of :: *'a multiset* => *'a set*
set-of M == {*x*. *x* :# *M*}

instance *multiset* :: (*type*) {*plus*, *minus*, *zero*} **..**

defs (**overloaded**)
union-def: *M* + *N* == *Abs-multiset* (λ*a*. *Rep-multiset M a* + *Rep-multiset N a*)
diff-def: *M* − *N* == *Abs-multiset* (λ*a*. *Rep-multiset M a* − *Rep-multiset N a*)
Zero-multiset-def [*simp*]: *0* == {#}
size-def: *size M* == *setsum* (*count M*) (*set-of M*)

constdefs
multiset-inter :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* (**infixl** #∩ *70*)
multiset-inter A B ≡ *A* − (*A* − *B*)

Preservation of the representing set *multiset*.

lemma *const0-in-multiset* [*simp*]: (λ*a*. *0*) ∈ *multiset*
by (*simp add*: *multiset-def*)

lemma *only1-in-multiset* [*simp*]: (λ*b*. *if b* = *a then 1 else 0*) ∈ *multiset*
by (*simp add*: *multiset-def*)

lemma *union-preserves-multiset* [*simp*]:
M ∈ *multiset* ==> *N* ∈ *multiset* ==> (λ*a*. *M a* + *N a*) ∈ *multiset*
apply (*simp add*: *multiset-def*)
apply (*drule* (*1*) *finite-UnI*)
apply (*simp del*: *finite-Un add*: *Un-def*)
done

lemma *diff-preserves-multiset* [*simp*]:
M ∈ *multiset* ==> (λ*a*. *M a* − *N a*) ∈ *multiset*
apply (*simp add*: *multiset-def*)
apply (*rule finite-subset*)
apply *auto*
done

8.2 Algebraic properties of multisets

8.2.1 Union

lemma *union-empty* [*simp*]: $M + \{\#\} = M \wedge \{\#\} + M = M$
by (*simp add*: *union-def Mempty-def*)

lemma *union-commute*: $M + N = N + (M::'a\ multiset)$
by (*simp add*: *union-def add-ac*)

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ multiset))$
by (*simp add*: *union-def add-ac*)

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
proof −
have $M + (N + K) = (N + K) + M$
by (*rule union-commute*)
also have $\ldots = N + (K + M)$
by (*rule union-assoc*)
also have $K + M = M + K$
by (*rule union-commute*)
finally show *?thesis* **.**
qed

lemmas *union-ac* = *union-assoc union-commute union-lcomm*

instance *multiset* :: (*type*) *comm-monoid-add*
proof
fix *a b c* :: *'a multiset*
show $(a + b) + c = a + (b + c)$ **by** (*rule union-assoc*)
show $a + b = b + a$ **by** (*rule union-commute*)
show $0 + a = a$ **by** *simp*
qed

8.2.2 Difference

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
by (*simp add*: *Mempty-def diff-def*)

lemma *diff-union-inverse2* [*simp*]: $M + \{\#a\#\} - \{\#a\#\} = M$
by (*simp add*: *union-def diff-def*)

8.2.3 Count of elements

lemma *count-empty* [*simp*]: *count* $\{\#\}$ *a* = *0*
by (*simp add*: *count-def Mempty-def*)

lemma *count-single* [*simp*]: *count* $\{\#b\#\}$ *a* = (*if b* = *a then 1 else 0*)
by (*simp add*: *count-def single-def*)

lemma *count-union* [*simp*]: *count* (*M* + *N*) *a* = *count M a* + *count N a*
by (*simp add*: *count-def union-def*)

lemma *count-diff* [*simp*]: *count* (*M* − *N*) *a* = *count M a* − *count N a*
by (*simp add*: *count-def diff-def*)

8.2.4 Set of elements

lemma *set-of-empty* [*simp*]: *set-of* {#} = {}
by (*simp add*: *set-of-def*)

lemma *set-of-single* [*simp*]: *set-of* {#*b*#} = {*b*}
by (*simp add*: *set-of-def*)

lemma *set-of-union* [*simp*]: *set-of* (*M* + *N*) = *set-of M* ∪ *set-of N*
by (*auto simp add*: *set-of-def*)

lemma *set-of-eq-empty-iff* [*simp*]: (*set-of M* = {}) = (*M* = {#})
by (*auto simp add*: *set-of-def Mempty-def count-def expand-fun-eq*)

lemma *mem-set-of-iff* [*simp*]: (*x* ∈ *set-of M*) = (*x* :# *M*)
by (*auto simp add*: *set-of-def*)

8.2.5 Size

lemma *size-empty* [*simp*]: *size* {#} = *0*
by (*simp add*: *size-def*)

lemma *size-single* [*simp*]: *size* {#*b*#} = *1*
by (*simp add*: *size-def*)

lemma *finite-set-of* [*iff*]: *finite* (*set-of M*)
using *Rep-multiset* [*of M*]
by (*simp add*: *multiset-def set-of-def count-def*)

lemma *setsum-count-Int*:
finite A ==> *setsum* (*count N*) (*A* ∩ *set-of N*) = *setsum* (*count N*) *A*
apply (*erule finite-induct*)
apply *simp*
apply (*simp add*: *Int-insert-left set-of-def*)
done

lemma *size-union* [*simp*]: *size* (*M* + *N*::'*a multiset*) = *size M* + *size N*
apply (*unfold size-def*)
apply (*subgoal-tac count* (*M* + *N*) = (λ*a*. *count M a* + *count N a*))
prefer *2*
apply (*rule ext*, *simp*)
apply (*simp* (*no-asm-simp*) *add*: *setsum-Un-nat setsum-addf setsum-count-Int*)
apply (*subst Int-commute*)
apply (*simp* (*no-asm-simp*) *add*: *setsum-count-Int*)

done

lemma *size-eq-0-iff-empty* [*iff*]: (*size M = 0*) = (*M = {#}*)
apply (*unfold size-def Mempty-def count-def*, *auto*)
apply (*simp add*: *set-of-def count-def expand-fun-eq*)
done

lemma *size-eq-Suc-imp-elem*: *size M = Suc n ==> ∃ a. a :# M*
apply (*unfold size-def*)
apply (*drule setsum-SucD*, *auto*)
done

8.2.6 Equality of multisets

lemma *multiset-eq-conv-count-eq*: (*M = N*) = (*∀ a. count M a = count N a*)
by (*simp add*: *count-def expand-fun-eq*)

lemma *single-not-empty* [*simp*]: *{#a#} ≠ {#} ∧ {#} ≠ {#a#}*
by (*simp add*: *single-def Mempty-def expand-fun-eq*)

lemma *single-eq-single* [*simp*]: (*{#a#} = {#b#}*) = (*a = b*)
by (*auto simp add*: *single-def expand-fun-eq*)

lemma *union-eq-empty* [*iff*]: (*M + N = {#}*) = (*M = {#} ∧ N = {#}*)
by (*auto simp add*: *union-def Mempty-def expand-fun-eq*)

lemma *empty-eq-union* [*iff*]: (*{#} = M + N*) = (*M = {#} ∧ N = {#}*)
by (*auto simp add*: *union-def Mempty-def expand-fun-eq*)

lemma *union-right-cancel* [*simp*]: (*M + K = N + K*) = (*M = (N::'a multiset)*)
by (*simp add*: *union-def expand-fun-eq*)

lemma *union-left-cancel* [*simp*]: (*K + M = K + N*) = (*M = (N::'a multiset)*)
by (*simp add*: *union-def expand-fun-eq*)

lemma *union-is-single*:
(*M + N = {#a#}*) = (*M = {#a#} ∧ N={#} ∨ M = {#} ∧ N = {#a#}*)
apply (*simp add*: *Mempty-def single-def union-def add-is-1 expand-fun-eq*)
apply *blast*
done

lemma *single-is-union*:
(*{#a#} = M + N*) = (*{#a#} = M ∧ N = {#} ∨ M = {#} ∧ {#a#} = N*)
apply (*unfold Mempty-def single-def union-def*)
apply (*simp add*: *add-is-1 one-is-add expand-fun-eq*)
apply (*blast dest*: *sym*)
done

lemma *add-eq-conv-diff*:
(*M* + {#*a*#} = *N* + {#*b*#}) =
(*M* = *N* ∧ *a* = *b* ∨ *M* = *N* − {#*a*#} + {#*b*#} ∧ *N* = *M* − {#*b*#} + {#*a*#})
apply (*unfold single-def union-def diff-def*)
apply (*simp* (*no-asm*) *add*: *expand-fun-eq*)
apply (*rule conjI*, *force*, *safe*, *simp-all*)
apply (*simp add*: *eq-sym-conv*)
done

declare *Rep-multiset-inject* [*symmetric*, *simp del*]

8.2.7 Intersection

lemma *multiset-inter-count*:
count (*A* #∩ *B*) *x* = *min* (*count A x*) (*count B x*)
by (*simp add*: *multiset-inter-def min-def*)

lemma *multiset-inter-commute*: *A* #∩ *B* = *B* #∩ *A*
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count*
min-max.below-inf.inf-commute)

lemma *multiset-inter-assoc*: *A* #∩ (*B* #∩ *C*) = *A* #∩ *B* #∩ *C*
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count*
min-max.below-inf.inf-assoc)

lemma *multiset-inter-left-commute*: *A* #∩ (*B* #∩ *C*) = *B* #∩ (*A* #∩ *C*)
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-def*)

lemmas *multiset-inter-ac* =
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *multiset-union-diff-commute*: *B* #∩ *C* = {#} ⟹ *A* + *B* − *C* = *A* − *C* + *B*
apply (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-def*
split: *split-if-asm*)
apply *clarsimp*
apply (*erule-tac x* = *a* **in** *allE*)
apply *auto*
done

8.3 Induction over multisets

lemma *setsum-decr*:
finite F ==> (*0*::*nat*) < *f a* ==>
setsum (*f* (*a* := *f a* − *1*)) *F* = (*if a*∈*F then setsum f F* − *1 else setsum f F*)
apply (*erule finite-induct*, *auto*)
apply (*drule-tac a* = *a* **in** *mk-disjoint-insert*, *auto*)

```
  done

lemma rep-multiset-induct-aux:
  assumes P (λa. (0::nat))
    and !!f b. f ∈ multiset ==> P f ==> P (f (b := f b + 1))
  shows ∀f. f ∈ multiset --> setsum f {x. 0 < f x} = n --> P f
proof −
  note premises = prems [unfolded multiset-def]
  show ?thesis
    apply (unfold multiset-def)
    apply (induct-tac n, simp, clarify)
     apply (subgoal-tac f = (λa.0))
      apply simp
      apply (rule premises)
     apply (rule ext, force, clarify)
    apply (frule setsum-SucD, clarify)
    apply (rename-tac a)
    apply (subgoal-tac finite {x. 0 < (f (a := f a − 1)) x})
     prefer 2
     apply (rule finite-subset)
      prefer 2
      apply assumption
     apply simp
     apply blast
    apply (subgoal-tac f = (f (a := f a − 1))(a := (f (a := f a − 1)) a + 1))
     prefer 2
     apply (rule ext)
     apply (simp (no-asm-simp))
     apply (erule ssubst, rule premises, blast)
    apply (erule allE, erule impE, erule-tac [2] mp, blast)
    apply (simp (no-asm-simp) add: setsum-decr del: fun-upd-apply One-nat-def)
    apply (subgoal-tac {x. x ≠ a --> 0 < f x} = {x. 0 < f x})
     prefer 2
     apply blast
    apply (subgoal-tac {x. x ≠ a ∧ 0 < f x} = {x. 0 < f x} − {a})
     prefer 2
     apply blast
    apply (simp add: le-imp-diff-is-add setsum-diff1-nat cong: conj-cong)
    done
qed

theorem rep-multiset-induct:
  f ∈ multiset ==> P (λa. 0) ==>
    (!!f b. f ∈ multiset ==> P f ==> P (f (b := f b + 1))) ==> P f
  using rep-multiset-induct-aux by blast

theorem multiset-induct [induct type: multiset]:
  assumes prem1: P {#}
    and prem2: !!M x. P M ==> P (M + {#x#})
```

```
  shows P M
proof −
  note defns = union-def single-def Mempty-def
  show ?thesis
    apply (rule Rep-multiset-inverse [THEN subst])
    apply (rule Rep-multiset [THEN rep-multiset-induct])
     apply (rule prem1 [unfolded defns])
    apply (subgoal-tac f(b := f b + 1) = (λa. f a + (if a=b then 1 else 0)))
     prefer 2
     apply (simp add: expand-fun-eq)
    apply (erule ssubst)
    apply (erule Abs-multiset-inverse [THEN subst])
    apply (erule prem2 [unfolded defns, simplified])
    done
qed

lemma MCollect-preserves-multiset:
    M ∈ multiset ==> (λx. if P x then M x else 0) ∈ multiset
  apply (simp add: multiset-def)
  apply (rule finite-subset, auto)
  done

lemma count-MCollect [simp]:
    count {# x:M. P x #} a = (if P a then count M a else 0)
  by (simp add: count-def MCollect-def MCollect-preserves-multiset)

lemma set-of-MCollect [simp]: set-of {# x:M. P x #} = set-of M ∩ {x. P x}
  by (auto simp add: set-of-def)

lemma multiset-partition: M = {# x:M. P x #} + {# x:M. ¬ P x #}
  by (subst multiset-eq-conv-count-eq, auto)

lemma add-eq-conv-ex:
  (M + {#a#} = N + {#b#}) =
    (M = N ∧ a = b ∨ (∃K. M = K + {#b#} ∧ N = K + {#a#}))
  by (auto simp add: add-eq-conv-diff)

declare multiset-typedef [simp del]
```

8.4 Multiset orderings

8.4.1 Well-foundedness

```
constdefs
  mult1 :: ('a × 'a) set => ('a multiset × 'a multiset) set
  mult1 r ==
    {(N, M). ∃a M0 K. M = M0 + {#a#} ∧ N = M0 + K ∧
      (∀b. b :# K −−> (b, a) ∈ r)}

  mult :: ('a × 'a) set => ('a multiset × 'a multiset) set
```

mult r == $(mult1\ r)^+$

lemma *not-less-empty* [*iff*]: $(M, \{\#\}) \notin mult1\ r$
by (*simp add*: *mult1-def*)

lemma *less-add*: $(N, M0 + \{\#a\#\}) \in mult1\ r \Longrightarrow$
$(\exists M.\ (M, M0) \in mult1\ r \wedge N = M + \{\#a\#\}) \vee$
$(\exists K.\ (\forall b.\ b :\# K \longrightarrow (b, a) \in r) \wedge N = M0 + K)$
(**concl is** *?case1* $(mult1\ r) \vee$ *?case2*)
proof (*unfold mult1-def*)
let *?r* $= \lambda K\ a.\ \forall b.\ b :\# K \longrightarrow (b, a) \in r$
let *?R* $= \lambda N\ M.\ \exists a\ M0\ K.\ M = M0 + \{\#a\#\} \wedge N = M0 + K \wedge ?r\ K\ a$
let *?case1* = *?case1* $\{(N, M).\ ?R\ N\ M\}$

assume $(N, M0 + \{\#a\#\}) \in \{(N, M).\ ?R\ N\ M\}$
hence $\exists a'\ M0'\ K.$
$M0 + \{\#a\#\} = M0' + \{\#a'\#\} \wedge N = M0' + K \wedge ?r\ K\ a'$ **by** *simp*
thus *?case1* $\vee$ *?case2*
proof (*elim exE conjE*)
fix $a'\ M0'\ K$
assume *N*: $N = M0' + K$ **and** *r*: $?r\ K\ a'$
assume $M0 + \{\#a\#\} = M0' + \{\#a'\#\}$
hence $M0 = M0' \wedge a = a' \vee$
$(\exists K'.\ M0 = K' + \{\#a'\#\} \wedge M0' = K' + \{\#a\#\})$
by (*simp only*: *add-eq-conv-ex*)
thus *?thesis*
proof (*elim disjE conjE exE*)
assume $M0 = M0'\ a = a'$
with *N r* **have** $?r\ K\ a \wedge N = M0 + K$ **by** *simp*
hence *?case2* **.. thus** *?thesis* **..**
next
fix K'
assume $M0' = K' + \{\#a\#\}$
with *N* **have** *n*: $N = K' + K + \{\#a\#\}$ **by** (*simp add*: *union-ac*)

assume $M0 = K' + \{\#a'\#\}$
with *r* **have** $?R\ (K' + K)\ M0$ **by** *blast*
with *n* **have** *?case1* **by** *simp* **thus** *?thesis* **..**
qed
qed
qed

lemma *all-accessible*: $wf\ r \Longrightarrow \forall M.\ M \in acc\ (mult1\ r)$
proof
let *?R* = *mult1 r*
let *?W* = *acc ?R*
{
fix *M M0 a*
assume *M0*: $M0 \in ?W$

```
    and wf-hyp: !!b. (b, a) ∈ r ==> (∀M ∈ ?W. M + {#b#} ∈ ?W)
    and acc-hyp: ∀M. (M, M0) ∈ ?R --> M + {#a#} ∈ ?W
  have M0 + {#a#} ∈ ?W
  proof (rule accI [of M0 + {#a#}])
    fix N
    assume (N, M0 + {#a#}) ∈ ?R
    hence ((∃M. (M, M0) ∈ ?R ∧ N = M + {#a#}) ∨
        (∃K. (∀b. b :# K --> (b, a) ∈ r) ∧ N = M0 + K))
      by (rule less-add)
    thus N ∈ ?W
    proof (elim exE disjE conjE)
      fix M assume (M, M0) ∈ ?R and N: N = M + {#a#}
      from acc-hyp have (M, M0) ∈ ?R --> M + {#a#} ∈ ?W ..
      hence M + {#a#} ∈ ?W ..
      thus N ∈ ?W by (simp only: N)
    next
      fix K
      assume N: N = M0 + K
      assume ∀b. b :# K --> (b, a) ∈ r
      have ?this --> M0 + K ∈ ?W (is ?P K)
      proof (induct K)
        from M0 have M0 + {#} ∈ ?W by simp
        thus ?P {#} ..

        fix K x assume hyp: ?P K
        show ?P (K + {#x#})
        proof
          assume a: ∀b. b :# (K + {#x#}) --> (b, a) ∈ r
          hence (x, a) ∈ r by simp
          with wf-hyp have b: ∀M ∈ ?W. M + {#x#} ∈ ?W by blast

          from a hyp have M0 + K ∈ ?W by simp
          with b have (M0 + K) + {#x#} ∈ ?W ..
          thus M0 + (K + {#x#}) ∈ ?W by (simp only: union-assoc)
        qed
      qed
      hence M0 + K ∈ ?W ..
      thus N ∈ ?W by (simp only: N)
    qed
  qed
} note tedious-reasoning = this

assume wf: wf r
fix M
show M ∈ ?W
proof (induct M)
  show {#} ∈ ?W
  proof (rule accI)
    fix b assume (b, {#}) ∈ ?R
```

```
    with not-less-empty show b ∈ ?W by contradiction
   qed

   fix M a assume M ∈ ?W
   from wf have ∀ M ∈ ?W. M + {#a#} ∈ ?W
   proof induct
    fix a
    assume !!b. (b, a) ∈ r ==> (∀ M ∈ ?W. M + {#b#} ∈ ?W)
    show ∀ M ∈ ?W. M + {#a#} ∈ ?W
    proof
     fix M assume M ∈ ?W
     thus M + {#a#} ∈ ?W
      by (rule acc-induct) (rule tedious-reasoning)
    qed
   qed
   thus M + {#a#} ∈ ?W ..
  qed
qed

theorem wf-mult1: wf r ==> wf (mult1 r)
 by (rule acc-wfI, rule all-accessible)

theorem wf-mult: wf r ==> wf (mult r)
 by (unfold mult-def, rule wf-trancl, rule wf-mult1)
```

8.4.2 Closure-free presentation

```
lemma diff-union-single-conv: a :# J ==> I + J - {#a#} = I + (J - {#a#})
by (simp add: multiset-eq-conv-count-eq)
```

One direction.

```
lemma mult-implies-one-step:
  trans r ==> (M, N) ∈ mult r ==>
    ∃ I J K. N = I + J ∧ M = I + K ∧ J ≠ {#} ∧
    (∀ k ∈ set-of K. ∃ j ∈ set-of J. (k, j) ∈ r)
 apply (unfold mult-def mult1-def set-of-def)
 apply (erule converse-trancl-induct, clarify)
  apply (rule-tac x = M0 in exI, simp, clarify)
 apply (case-tac a :# K)
  apply (rule-tac x = I in exI)
  apply (simp (no-asm))
  apply (rule-tac x = (K - {#a#}) + Ka in exI)
  apply (simp (no-asm-simp) add: union-assoc [symmetric])
  apply (drule-tac f = λM. M - {#a#} in arg-cong)
  apply (simp add: diff-union-single-conv)
  apply (simp (no-asm-use) add: trans-def)
  apply blast
 apply (subgoal-tac a :# I)
  apply (rule-tac x = I - {#a#} in exI)
```

```
  apply (rule-tac x = J + {#a#} in exI)
  apply (rule-tac x = K + Ka in exI)
  apply (rule conjI)
   apply (simp add: multiset-eq-conv-count-eq split: nat-diff-split)
  apply (rule conjI)
   apply (drule-tac f = λM. M - {#a#} in arg-cong, simp)
   apply (simp add: multiset-eq-conv-count-eq split: nat-diff-split)
  apply (simp (no-asm-use) add: trans-def)
  apply blast
 apply (subgoal-tac a :# (M0 + {#a#}))
  apply simp
 apply (simp (no-asm))
 done

lemma elem-imp-eq-diff-union: a :# M ==> M = M - {#a#} + {#a#}
by (simp add: multiset-eq-conv-count-eq)

lemma size-eq-Suc-imp-eq-union: size M = Suc n ==> ∃ a N. M = N + {#a#}
 apply (erule size-eq-Suc-imp-elem [THEN exE])
 apply (drule elem-imp-eq-diff-union, auto)
 done

lemma one-step-implies-mult-aux:
 trans r ==>
  ∀ I J K. (size J = n ∧ J ≠ {#}) ∧ (∀ k ∈ set-of K. ∃ j ∈ set-of J. (k, j) ∈ r))
    --> (I + K, I + J) ∈ mult r
 apply (induct-tac n, auto)
 apply (frule size-eq-Suc-imp-eq-union, clarify)
 apply (rename-tac J', simp)
 apply (erule notE, auto)
 apply (case-tac J' = {#})
  apply (simp add: mult-def)
  apply (rule r-into-trancl)
  apply (simp add: mult1-def set-of-def, blast)
```

Now we know $J' \neq \{\#\}$.

```
 apply (cut-tac M = K and P = λx. (x, a) ∈ r in multiset-partition)
 apply (erule-tac P = ∀ k ∈ set-of K. ?P k in rev-mp)
 apply (erule ssubst)
 apply (simp add: Ball-def, auto)
 apply (subgoal-tac
   ((I + {# x : K. (x, a) ∈ r #}) + {# x : K. (x, a) ∉ r #},
     (I + {# x : K. (x, a) ∈ r #}) + J') ∈ mult r)
  prefer 2
  apply force
 apply (simp (no-asm-use) add: union-assoc [symmetric] mult-def)
 apply (erule trancl-trans)
 apply (rule r-into-trancl)
 apply (simp add: mult1-def set-of-def)
```

apply (*rule-tac* $x = a$ **in** *exI*)
apply (*rule-tac* $x = I + J'$ **in** *exI*)
apply (*simp add*: *union-ac*)
done

lemma *one-step-implies-mult*:
$\mathit{trans}\ r \Longrightarrow J \neq \{\#\} \Longrightarrow \forall k \in \mathit{set\text{-}of}\ K.\ \exists j \in \mathit{set\text{-}of}\ J.\ (k, j) \in r$
$\Longrightarrow (I + K, I + J) \in \mathit{mult}\ r$
apply (*insert one-step-implies-mult-aux*, *blast*)
done

8.4.3 Partial-order properties

instance *multiset* :: (*type*) *ord* **..**

defs (**overloaded**)
less-multiset-def: $M' < M == (M', M) \in \mathit{mult}\ \{(x', x).\ x' < x\}$
le-multiset-def: $M' <= M == M' = M \vee M' < (M::'a\ \mathit{multiset})$

lemma *trans-base-order*: $\mathit{trans}\ \{(x', x).\ x' < (x::'a::\mathit{order})\}$
apply (*unfold trans-def*)
apply (*blast intro*: *order-less-trans*)
done

Irreflexivity.

lemma *mult-irrefl-aux*:
$\mathit{finite}\ A \Longrightarrow (\forall x \in A.\ \exists y \in A.\ x < (y::'a::\mathit{order})) \longrightarrow A = \{\}$
apply (*erule finite-induct*)
apply (*auto intro*: *order-less-trans*)
done

lemma *mult-less-not-refl*: $\neg\ M < (M::'a::\mathit{order}\ \mathit{multiset})$
apply (*unfold less-multiset-def*, *auto*)
apply (*drule trans-base-order* [*THEN mult-implies-one-step*], *auto*)
apply (*drule finite-set-of* [*THEN mult-irrefl-aux* [*rule-format* (*no-asm*)]])
apply (*simp add*: *set-of-eq-empty-iff*)
done

lemma *mult-less-irrefl* [*elim*!]: $M < (M::'a::\mathit{order}\ \mathit{multiset}) \Longrightarrow R$
by (*insert mult-less-not-refl*, *fast*)

Transitivity.

theorem *mult-less-trans*: $K < M \Longrightarrow M < N \Longrightarrow K < (N::'a::\mathit{order}\ \mathit{multiset})$
apply (*unfold less-multiset-def mult-def*)
apply (*blast intro*: *trancl-trans*)
done

Asymmetry.

theorem *mult-less-not-sym*: $M < N \Longrightarrow \neg\ N < (M::'a::\mathit{order}\ \mathit{multiset})$

apply *auto*
apply (*rule mult-less-not-refl* [*THEN notE*])
apply (*erule mult-less-trans*, *assumption*)
done

theorem *mult-less-asym*:
M < *N* ==> (¬ *P* ==> *N* < (*M*::'*a*::*order multiset*)) ==> *P*
by (*insert mult-less-not-sym*, *blast*)

theorem *mult-le-refl* [*iff*]: *M* <= (*M*::'*a*::*order multiset*)
by (*unfold le-multiset-def*, *auto*)

Anti-symmetry.

theorem *mult-le-antisym*:
M <= *N* ==> *N* <= *M* ==> *M* = (*N*::'*a*::*order multiset*)
apply (*unfold le-multiset-def*)
apply (*blast dest*: *mult-less-not-sym*)
done

Transitivity.

theorem *mult-le-trans*:
K <= *M* ==> *M* <= *N* ==> *K* <= (*N*::'*a*::*order multiset*)
apply (*unfold le-multiset-def*)
apply (*blast intro*: *mult-less-trans*)
done

theorem *mult-less-le*: (*M* < *N*) = (*M* <= *N* ∧ *M* ≠ (*N*::'*a*::*order multiset*))
by (*unfold le-multiset-def*, *auto*)

Partial order.

instance *multiset* :: (*order*) *order*
apply *intro-classes*
apply (*rule mult-le-refl*)
apply (*erule mult-le-trans*, *assumption*)
apply (*erule mult-le-antisym*, *assumption*)
apply (*rule mult-less-le*)
done

8.4.4 Monotonicity of multiset union

lemma *mult1-union*:
(*B*, *D*) ∈ *mult1 r* ==> *trans r* ==> (*C* + *B*, *C* + *D*) ∈ *mult1 r*
apply (*unfold mult1-def*, *auto*)
apply (*rule-tac x* = *a* **in** *exI*)
apply (*rule-tac x* = *C* + *M0* **in** *exI*)
apply (*simp add*: *union-assoc*)
done

lemma *union-less-mono2*: *B* < *D* ==> *C* + *B* < *C* + (*D*::'*a*::*order multiset*)

apply (*unfold less-multiset-def mult-def*)
apply (*erule trancl-induct*)
apply (*blast intro*: *mult1-union transI order-less-trans r-into-trancl*)
apply (*blast intro*: *mult1-union transI order-less-trans r-into-trancl trancl-trans*)
done

lemma *union-less-mono1*: $B < D \Longrightarrow B + C < D + (C::'a::order\ multiset)$
apply (*subst union-commute* [*of B C*])
apply (*subst union-commute* [*of D C*])
apply (*erule union-less-mono2*)
done

lemma *union-less-mono*:
$A < C \Longrightarrow B < D \Longrightarrow A + B < C + (D::'a::order\ multiset)$
apply (*blast intro*!: *union-less-mono1 union-less-mono2 mult-less-trans*)
done

lemma *union-le-mono*:
$A <= C \Longrightarrow B <= D \Longrightarrow A + B <= C + (D::'a::order\ multiset)$
apply (*unfold le-multiset-def*)
apply (*blast intro*: *union-less-mono union-less-mono1 union-less-mono2*)
done

lemma *empty-leI* [*iff*]: $\{\#\} <= (M::'a::order\ multiset)$
apply (*unfold le-multiset-def less-multiset-def*)
apply (*case-tac* $M = \{\#\}$)
prefer *2*
apply (*subgoal-tac* $(\{\#\} + \{\#\}, \{\#\} + M) \in mult\ (Collect\ (split\ op <)))$
prefer *2*
apply (*rule one-step-implies-mult*)
apply (*simp only*: *trans-def*, *auto*)
done

lemma *union-upper1*: $A <= A + (B::'a::order\ multiset)$
proof −
have $A + \{\#\} <= A + B$ **by** (*blast intro*: *union-le-mono*)
thus *?thesis* **by** *simp*
qed

lemma *union-upper2*: $B <= A + (B::'a::order\ multiset)$
by (*subst union-commute*, *rule union-upper1*)

8.5 Link with lists

consts
multiset-of :: *'a list* ⇒ *'a multiset*
primrec
multiset-of [] = $\{\#\}$
multiset-of (*a # x*) = *multiset-of x* + $\{\#\ a\ \#\}$

lemma *multiset-of-zero-iff*[*simp*]: (*multiset-of x* = {#}) = (*x* = [])
by (*induct-tac x*, *auto*)

lemma *multiset-of-zero-iff-right*[*simp*]: ({#} = *multiset-of x*) = (*x* = [])
by (*induct-tac x*, *auto*)

lemma *set-of-multiset-of*[*simp*]: *set-of*(*multiset-of x*) = *set x*
by (*induct-tac x*, *auto*)

lemma *mem-set-multiset-eq*: *x* ∈ *set xs* = (*x* :# *multiset-of xs*)
by (*induct xs*) *auto*

lemma *multiset-of-append*[*simp*]:
multiset-of (*xs* @ *ys*) = *multiset-of xs* + *multiset-of ys*
by (*rule-tac x*=*ys* **in** *spec*, *induct-tac xs*, *auto simp*: *union-ac*)

lemma *surj-multiset-of*: *surj multiset-of*
apply (*unfold surj-def*, *rule allI*)
apply (*rule-tac M*=*y* **in** *multiset-induct*, *auto*)
apply (*rule-tac x* = *x* # *xa* **in** *exI*, *auto*)
done

lemma *set-count-greater-0*: *set x* = {*a*. *0* < *count* (*multiset-of x*) *a*}
by (*induct-tac x*, *auto*)

lemma *distinct-count-atmost-1*:
distinct x = (! *a*. *count* (*multiset-of x*) *a* = (*if a* ∈ *set x then 1 else 0*))
apply (*induct-tac x*, *simp*, *rule iffI*, *simp-all*)
apply (*rule conjI*)
apply (*simp-all add*: *set-of-multiset-of* [*THEN sym*] *del*: *set-of-multiset-of*)
apply (*erule-tac x*=*a* **in** *allE*, *simp*, *clarify*)
apply (*erule-tac x*=*aa* **in** *allE*, *simp*)
done

lemma *multiset-of-eq-setD*:
multiset-of xs = *multiset-of ys* ⟹ *set xs* = *set ys*
by (*rule*) (*auto simp add*:*multiset-eq-conv-count-eq set-count-greater-0*)

lemma *set-eq-iff-multiset-of-eq-distinct*:
⟦*distinct x*; *distinct y*⟧
⟹ (*set x* = *set y*) = (*multiset-of x* = *multiset-of y*)
by (*auto simp*: *multiset-eq-conv-count-eq distinct-count-atmost-1*)

lemma *set-eq-iff-multiset-of-remdups-eq*:
(*set x* = *set y*) = (*multiset-of* (*remdups x*) = *multiset-of* (*remdups y*))
apply (*rule iffI*)
apply (*simp add*: *set-eq-iff-multiset-of-eq-distinct*[*THEN iffD1*])
apply (*drule distinct-remdups*[*THEN distinct-remdups*

[*THEN set-eq-iff-multiset-of-eq-distinct*[*THEN iffD2*]]])
apply *simp*
done

lemma *multiset-of-compl-union*[*simp*]:
multiset-of [*x*∈*xs*. *P x*] + *multiset-of* [*x*∈*xs*. ¬*P x*] = *multiset-of xs*
by (*induct xs*) (*auto simp*: *union-ac*)

lemma *count-filter*:
count (*multiset-of xs*) *x* = *length* [*y* ∈ *xs*. *y* = *x*]
by (*induct xs*, *auto*)

8.6 Pointwise ordering induced by count

consts
mset-le :: ['*a multiset*, '*a multiset*] ⇒ *bool*

syntax
-*mset-le* :: '*a multiset* ⇒ '*a multiset* ⇒ *bool* (- ≤# - [*50*,*51*] *50*)
translations
x ≤# *y* == *mset-le x y*

defs
mset-le-def: *xs* ≤# *ys* == (∀ *a*. *count xs a* ≤ *count ys a*)

lemma *mset-le-refl*[*simp*]: *xs* ≤# *xs*
by (*unfold mset-le-def*) *auto*

lemma *mset-le-trans*: ⟦ *xs* ≤# *ys*; *ys* ≤# *zs* ⟧ ⟹ *xs* ≤# *zs*
by (*unfold mset-le-def*) (*fast intro*: *order-trans*)

lemma *mset-le-antisym*: ⟦ *xs*≤# *ys*; *ys* ≤# *xs*⟧ ⟹ *xs* = *ys*
apply (*unfold mset-le-def*)
apply (*rule multiset-eq-conv-count-eq*[*THEN iffD2*])
apply (*blast intro*: *order-antisym*)
done

lemma *mset-le-exists-conv*:
(*xs* ≤# *ys*) = (∃ *zs*. *ys* = *xs* + *zs*)
apply (*unfold mset-le-def*, *rule iffI*, *rule-tac x* = *ys* − *xs* **in** *exI*)
apply (*auto intro*: *multiset-eq-conv-count-eq* [*THEN iffD2*])
done

lemma *mset-le-mono-add-right-cancel*[*simp*]: (*xs* + *zs* ≤# *ys* + *zs*) = (*xs* ≤# *ys*)
by (*unfold mset-le-def*) *auto*

lemma *mset-le-mono-add-left-cancel*[*simp*]: (*zs* + *xs* ≤# *zs* + *ys*) = (*xs* ≤# *ys*)
by (*unfold mset-le-def*) *auto*

lemma *mset-le-mono-add*: ⟦ *xs* ≤# *ys*; *vs* ≤# *ws* ⟧ ⟹ *xs* + *vs* ≤# *ys* + *ws*
apply (*unfold mset-le-def*)
apply *auto*
apply (*erule-tac x*=*a* **in** *allE*)+
apply *auto*
done

lemma *mset-le-add-left*[*simp*]: *xs* ≤# *xs* + *ys*
by (*unfold mset-le-def*) *auto*

lemma *mset-le-add-right*[*simp*]: *ys* ≤# *xs* + *ys*
by (*unfold mset-le-def*) *auto*

lemma *multiset-of-remdups-le*: *multiset-of* (*remdups x*) ≤# *multiset-of x*
apply (*induct x*)
apply *auto*
apply (*rule mset-le-trans*)
apply *auto*
done

end

9 NatPair: Pairs of Natural Numbers

theory *NatPair*
imports *Main*
begin

An injective function from $\mathbb{N}^2$ to $\mathbb{N}$. Definition and proofs are from [3, page 85].

constdefs
nat2-to-nat:: (*nat* ∗ *nat*) ⇒ *nat*
nat2-to-nat pair ≡ *let* (*n*,*m*) = *pair in* (*n*+*m*) ∗ *Suc* (*n*+*m*) *div* 2 + *n*

lemma *dvd2-a-x-suc-a*: 2 *dvd a* ∗ (*Suc a*)
proof (*cases* 2 *dvd a*)
case *True*
thus *?thesis* **by** (*rule dvd-mult2*)
next
case *False*
hence *Suc* (*a mod* 2) = 2 **by** (*simp add*: *dvd-eq-mod-eq-0*)
hence *Suc a mod* 2 = 0 **by** (*simp add*: *mod-Suc*)
hence 2 *dvd Suc a* **by** (*simp only*:*dvd-eq-mod-eq-0*)
thus *?thesis* **by** (*rule dvd-mult*)
qed

lemma
assumes *eq*: *nat2-to-nat* (*u*,*v*) = *nat2-to-nat* (*x*,*y*)

```
  shows nat2-to-nat-help: u+v ≤ x+y
proof (rule classical)
  assume ¬ ?thesis
  hence contrapos: x+y < u+v
    by simp
  have nat2-to-nat (x,y) < (x+y) * Suc (x+y) div 2 + Suc (x + y)
    by (unfold nat2-to-nat-def) (simp add: Let-def)
  also have ... = (x+y)*Suc(x+y) div 2 + 2 * Suc(x+y) div 2
    by (simp only: div-mult-self1-is-m)
  also have ... = (x+y)*Suc(x+y) div 2 + 2 * Suc(x+y) div 2
    + ((x+y)*Suc(x+y) mod 2 + 2 * Suc(x+y) mod 2) div 2
  proof −
    have 2 dvd (x+y)*Suc(x+y)
      by (rule dvd2-a-x-suc-a)
    hence (x+y)*Suc(x+y) mod 2 = 0
      by (simp only: dvd-eq-mod-eq-0)
    also
    have 2 * Suc(x+y) mod 2 = 0
      by (rule mod-mult-self1-is-0)
    ultimately have
      ((x+y)*Suc(x+y) mod 2 + 2 * Suc(x+y) mod 2) div 2 = 0
      by simp
    thus ?thesis
      by simp
  qed
  also have ... = ((x+y)*Suc(x+y) + 2*Suc(x+y)) div 2
    by (rule div-add1-eq [symmetric])
  also have ... = ((x+y+2)*Suc(x+y)) div 2
    by (simp only: add-mult-distrib [symmetric])
  also from contrapos have ... ≤ ((Suc(u+v))*(u+v)) div 2
    by (simp only: mult-le-mono div-le-mono)
  also have ... ≤ nat2-to-nat (u,v)
    by (unfold nat2-to-nat-def) (simp add: Let-def)
  finally show ?thesis
    by (simp only: eq)
qed

theorem nat2-to-nat-inj: inj nat2-to-nat
proof −
  {
    fix u v x y assume nat2-to-nat (u,v) = nat2-to-nat (x,y)
    hence u+v ≤ x+y by (rule nat2-to-nat-help)
    also from prems [symmetric] have x+y ≤ u+v
      by (rule nat2-to-nat-help)
    finally have eq: u+v = x+y .
    with prems have ux: u=x
      by (simp add: nat2-to-nat-def Let-def)
    with eq have vy: v=y
      by simp
```

```
    with ux have (u,v) = (x,y)
      by simp
  }
  hence ⋀x y. nat2-to-nat x = nat2-to-nat y ⟹ x=y
    by fast
  thus ?thesis
    by (unfold inj-on-def) simp
qed

end
```

10 Nat-Infinity: Natural numbers with infinity

theory *Nat-Infinity*
imports *Main*
begin

10.1 Definitions

We extend the standard natural numbers by a special value indicating infinity. This includes extending the ordering relations $op <$ and $op \leq$.

datatype *inat* = *Fin nat* | *Infty*

instance *inat* :: {*ord*, *zero*} **..**

consts
  iSuc :: *inat* => *inat*

syntax (*xsymbols*)
  Infty :: *inat* (∞)

syntax (*HTML* **output**)
  Infty :: *inat* (∞)

defs
  Zero-inat-def: *0* == *Fin 0*
  iSuc-def: *iSuc i* == *case i of Fin n* => *Fin* (*Suc n*) | ∞ => ∞
  iless-def: *m* < *n* ==
    case m of Fin m1 => (*case n of Fin n1* => *m1* < *n1* | ∞ => *True*)
    | ∞ => *False*
  ile-def: (*m*::*inat*) ≤ *n* == ¬ (*n* < *m*)

lemmas *inat-defs* = *Zero-inat-def iSuc-def iless-def ile-def*
lemmas *inat-splits* = *inat.split inat.split-asm*

Below is a not quite complete set of theorems. Use the method (*simp add*: *inat-defs split*:*inat-splits*, *arith?*) to prove new theorems or solve arithmetic

subgoals involving *inat* on the fly.

10.2 Constructors

lemma *Fin-0*: *Fin 0 = 0*
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *Infty-ne-i0* [*simp*]: $\infty \neq 0$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *i0-ne-Infty* [*simp*]: $0 \neq \infty$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *iSuc-Fin* [*simp*]: *iSuc* (*Fin n*) = *Fin* (*Suc n*)
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *iSuc-Infty* [*simp*]: *iSuc* $\infty = \infty$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *iSuc-ne-0* [*simp*]: *iSuc n* $\neq$ *0*
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *iSuc-inject* [*simp*]: (*iSuc x* = *iSuc y*) = ($x = y$)
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

10.3 Ordering relations

lemma *Infty-ilessE* [*elim*!]: $\infty <$ *Fin m* ==> *R*
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *iless-linear*: $m < n \vee m = n \vee n <$ (m::*inat*)
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *iless-not-refl* [*simp*]: $\neg n <$ (n::*inat*)
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *iless-trans*: $i < j \Longrightarrow j < k \Longrightarrow i <$ (k::*inat*)
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *iless-not-sym*: $n < m \Longrightarrow \neg m <$ (n::*inat*)
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *Fin-iless-mono* [*simp*]: (*Fin n* < *Fin m*) = ($n < m$)
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *Fin-iless-Infty* [*simp*]: *Fin n* $< \infty$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *Infty-eq* [*simp*]: ($n < \infty$) = ($n \neq \infty$)

by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *i0-eq* [*simp*]: $((0::inat) < n) = (n \neq 0)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *i0-iless-iSuc* [*simp*]: $0 < iSuc\ n$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *not-ilessi0* [*simp*]: $\neg\ n < (0::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *Fin-iless*: $n < Fin\ m \Longrightarrow \exists k.\ n = Fin\ k$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *iSuc-mono* [*simp*]: $(iSuc\ n < iSuc\ m) = (n < m)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *ile-def2*: $(m \leq n) = (m < n \vee m = (n::inat))$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *ile-refl* [*simp*]: $n \leq (n::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *ile-trans*: $i \leq j \Longrightarrow j \leq k \Longrightarrow i \leq (k::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *ile-iless-trans*: $i \leq j \Longrightarrow j < k \Longrightarrow i < (k::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *iless-ile-trans*: $i < j \Longrightarrow j \leq k \Longrightarrow i < (k::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *Infty-ub* [*simp*]: $n \leq \infty$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *i0-lb* [*simp*]: $(0::inat) \leq n$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *Infty-ileE* [*elim*!]: $\infty \leq Fin\ m \Longrightarrow R$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *Fin-ile-mono* [*simp*]: $(Fin\ n \leq Fin\ m) = (n \leq m)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *ilessI1*: $n \leq m \Longrightarrow n \neq m \Longrightarrow n < (m::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *ileI1*: $m < n \Longrightarrow iSuc\ m \leq n$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *Suc-ile-eq*: $(Fin\ (Suc\ m) \leq n) = (Fin\ m < n)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *iSuc-ile-mono* [*simp*]: $(iSuc\ n \leq iSuc\ m) = (n \leq m)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *iless-Suc-eq* [*simp*]: $(Fin\ m < iSuc\ n) = (Fin\ m \leq n)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *not-iSuc-ilei0* [*simp*]: $\neg\ iSuc\ n \leq 0$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *ile-iSuc* [*simp*]: $n \leq iSuc\ n$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *Fin-ile*: $n \leq Fin\ m \Longrightarrow \exists k.\ n = Fin\ k$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith?*)

lemma *chain-incr*: $\forall i.\ \exists j.\ Y\ i < Y\ j \Longrightarrow \exists j.\ Fin\ k < Y\ j$
apply (*induct-tac k*)
apply (*simp* (*no-asm*) *only*: *Fin-0*)
apply (*fast intro*: *ile-iless-trans i0-lb*)
apply (*erule exE*)
apply (*drule spec*)
apply (*erule exE*)
apply (*drule ileI1*)
apply (*rule iSuc-Fin* [*THEN subst*])
apply (*rule exI*)
apply (*erule* (*1*) *ile-iless-trans*)
done

end

11 Nested-Environment: Nested environments

theory *Nested-Environment*
imports *Main*
begin

Consider a partial function $e :: 'a \Rightarrow 'b\ option$; this may be understood as an *environment* mapping indexes $'a$ to optional entry values $'b$ (cf. the basic theory *Map* of Isabelle/HOL). This basic idea is easily generalized to that of a *nested environment*, where entries may be either basic values or again proper environments. Then each entry is accessed by a *path*, i.e. a list

of indexes leading to its position within the structure.

```
datatype ('a, 'b, 'c) env =
    Val 'a
  | Env 'b  'c => ('a, 'b, 'c) env option
```

In the type $('a, 'b, 'c)$ *env* the parameter $'a$ refers to basic values (occurring in terminal positions), type $'b$ to values associated with proper (inner) environments, and type $'c$ with the index type for branching. Note that there is no restriction on any of these types. In particular, arbitrary branching may yield rather large (transfinite) tree structures.

11.1 The lookup operation

Lookup in nested environments works by following a given path of index elements, leading to an optional result (a terminal value or nested environment). A *defined position* within a nested environment is one where *lookup* at its path does not yield *None*.

```
consts
  lookup :: ('a, 'b, 'c) env => 'c list => ('a, 'b, 'c) env option
  lookup-option :: ('a, 'b, 'c) env option => 'c list => ('a, 'b, 'c) env option

primrec (lookup)
  lookup (Val a) xs = (if xs = [] then Some (Val a) else None)
  lookup (Env b es) xs =
    (case xs of
      [] => Some (Env b es)
    | y # ys => lookup-option (es y) ys)
  lookup-option None xs = None
  lookup-option (Some e) xs = lookup e xs

hide const lookup-option
```

The characteristic cases of *lookup* are expressed by the following equalities.

```
theorem lookup-nil: lookup e [] = Some e
  by (cases e) simp-all

theorem lookup-val-cons: lookup (Val a) (x # xs) = None
  by simp

theorem lookup-env-cons:
  lookup (Env b es) (x # xs) =
    (case es x of
      None => None
    | Some e => lookup e xs)
  by (cases es x) simp-all
```

```
lemmas lookup.simps [simp del]
  and lookup-simps [simp] = lookup-nil lookup-val-cons lookup-env-cons

theorem lookup-eq:
  lookup env xs =
    (case xs of
      [] => Some env
    | x # xs =>
      (case env of
        Val a => None
      | Env b es =>
          (case es x of
            None => None
          | Some e => lookup e xs)))
  by (simp split: list.split env.split)
```

Displaced *lookup* operations, relative to a certain base path prefix, may be reduced as follows. There are two cases, depending whether the environment actually extends far enough to follow the base path.

```
theorem lookup-append-none:
  !!env. lookup env xs = None ==> lookup env (xs @ ys) = None
  (is PROP ?P xs)
proof (induct xs)
  fix env :: ('a, 'b, 'c) env
  {
    assume lookup env [] = None
    hence False by simp
    thus lookup env ([] @ ys) = None ..
  next
    fix x xs
    assume hyp: PROP ?P xs
    assume asm: lookup env (x # xs) = None
    show lookup env ((x # xs) @ ys) = None
    proof (cases env)
      case Val
      thus ?thesis by simp
    next
      fix b es assume env: env = Env b es
      show ?thesis
      proof (cases es x)
        assume es x = None
        with env show ?thesis by simp
      next
        fix e assume es: es x = Some e
        show ?thesis
        proof (cases lookup e xs)
          case None
          hence lookup e (xs @ ys) = None by (rule hyp)
          with env es show ?thesis by simp
```

```
      next
        case Some
        with asm env es have False by simp
        thus ?thesis ..
      qed
    qed
  qed
 }
qed

theorem lookup-append-some:
 !!env e. lookup env xs = Some e ==> lookup env (xs @ ys) = lookup e ys
 (is PROP ?P xs)
proof (induct xs)
 fix env e :: ('a, 'b, 'c) env
 {
  assume lookup env [] = Some e
  hence env = e by simp
  thus lookup env ([] @ ys) = lookup e ys by simp
 next
  fix x xs
  assume hyp: PROP ?P xs
  assume asm: lookup env (x # xs) = Some e
  show lookup env ((x # xs) @ ys) = lookup e ys
  proof (cases env)
   fix a assume env = Val a
   with asm have False by simp
   thus ?thesis ..
  next
   fix b es assume env: env = Env b es
   show ?thesis
   proof (cases es x)
    assume es x = None
    with asm env have False by simp
    thus ?thesis ..
   next
    fix e' assume es: es x = Some e'
    show ?thesis
    proof (cases lookup e' xs)
     case None
     with asm env es have False by simp
     thus ?thesis ..
    next
     case Some
     with asm env es have lookup e' xs = Some e
      by simp
     hence lookup e' (xs @ ys) = lookup e ys by (rule hyp)
     with env es show ?thesis by simp
    qed
```

```
    qed
  qed
 }
qed
```

Successful *lookup* deeper down an environment structure means we are able to peek further up as well. Note that this is basically just the contrapositive statement of *lookup-append-none* above.

```
theorem lookup-some-append:
  lookup env (xs @ ys) = Some e ==> ∃ e. lookup env xs = Some e
proof −
  assume lookup env (xs @ ys) = Some e
  hence lookup env (xs @ ys) ≠ None by simp
  hence lookup env xs ≠ None
    by (rule contrapos-nn) (simp only: lookup-append-none)
  thus ?thesis by simp
qed
```

The subsequent statement describes in more detail how a successful *lookup* with a non-empty path results in a certain situation at any upper position.

```
theorem lookup-some-upper: !!env e.
  lookup env (xs @ y # ys) = Some e ==>
    ∃ b' es' env'.
      lookup env xs = Some (Env b' es') ∧
      es' y = Some env' ∧
      lookup env' ys = Some e
  (is PROP ?P xs is !!env e. ?A env e xs ==> ?C env e xs)
proof (induct xs)
  fix env e let ?A = ?A env e and ?C = ?C env e
  {
    assume ?A []
    hence lookup env (y # ys) = Some e by simp
    then obtain b' es' env' where
        env: env = Env b' es' and
        es': es' y = Some env' and
        look': lookup env' ys = Some e
      by (auto simp add: lookup-eq split: option.splits env.splits)
    from env have lookup env [] = Some (Env b' es') by simp
    with es' look' show ?C [] by blast
  next
    fix x xs
    assume hyp: PROP ?P xs
    assume ?A (x # xs)
    then obtain b' es' env' where
        env: env = Env b' es' and
        es': es' x = Some env' and
        look': lookup env' (xs @ y # ys) = Some e
      by (auto simp add: lookup-eq split: option.splits env.splits)
```

```
    from hyp [OF look'] obtain b'' es'' env'' where
      upper': lookup env' xs = Some (Env b'' es'') and
      es'': es'' y = Some env'' and
      look'': lookup env'' ys = Some e
     by blast
    from env es' upper' have lookup env (x # xs) = Some (Env b'' es'')
     by simp
    with es'' look'' show ?C (x # xs) by blast
  }
qed
```

11.2 The update operation

Update at a certain position in a nested environment may either delete an existing entry, or overwrite an existing one. Note that update at undefined positions is simple absorbed, i.e. the environment is left unchanged.

```
consts
  update :: 'c list => ('a, 'b, 'c) env option
    => ('a, 'b, 'c) env => ('a, 'b, 'c) env
  update-option :: 'c list => ('a, 'b, 'c) env option
    => ('a, 'b, 'c) env option => ('a, 'b, 'c) env option

primrec (update)
  update xs opt (Val a) =
    (if xs = [] then (case opt of None => Val a | Some e => e)
    else Val a)
  update xs opt (Env b es) =
    (case xs of
      [] => (case opt of None => Env b es | Some e => e)
    | y # ys => Env b (es (y := update-option ys opt (es y))))
  update-option xs opt None =
    (if xs = [] then opt else None)
  update-option xs opt (Some e) =
    (if xs = [] then opt else Some (update xs opt e))

hide const update-option
```

The characteristic cases of *update* are expressed by the following equalities.

```
theorem update-nil-none: update [] None env = env
  by (cases env) simp-all

theorem update-nil-some: update [] (Some e) env = e
  by (cases env) simp-all

theorem update-cons-val: update (x # xs) opt (Val a) = Val a
  by simp

theorem update-cons-nil-env:
```

```
    update [x] opt (Env b es) = Env b (es (x := opt))
  by (cases es x) simp-all

theorem update-cons-cons-env:
  update (x # y # ys) opt (Env b es) =
    Env b (es (x :=
      (case es x of
        None => None
      | Some e => Some (update (y # ys) opt e))))
  by (cases es x) simp-all

lemmas update.simps [simp del]
  and update-simps [simp] = update-nil-none update-nil-some
    update-cons-val update-cons-nil-env update-cons-cons-env

lemma update-eq:
  update xs opt env =
    (case xs of
      [] =>
        (case opt of
          None => env
        | Some e => e)
    | x # xs =>
        (case env of
          Val a => Val a
        | Env b es =>
            (case xs of
              [] => Env b (es (x := opt))
            | y # ys =>
                Env b (es (x :=
                  (case es x of
                    None => None
                  | Some e => Some (update (y # ys) opt e)))))))
  by (simp split: list.split env.split option.split)
```

The most basic correspondence of *lookup* and *update* states that after *update* at a defined position, subsequent *lookup* operations would yield the new value.

```
theorem lookup-update-some:
  !!env e. lookup env xs = Some e ==>
    lookup (update xs (Some env') env) xs = Some env'
  (is PROP ?P xs)
proof (induct xs)
  fix env e :: ('a, 'b, 'c) env
  {
    assume lookup env [] = Some e
    hence env = e by simp
    thus lookup (update [] (Some env') env) [] = Some env'
      by simp
```

```
  next
    fix x xs
    assume hyp: PROP ?P xs
    assume asm: lookup env (x # xs) = Some e
    show lookup (update (x # xs) (Some env') env) (x # xs) = Some env'
    proof (cases env)
      fix a assume env = Val a
      with asm have False by simp
      thus ?thesis ..
    next
      fix b es assume env: env = Env b es
      show ?thesis
      proof (cases es x)
        assume es x = None
        with asm env have False by simp
        thus ?thesis ..
      next
        fix e' assume es: es x = Some e'
        show ?thesis
        proof (cases xs)
          assume xs = []
          with env show ?thesis by simp
        next
          fix x' xs' assume xs: xs = x' # xs'
          from asm env es have lookup e' xs = Some e by simp
          hence lookup (update xs (Some env') e') xs = Some env' by (rule hyp)
          with env es xs show ?thesis by simp
        qed
      qed
    qed
  }
qed
```

The properties of displaced *update* operations are analogous to those of *lookup* above. There are two cases: below an undefined position *update* is absorbed altogether, and below a defined positions *update* affects subsequent *lookup* operations in the obvious way.

```
theorem update-append-none:
  !!env. lookup env xs = None ==> update (xs @ y # ys) opt env = env
  (is PROP ?P xs)
proof (induct xs)
  fix env :: ('a, 'b, 'c) env
  {
    assume lookup env [] = None
    hence False by simp
    thus update ([] @ y # ys) opt env = env ..
  next
    fix x xs
    assume hyp: PROP ?P xs
```

```
    assume asm: lookup env (x # xs) = None
    show update ((x # xs) @ y # ys) opt env = env
    proof (cases env)
      fix a assume env = Val a
      thus ?thesis by simp
    next
      fix b es assume env: env = Env b es
      show ?thesis
      proof (cases es x)
        assume es: es x = None
        show ?thesis
          by (cases xs) (simp-all add: es env fun-upd-idem-iff)
      next
        fix e assume es: es x = Some e
        show ?thesis
        proof (cases xs)
          assume xs = []
          with asm env es have False by simp
          thus ?thesis ..
        next
          fix x' xs' assume xs: xs = x' # xs'
          from asm env es have lookup e xs = None by simp
          hence update (xs @ y # ys) opt e = e by (rule hyp)
          with env es xs show update ((x # xs) @ y # ys) opt env = env
            by (simp add: fun-upd-idem-iff)
        qed
      qed
    qed
  }
qed

theorem update-append-some:
  !!env e. lookup env xs = Some e ==>
    lookup (update (xs @ y # ys) opt env) xs = Some (update (y # ys) opt e)
  (is PROP ?P xs)
proof (induct xs)
  fix env e :: ('a, 'b, 'c) env
  {
    assume lookup env [] = Some e
    hence env = e by simp
    thus lookup (update ([] @ y # ys) opt env) [] = Some (update (y # ys) opt e)
      by simp
  next
    fix x xs
    assume hyp: PROP ?P xs
    assume asm: lookup env (x # xs) = Some e
    show lookup (update ((x # xs) @ y # ys) opt env) (x # xs)
      = Some (update (y # ys) opt e)
    proof (cases env)
```

```
    fix a assume env = Val a
    with asm have False by simp
    thus ?thesis ..
  next
    fix b es assume env: env = Env b es
    show ?thesis
    proof (cases es x)
      assume es x = None
      with asm env have False by simp
      thus ?thesis ..
    next
      fix e' assume es: es x = Some e'
      show ?thesis
      proof (cases xs)
        assume xs: xs = []
        from asm env es xs have e = e' by simp
        with env es xs show ?thesis by simp
      next
        fix x' xs' assume xs: xs = x' # xs'
        from asm env es have lookup e' xs = Some e by simp
        hence lookup (update (xs @ y # ys) opt e') xs =
          Some (update (y # ys) opt e) by (rule hyp)
        with env es xs show ?thesis by simp
      qed
    qed
  qed
}
qed
```

Apparently, *update* does not affect the result of subsequent *lookup* operations at independent positions, i.e. in case that the paths for *update* and *lookup* fork at a certain point.

```
theorem lookup-update-other:
  !!env. y ≠ (z::'c) ==> lookup (update (xs @ z # zs) opt env) (xs @ y # ys) =
    lookup env (xs @ y # ys)
  (is PROP ?P xs)
proof (induct xs)
  fix env :: ('a, 'b, 'c) env
  assume neq: y ≠ z
  {
    show lookup (update ([] @ z # zs) opt env) ([] @ y # ys) =
      lookup env ([] @ y # ys)
    proof (cases env)
      case Val
      thus ?thesis by simp
    next
      case Env
      show ?thesis
      proof (cases zs)
```

```
      case Nil
      with neq Env show ?thesis by simp
    next
      case Cons
      with neq Env show ?thesis by simp
    qed
  qed
 next
  fix x xs
  assume hyp: PROP ?P xs
  show lookup (update ((x # xs) @ z # zs) opt env) ((x # xs) @ y # ys) =
    lookup env ((x # xs) @ y # ys)
  proof (cases env)
    case Val
    thus ?thesis by simp
  next
    fix y es assume env: env = Env y es
    show ?thesis
    proof (cases xs)
      assume xs: xs = []
      show ?thesis
      proof (cases es x)
        case None
        with env xs show ?thesis by simp
      next
        case Some
        with hyp env xs and neq show ?thesis by simp
      qed
    next
      fix x' xs' assume xs: xs = x' # xs'
      show ?thesis
      proof (cases es x)
        case None
        with env xs show ?thesis by simp
      next
        case Some
        with hyp env xs neq show ?thesis by simp
      qed
    qed
  qed
 }
qed

end
```

12 Permutation: Permutations

theory *Permutation*
imports *Multiset*
begin

consts
perm :: (*'a list* ∗ *'a list*) *set*

syntax
-perm :: *'a list* => *'a list* => *bool* (- <~~> - [*50*, *50*] *50*)
translations
x <~~> *y* == (*x*, *y*) ∈ *perm*

inductive *perm*
intros
Nil [*intro*!]: [] <~~> []
swap [*intro*!]: *y* # *x* # *l* <~~> *x* # *y* # *l*
Cons [*intro*!]: *xs* <~~> *ys* ==> *z* # *xs* <~~> *z* # *ys*
trans [*intro*]: *xs* <~~> *ys* ==> *ys* <~~> *zs* ==> *xs* <~~> *zs*

lemma *perm-refl* [*iff*]: *l* <~~> *l*
by (*induct l*) *auto*

12.1 Some examples of rule induction on permutations

lemma *xperm-empty-imp-aux*: *xs* <~~> *ys* ==> *xs* = [] --> *ys* = []
— the form of the premise lets the induction bind *xs* and *ys*
apply (*erule perm.induct*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *xperm-empty-imp*: [] <~~> *ys* ==> *ys* = []
using *xperm-empty-imp-aux* **by** *blast*

This more general theorem is easier to understand!

lemma *perm-length*: *xs* <~~> *ys* ==> *length xs* = *length ys*
by (*erule perm.induct*) *simp-all*

lemma *perm-empty-imp*: [] <~~> *xs* ==> *xs* = []
by (*drule perm-length*) *auto*

lemma *perm-sym*: *xs* <~~> *ys* ==> *ys* <~~> *xs*
by (*erule perm.induct*) *auto*

lemma *perm-mem* [*rule-format*]: *xs* <~~> *ys* ==> *x mem xs* --> *x mem ys*
by (*erule perm.induct*) *auto*

12.2 Ways of making new permutations

We can insert the head anywhere in the list.

lemma *perm-append-Cons*: $a \# xs \mathbin{@} ys <\sim\sim> xs \mathbin{@} a \# ys$
by (*induct xs*) *auto*

lemma *perm-append-swap*: $xs \mathbin{@} ys <\sim\sim> ys \mathbin{@} xs$
apply (*induct xs*)
apply *simp-all*
apply (*blast intro*: *perm-append-Cons*)
done

lemma *perm-append-single*: $a \# xs <\sim\sim> xs \mathbin{@} [a]$
by (*rule perm.trans* [*OF* - *perm-append-swap*]) *simp*

lemma *perm-rev*: *rev xs* $<\sim\sim>$ *xs*
apply (*induct xs*)
apply *simp-all*
apply (*blast intro*!: *perm-append-single intro*: *perm-sym*)
done

lemma *perm-append1*: $xs <\sim\sim> ys \Longrightarrow l \mathbin{@} xs <\sim\sim> l \mathbin{@} ys$
by (*induct l*) *auto*

lemma *perm-append2*: $xs <\sim\sim> ys \Longrightarrow xs \mathbin{@} l <\sim\sim> ys \mathbin{@} l$
by (*blast intro*!: *perm-append-swap perm-append1*)

12.3 Further results

lemma *perm-empty* [*iff*]: $([] <\sim\sim> xs) = (xs = [])$
by (*blast intro*: *perm-empty-imp*)

lemma *perm-empty2* [*iff*]: $(xs <\sim\sim> []) = (xs = [])$
apply *auto*
apply (*erule perm-sym* [*THEN perm-empty-imp*])
done

lemma *perm-sing-imp* [*rule-format*]: $ys <\sim\sim> xs \Longrightarrow xs = [y] \dashrightarrow ys = [y]$
by (*erule perm.induct*) *auto*

lemma *perm-sing-eq* [*iff*]: $(ys <\sim\sim> [y]) = (ys = [y])$
by (*blast intro*: *perm-sing-imp*)

lemma *perm-sing-eq2* [*iff*]: $([y] <\sim\sim> ys) = (ys = [y])$
by (*blast dest*: *perm-sym*)

12.4 Removing elements

consts

remove :: *'a => 'a list => 'a list*
primrec
remove x [] = []
remove x (y # ys) = (if x = y then ys else y # remove x ys)

lemma *perm-remove*: *x ∈ set ys ==> ys <~~> x # remove x ys*
by (*induct ys*) *auto*

lemma *remove-commute*: *remove x (remove y l) = remove y (remove x l)*
by (*induct l*) *auto*

lemma *multiset-of-remove*[*simp*]:
multiset-of (remove a x) = multiset-of x - {#a#}
apply (*induct x*)
apply (*auto simp*: *multiset-eq-conv-count-eq*)
done

Congruence rule

lemma *perm-remove-perm*: *xs <~~> ys ==> remove z xs <~~> remove z ys*
by (*erule perm.induct*) *auto*

lemma *remove-hd* [*simp*]: *remove z (z # xs) = xs*
by *auto*

lemma *cons-perm-imp-perm*: *z # xs <~~> z # ys ==> xs <~~> ys*
by (*drule-tac z = z* **in** *perm-remove-perm*) *auto*

lemma *cons-perm-eq* [*iff*]: *(z#xs <~~> z#ys) = (xs <~~> ys)*
by (*blast intro*: *cons-perm-imp-perm*)

lemma *append-perm-imp-perm*: *!!xs ys. zs @ xs <~~> zs @ ys ==> xs <~~> ys*
apply (*induct zs rule*: *rev-induct*)
apply (*simp-all* (*no-asm-use*))
apply *blast*
done

lemma *perm-append1-eq* [*iff*]: *(zs @ xs <~~> zs @ ys) = (xs <~~> ys)*
by (*blast intro*: *append-perm-imp-perm perm-append1*)

lemma *perm-append2-eq* [*iff*]: *(xs @ zs <~~> ys @ zs) = (xs <~~> ys)*
apply (*safe intro!*: *perm-append2*)
apply (*rule append-perm-imp-perm*)
apply (*rule perm-append-swap* [*THEN perm.trans*])
— the previous step helps this *blast* call succeed quickly
apply (*blast intro*: *perm-append-swap*)
done

lemma *multiset-of-eq-perm*: *(multiset-of xs = multiset-of ys) = (xs <~~> ys)*

apply (*rule iffI*)
apply (*erule-tac* [*2*] *perm.induct*, *simp-all add*: *union-ac*)
apply (*erule rev-mp*, *rule-tac x=ys* **in** *spec*)
apply (*induct-tac xs*, *auto*)
apply (*erule-tac x* = *remove a x* **in** *allE*, *drule sym*, *simp*)
apply (*subgoal-tac a* ∈ *set x*)
apply (*drule-tac z=a* **in** *perm.Cons*)
apply (*erule perm.trans*, *rule perm-sym*, *erule perm-remove*)
apply (*drule-tac f=set-of* **in** *arg-cong*, *simp*)
done

lemma *multiset-of-le-perm-append*:
(*multiset-of xs* ≤# *multiset-of ys*) = (∃ *zs*. *xs* @ *zs* <~~> *ys*)
apply (*auto simp*: *multiset-of-eq-perm*[*THEN sym*] *mset-le-exists-conv*)
apply (*insert surj-multiset-of*, *drule surjD*)
apply (*blast intro*: *sym*)+
done

end

13 Primes: Primality on nat

theory *Primes*
imports *Main*
begin

constdefs
coprime :: *nat* => *nat* => *bool*
coprime m n == *gcd* (*m*, *n*) = *1*

prime :: *nat* ⇒ *bool*
prime p == *1* < *p* ∧ (∀ *m*. *m dvd p* −−> *m* = *1* ∨ *m* = *p*)

lemma *two-is-prime*: *prime 2*
apply (*auto simp add*: *prime-def*)
apply (*case-tac m*)
apply (*auto dest*!: *dvd-imp-le*)
done

lemma *prime-imp-relprime*: *prime p* ==> ¬ *p dvd n* ==> *gcd* (*p*, *n*) = *1*
apply (*auto simp add*: *prime-def*)
apply (*drule-tac x* = *gcd* (*p*, *n*) **in** *spec*)
apply *auto*
apply (*insert gcd-dvd2* [*of p n*])
apply *simp*
done

This theorem leads immediately to a proof of the uniqueness of factorization. If p divides a product of primes then it is one of those primes.

lemma *prime-dvd-mult*: *prime p ==> p dvd m * n ==> p dvd m ∨ p dvd n*
by (*blast intro*: *relprime-dvd-mult prime-imp-relprime*)

lemma *prime-dvd-square*: *prime p ==> p dvd m^Suc (Suc 0) ==> p dvd m*
by (*auto dest*: *prime-dvd-mult*)

lemma *prime-dvd-power-two*: *prime p ==> p dvd m^2 ==> p dvd m*
by (*rule prime-dvd-square*) (*simp-all add*: *power2-eq-square*)

end

14 Quotient: Quotient types

theory *Quotient*
imports *Main*
begin

We introduce the notion of quotient types over equivalence relations via axiomatic type classes.

14.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations $\sim :: 'a => 'a => bool$.

axclass *eqv* ⊆ *type*
consts
eqv :: ('*a*::*eqv*) => '*a* => *bool* (**infixl** ∼ *50*)

axclass *equiv* ⊆ *eqv*
equiv-refl [*intro*]: *x* ∼ *x*
equiv-trans [*trans*]: *x* ∼ *y* ==> *y* ∼ *z* ==> *x* ∼ *z*
equiv-sym [*sym*]: *x* ∼ *y* ==> *y* ∼ *x*

lemma *equiv-not-sym* [*sym*]: ¬ (*x* ∼ *y*) ==> ¬ (*y* ∼ (*x*::'*a*::*equiv*))
proof −
assume ¬ (*x* ∼ *y*) **thus** ¬ (*y* ∼ *x*)
by (*rule contrapos-nn*) (*rule equiv-sym*)
qed

lemma *not-equiv-trans1* [*trans*]: ¬ (*x* ∼ *y*) ==> *y* ∼ *z* ==> ¬ (*x* ∼ (*z*::'*a*::*equiv*))
proof −
assume ¬ (*x* ∼ *y*) **and** *yz*: *y* ∼ *z*
show ¬ (*x* ∼ *z*)
proof

assume $x \sim z$
also from *yz* **have** $z \sim y$ **..**
finally have $x \sim y$ **.**
thus *False* **by** *contradiction*
qed
qed

lemma *not-equiv-trans2* [*trans*]: $x \sim y \Longrightarrow \neg\ (y \sim z) \Longrightarrow \neg\ (x \sim (z::'a::equiv))$
proof −
assume $\neg\ (y \sim z)$ **hence** $\neg\ (z \sim y)$ **..**
also assume $x \sim y$ **hence** $y \sim x$ **..**
finally have $\neg\ (z \sim x)$ **. thus** $(\neg\ x \sim z)$ **..**
qed

The quotient type $'a\ quot$ consists of all *equivalence classes* over elements of the base type $'a$.

typedef $'a\ quot = \{\{x.\ a \sim x\} \mid a::'a::eqv.\ True\}$
by *blast*

lemma *quotI* [*intro*]: $\{x.\ a \sim x\} \in quot$
by (*unfold quot-def*) *blast*

lemma *quotE* [*elim*]: $R \in quot \Longrightarrow (!!a.\ R = \{x.\ a \sim x\} \Longrightarrow C) \Longrightarrow C$
by (*unfold quot-def*) *blast*

Abstracted equivalence classes are the canonical representation of elements of a quotient type.

constdefs
$class :: 'a::equiv \Rightarrow 'a\ quot \quad (\lfloor\text{-}\rfloor)$
$\lfloor a \rfloor == Abs\text{-}quot\ \{x.\ a \sim x\}$

theorem *quot-exhaust*: $\exists\, a.\ A = \lfloor a \rfloor$
proof (*cases A*)
fix *R* **assume** *R*: $A = Abs\text{-}quot\ R$
assume $R \in quot$ **hence** $\exists\, a.\ R = \{x.\ a \sim x\}$ **by** *blast*
with *R* **have** $\exists\, a.\ A = Abs\text{-}quot\ \{x.\ a \sim x\}$ **by** *blast*
thus *?thesis* **by** (*unfold class-def*)
qed

lemma *quot-cases* [*cases type*: *quot*]: $(!!a.\ A = \lfloor a \rfloor \Longrightarrow C) \Longrightarrow C$
by (*insert quot-exhaust*) *blast*

14.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

theorem *quot-equality* [*iff?*]: $(\lfloor a \rfloor = \lfloor b \rfloor) = (a \sim b)$
proof

```
  assume eq: ⌊a⌋ = ⌊b⌋
  show a ∼ b
  proof −
    from eq have {x. a ∼ x} = {x. b ∼ x}
      by (simp only: class-def Abs-quot-inject quotI)
    moreover have a ∼ a ..
    ultimately have a ∈ {x. b ∼ x} by blast
    hence b ∼ a by blast
    thus ?thesis ..
  qed
next
  assume ab: a ∼ b
  show ⌊a⌋ = ⌊b⌋
  proof −
    have {x. a ∼ x} = {x. b ∼ x}
    proof (rule Collect-cong)
      fix x show (a ∼ x) = (b ∼ x)
      proof
        from ab have b ∼ a ..
        also assume a ∼ x
        finally show b ∼ x .
      next
        note ab
        also assume b ∼ x
        finally show a ∼ x .
      qed
    qed
    thus ?thesis by (simp only: class-def)
  qed
qed
```

14.3 Picking representing elements

```
constdefs
  pick :: 'a::equiv quot => 'a
  pick A == SOME a. A = ⌊a⌋

theorem pick-equiv [intro]: pick ⌊a⌋ ∼ a
proof (unfold pick-def)
  show (SOME x. ⌊a⌋ = ⌊x⌋) ∼ a
  proof (rule someI2)
    show ⌊a⌋ = ⌊a⌋ ..
    fix x assume ⌊a⌋ = ⌊x⌋
    hence a ∼ x .. thus x ∼ a ..
  qed
qed

theorem pick-inverse [intro]: ⌊pick A⌋ = A
proof (cases A)
```

fix a **assume** a: $A = \lfloor a \rfloor$
hence *pick* $A \sim a$ **by** (*simp only*: *pick-equiv*)
hence $\lfloor$*pick* $A\rfloor = \lfloor a \rfloor$ **..**
with a **show** *?thesis* **by** *simp*
qed

The following rules support canonical function definitions on quotient types (with up to two arguments). Note that the stripped-down version without additional conditions is sufficient most of the time.

theorem *quot-cond-function*:
(!!$X\ Y$. $P\ X\ Y$ ==> $f\ X\ Y == g$ (*pick* X) (*pick* Y)) ==>
(!!$x\ x'\ y\ y'$. $\lfloor x \rfloor = \lfloor x' \rfloor$ ==> $\lfloor y \rfloor = \lfloor y' \rfloor$
==> $P\ \lfloor x \rfloor\ \lfloor y \rfloor$ ==> $P\ \lfloor x' \rfloor\ \lfloor y' \rfloor$ ==> $g\ x\ y = g\ x'\ y'$) ==>
$P\ \lfloor a \rfloor\ \lfloor b \rfloor$ ==> $f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
(**is** *PROP ?eq* ==> *PROP ?cong* ==> - ==> -)
proof −
assume *cong*: *PROP ?cong*
assume *PROP ?eq* **and** $P\ \lfloor a \rfloor\ \lfloor b \rfloor$
hence $f\ \lfloor a \rfloor\ \lfloor b \rfloor = g$ (*pick* $\lfloor a \rfloor$) (*pick* $\lfloor b \rfloor$) **by** (*simp only*:)
also have ... $= g\ a\ b$
proof (*rule cong*)
show $\lfloor$*pick* $\lfloor a \rfloor\rfloor = \lfloor a \rfloor$ **..**
moreover
show $\lfloor$*pick* $\lfloor b \rfloor\rfloor = \lfloor b \rfloor$ **..**
moreover
show $P\ \lfloor a \rfloor\ \lfloor b \rfloor$ **.**
ultimately show $P\ \lfloor$*pick* $\lfloor a \rfloor\rfloor\ \lfloor$*pick* $\lfloor b \rfloor\rfloor$ **by** (*simp only*:)
qed
finally show *?thesis* **.**
qed

theorem *quot-function*:
(!!$X\ Y$. $f\ X\ Y == g$ (*pick* X) (*pick* Y)) ==>
(!!$x\ x'\ y\ y'$. $\lfloor x \rfloor = \lfloor x' \rfloor$ ==> $\lfloor y \rfloor = \lfloor y' \rfloor$ ==> $g\ x\ y = g\ x'\ y'$) ==>
$f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
proof −
case *rule-context* **from** *this TrueI*
show *?thesis* **by** (*rule quot-cond-function*)
qed

theorem *quot-function'*:
(!!$X\ Y$. $f\ X\ Y == g$ (*pick* X) (*pick* Y)) ==>
(!!$x\ x'\ y\ y'$. $x \sim x'$ ==> $y \sim y'$ ==> $g\ x\ y = g\ x'\ y'$) ==>
$f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
by (*rule quot-function*) (*simp only*: *quot-equality*)+

end

15 While-Combinator: A general "while" combinator

theory *While-Combinator*
imports *Main*
begin

We define a while-combinator *while* and prove: (a) an unrestricted unfolding law (even if while diverges!) (I got this idea from Wolfgang Goerigk), and (b) the invariant rule for reasoning about *while*.

consts *while-aux* :: $('a \Rightarrow bool) \times ('a \Rightarrow 'a) \times 'a \Rightarrow 'a$
recdef (**permissive**) *while-aux*
  same-fst $(\lambda b.\ True)$ $(\lambda b.\ same\text{-}fst\ (\lambda c.\ True)\ (\lambda c.$
    $\{(t, s).\ b\ s \wedge c\ s = t \wedge$
      $\neg\ (\exists f.\ f\ (0::nat) = s \wedge (\forall i.\ b\ (f\ i) \wedge c\ (f\ i) = f\ (i + 1)))\})$
  while-aux $(b, c, s) =$
    (*if* $(\exists f.\ f\ (0::nat) = s \wedge (\forall i.\ b\ (f\ i) \wedge c\ (f\ i) = f\ (i + 1)))$
      then arbitrary
      else if b s then while-aux $(b, c, c\ s)$
      else s)

recdef-tc *while-aux-tc*: *while-aux*
  apply (*rule wf-same-fst*)
  apply (*rule wf-same-fst*)
  apply (*simp add*: *wf-iff-no-infinite-down-chain*)
  apply *blast*
  done

constdefs
  while :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$
  while b c s == *while-aux* (b, c, s)

lemma *while-aux-unfold*:
  while-aux $(b, c, s) =$
    (*if* $\exists f.\ f\ (0::nat) = s \wedge (\forall i.\ b\ (f\ i) \wedge c\ (f\ i) = f\ (i + 1))$
      then arbitrary
      else if b s then while-aux $(b, c, c\ s)$
      else s)
  apply (*rule while-aux-tc* [*THEN while-aux.simps* [*THEN trans*]])
  apply (*rule refl*)
  done

The recursion equation for *while*: directly executable!

theorem *while-unfold* [*code*]:
    while b c s = (*if b s then while b c* (*c s*) *else s*)
  apply (*unfold while-def*)
  apply (*rule while-aux-unfold* [*THEN trans*])
  apply *auto*

apply (*subst while-aux-unfold*)
apply *simp*
apply *clarify*
apply (*erule-tac* $x = \lambda i.\ f\ (Suc\ i)$ **in** *allE*)
apply *blast*
done

hide *const while-aux*

lemma *def-while-unfold*: **assumes** *fdef*: $f == while\ test\ do$
shows $f\ x = (if\ test\ x\ then\ f(do\ x)\ else\ x)$
proof −
have $f\ x = while\ test\ do\ x$ **using** *fdef* **by** *simp*
also have $\ldots = (if\ test\ x\ then\ while\ test\ do\ (do\ x)\ else\ x)$
by(*rule while-unfold*)
also have $\ldots = (if\ test\ x\ then\ f(do\ x)\ else\ x)$ **by**(*simp add*:*fdef*[*symmetric*])
finally show *?thesis* **.**
qed

The proof rule for *while*, where P is the invariant.

theorem *while-rule-lemma*[*rule-format*]:
$[|\ !!s.\ P\ s\ ==>\ b\ s\ ==>\ P\ (c\ s);$
$!!s.\ P\ s\ ==>\ \neg\ b\ s\ ==>\ Q\ s;$
$wf\ \{(t,\ s).\ P\ s\ \wedge\ b\ s\ \wedge\ t = c\ s\}\ |]\ ==>$
$P\ s\ -->\ Q\ (while\ b\ c\ s)$
proof −
case *rule-context*
assume *wf*: $wf\ \{(t,\ s).\ P\ s\ \wedge\ b\ s\ \wedge\ t = c\ s\}$
show *?thesis*
apply (*induct s rule*: *wf* [*THEN wf-induct*])
apply *simp*
apply *clarify*
apply (*subst while-unfold*)
apply (*simp add*: *rule-context*)
done
qed

theorem *while-rule*:
$[|\ P\ s;$
$!!s.\ [|\ P\ s;\ b\ s\ \ |]\ ==>\ P\ (c\ s);$
$!!s.\ [|\ P\ s;\ \neg\ b\ s\ \ |]\ ==>\ Q\ s;$
$wf\ r;$
$!!s.\ [|\ P\ s;\ b\ s\ \ |]\ ==>\ (c\ s,\ s) \in r\ |]\ ==>$
$Q\ (while\ b\ c\ s)$
apply (*rule while-rule-lemma*)
prefer *4* **apply** *assumption*
apply *blast*
apply *blast*
apply(*erule wf-subset*)

apply *blast*
done

An application: computation of the *lfp* on finite sets via iteration.

theorem *lfp-conv-while*:
[| *mono f*; *finite U*; *f U* = *U* |] ==>
lfp f = *fst* (*while* (λ(*A*, *fA*). *A* ≠ *fA*) (λ(*A*, *fA*). (*fA*, *f fA*)) ({}, *f* {}))
apply (*rule-tac P* = λ(*A*, *B*). (*A* ⊆ *U* ∧ *B* = *f A* ∧ *A* ⊆ *B* ∧ *B* ⊆ *lfp f*) **and**
r = ((*Pow U* × *UNIV*) × (*Pow U* × *UNIV*)) ∩
inv-image finite-psubset (*op* − *U o fst*) **in** *while-rule*)
apply (*subst lfp-unfold*)
apply *assumption*
apply (*simp add*: *monoD*)
apply (*subst lfp-unfold*)
apply *assumption*
apply *clarsimp*
apply (*blast dest*: *monoD*)
apply (*fastsimp intro*!: *lfp-lowerbound*)
apply (*blast intro*: *wf-finite-psubset Int-lower2* [*THEN* [*2*] *wf-subset*])
apply (*clarsimp simp add*: *inv-image-def finite-psubset-def order-less-le*)
apply (*blast intro*!: *finite-Diff dest*: *monoD*)
done

An example of using the *while* combinator.

Cannot use *set-eq-subset* because it leads to looping because the antisymmetry simproc turns the subset relationship back into equality.

lemma *seteq*: (*A* = *B*) = ((!*a* : *A*. *a*:*B*) & (!*b*:*B*. *b*:*A*))
by *blast*

theorem *P* (*lfp* (λ*N*::*int set*. {*0*} ∪ {(*n* + *2*) *mod 6* | *n*. *n* ∈ *N*})) =
P {*0*, *4*, *2*}
proof −
have *aux*: !!*f A B*. {*f n* | *n*. *A n* ∨ *B n*} = {*f n* | *n*. *A n*} ∪ {*f n* | *n*. *B n*}
apply *blast*
done
show *?thesis*
apply (*subst lfp-conv-while* [**where** *?U* = {*0*, *1*, *2*, *3*, *4*, *5*}])
apply (*rule monoI*)
apply *blast*
apply *simp*
apply (*simp add*: *aux set-eq-subset*)

The fixpoint computation is performed purely by rewriting:

apply (*simp add*: *while-unfold aux seteq del*: *subset-empty*)
done
qed

end

16 Word: Binary Words

theory *Word*
imports *Main*
uses *word-setup.ML*
begin

16.1 Auxilary Lemmas

lemma *max-le* [*intro*!]: [| $x \leq z$; $y \leq z$ |] ==> $max\ x\ y \leq z$
  by (*simp add*: *max-def*)

lemma *max-mono*:
  fixes x :: *'a*::*linorder*
  assumes *mf*: *mono f*
  shows $max\ (f\ x)\ (f\ y) \leq f\ (max\ x\ y)$
proof −
  from *mf* **and** *le-maxI1* [*of x y*]
  have *fx*: $f\ x \leq f\ (max\ x\ y)$
    by (*rule monoD*)
  from *mf* **and** *le-maxI2* [*of y x*]
  have *fy*: $f\ y \leq f\ (max\ x\ y)$
    by (*rule monoD*)
  from *fx* **and** *fy*
  show $max\ (f\ x)\ (f\ y) \leq f\ (max\ x\ y)$
    by *auto*
qed

declare *zero-le-power* [*intro*]
  and *zero-less-power* [*intro*]

lemma *int-nat-two-exp*: *2* ^ *k* = *int* (*2* ^ *k*)
  by (*simp add*: *zpower-int* [*symmetric*])

16.2 Bits

datatype *bit*
  = *Zero* (**0**)
  | *One* (**1**)

consts
  bitval :: *bit* => *nat*

primrec
  bitval **0** = *0*
  bitval **1** = *1*

consts
bitnot :: *bit* => *bit*
bitand :: *bit* => *bit* => *bit* (**infixr** *bitand 35*)
bitor :: *bit* => *bit* => *bit* (**infixr** *bitor 30*)
bitxor :: *bit* => *bit* => *bit* (**infixr** *bitxor 30*)

syntax (*xsymbols*)
bitnot :: *bit* => *bit* ($\neg_b$ - [*40*] *40*)
bitand :: *bit* => *bit* => *bit* (**infixr** $\wedge_b$ *35*)
bitor :: *bit* => *bit* => *bit* (**infixr** $\vee_b$ *30*)
bitxor :: *bit* => *bit* => *bit* (**infixr** $\oplus_b$ *30*)

syntax (*HTML* **output**)
bitnot :: *bit* => *bit* ($\neg_b$ - [*40*] *40*)
bitand :: *bit* => *bit* => *bit* (**infixr** $\wedge_b$ *35*)
bitor :: *bit* => *bit* => *bit* (**infixr** $\vee_b$ *30*)
bitxor :: *bit* => *bit* => *bit* (**infixr** $\oplus_b$ *30*)

primrec
bitnot-zero: (*bitnot* **0**) = **1**
bitnot-one : (*bitnot* **1**) = **0**

primrec
bitand-zero: (**0** *bitand y*) = **0**
bitand-one: (**1** *bitand y*) = *y*

primrec
bitor-zero: (**0** *bitor y*) = *y*
bitor-one: (**1** *bitor y*) = **1**

primrec
bitxor-zero: (**0** *bitxor y*) = *y*
bitxor-one: (**1** *bitxor y*) = (*bitnot y*)

lemma *bitnot-bitnot* [*simp*]: (*bitnot* (*bitnot b*)) = *b*
by (*cases b*,*simp-all*)

lemma *bitand-cancel* [*simp*]: (*b bitand b*) = *b*
by (*cases b*,*simp-all*)

lemma *bitor-cancel* [*simp*]: (*b bitor b*) = *b*
by (*cases b*,*simp-all*)

lemma *bitxor-cancel* [*simp*]: (*b bitxor b*) = **0**
by (*cases b*,*simp-all*)

16.3 Bit Vectors

First, a couple of theorems expressing case analysis and induction principles for bit vectors.

lemma *bit-list-cases*:
  assumes *empty*: $w = [] \Longrightarrow P\ w$
  and *zero*: $!!bs.\ w = \mathbf{0} \# bs \Longrightarrow P\ w$
  and *one*: $!!bs.\ w = \mathbf{1} \# bs \Longrightarrow P\ w$
  shows $P\ w$
proof (*cases w*)
  assume $w = []$
  thus *?thesis*
    by (*rule empty*)
next
  fix *b bs*
  assume [*simp*]: $w = b \# bs$
  show $P\ w$
  proof (*cases b*)
    assume $b = \mathbf{0}$
    hence $w = \mathbf{0} \# bs$
      by *simp*
    thus *?thesis*
      by (*rule zero*)
  next
    assume $b = \mathbf{1}$
    hence $w = \mathbf{1} \# bs$
      by *simp*
    thus *?thesis*
      by (*rule one*)
  qed
qed

lemma *bit-list-induct*:
  assumes *empty*: $P\ []$
  and *zero*: $!!bs.\ P\ bs \Longrightarrow P\ (\mathbf{0}\#bs)$
  and *one*: $!!bs.\ P\ bs \Longrightarrow P\ (\mathbf{1}\#bs)$
  shows $P\ w$
proof (*induct w*,*simp-all add*: *empty*)
  fix *b bs*
  assume [*intro*!]: $P\ bs$
  show $P\ (b\#bs)$
    by (*cases b*,*auto intro*!: *zero one*)
qed

constdefs
  bv-msb :: *bit list* => *bit*
  bv-msb w == *if w* = [] *then* **0** *else hd w*
  bv-extend :: [*nat*,*bit*,*bit list*]=>*bit list*
  bv-extend i b w == (*replicate* (*i* − *length w*) *b*) @ *w*

bv-not :: *bit list* => *bit list*
bv-not w == *map bitnot w*

lemma *bv-length-extend* [*simp*]: *length w* ≤ *i* ==> *length* (*bv-extend i b w*) = *i*
by (*simp add*: *bv-extend-def*)

lemma *bv-not-Nil* [*simp*]: *bv-not* [] = []
by (*simp add*: *bv-not-def*)

lemma *bv-not-Cons* [*simp*]: *bv-not* (*b*#*bs*) = (*bitnot b*) # *bv-not bs*
by (*simp add*: *bv-not-def*)

lemma *bv-not-bv-not* [*simp*]: *bv-not* (*bv-not w*) = *w*
by (*rule bit-list-induct* [*of* - *w*],*simp-all*)

lemma *bv-msb-Nil* [*simp*]: *bv-msb* [] = **0**
by (*simp add*: *bv-msb-def*)

lemma *bv-msb-Cons* [*simp*]: *bv-msb* (*b*#*bs*) = *b*
by (*simp add*: *bv-msb-def*)

lemma *bv-msb-bv-not* [*simp*]: *0* < *length w* ==> *bv-msb* (*bv-not w*) = (*bitnot* (*bv-msb w*))
by (*cases w*,*simp-all*)

lemma *bv-msb-one-length* [*simp*,*intro*]: *bv-msb w* = **1** ==> *0* < *length w*
by (*cases w*,*simp-all*)

lemma *length-bv-not* [*simp*]: *length* (*bv-not w*) = *length w*
by (*induct w*,*simp-all*)

constdefs
bv-to-nat :: *bit list* => *nat*
bv-to-nat == *foldl* (%*bn b*. *2* * *bn* + *bitval b*) *0*

lemma *bv-to-nat-Nil* [*simp*]: *bv-to-nat* [] = *0*
by (*simp add*: *bv-to-nat-def*)

lemma *bv-to-nat-helper* [*simp*]: *bv-to-nat* (*b* # *bs*) = *bitval b* * *2* ^ *length bs* + *bv-to-nat bs*
proof −
let *?bv-to-nat'* = *foldl* (λ*bn b*. *2* * *bn* + *bitval b*)
have *helper*: ⋀*base*. *?bv-to-nat' base bs* = *base* * *2* ^ *length bs* + *?bv-to-nat' 0 bs*
proof (*induct bs*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons x xs base*)
show *?case*

```
    apply (simp only: foldl.simps)
    apply (subst Cons [of 2 * base + bitval x])
    apply simp
    apply (subst Cons [of bitval x])
    apply (simp add: add-mult-distrib)
    done
  qed
  show ?thesis by (simp add: bv-to-nat-def) (rule helper)
qed

lemma bv-to-nat0 [simp]: bv-to-nat (0#bs) = bv-to-nat bs
  by simp

lemma bv-to-nat1 [simp]: bv-to-nat (1#bs) = 2 ^ length bs + bv-to-nat bs
  by simp

lemma bv-to-nat-upper-range: bv-to-nat w < 2 ^ length w
proof (induct w,simp-all)
  fix b bs
  assume bv-to-nat bs < 2 ^ length bs
  show bitval b * 2 ^ length bs + bv-to-nat bs < 2 * 2 ^ length bs
  proof (cases b,simp-all)
    have bv-to-nat bs < 2 ^ length bs
      .
    also have ... < 2 * 2 ^ length bs
      by auto
    finally show bv-to-nat bs < 2 * 2 ^ length bs
      by simp
  next
    have bv-to-nat bs < 2 ^ length bs
      .
    hence 2 ^ length bs + bv-to-nat bs < 2 ^ length bs + 2 ^ length bs
      by arith
    also have ... = 2 * (2 ^ length bs)
      by simp
    finally show bv-to-nat bs < 2 ^ length bs
      by simp
  qed
qed

lemma bv-extend-longer [simp]:
  assumes wn: n ≤ length w
  shows       bv-extend n b w = w
  by (simp add: bv-extend-def wn)

lemma bv-extend-shorter [simp]:
  assumes wn: length w < n
  shows       bv-extend n b w = bv-extend n b (b#w)
proof −
```

```
  from wn
  have s: n − Suc (length w) + 1 = n − length w
    by arith
  have bv-extend n b w = replicate (n − length w) b @ w
    by (simp add: bv-extend-def)
  also have ... = replicate (n − Suc (length w) + 1) b @ w
    by (subst s,rule)
  also have ... = (replicate (n − Suc (length w)) b @ replicate 1 b) @ w
    by (subst replicate-add,rule)
  also have ... = replicate (n − Suc (length w)) b @ b # w
    by simp
  also have ... = bv-extend n b (b#w)
    by (simp add: bv-extend-def)
  finally show bv-extend n b w = bv-extend n b (b#w)
    .
qed

consts
  rem-initial :: bit => bit list => bit list

primrec
  rem-initial b [] = []
  rem-initial b (x#xs) = (if b = x then rem-initial b xs else x#xs)

lemma rem-initial-length: length (rem-initial b w) ≤ length w
  by (rule bit-list-induct [of - w],simp-all (no-asm),safe,simp-all)

lemma rem-initial-equal:
  assumes p: length (rem-initial b w) = length w
  shows      rem-initial b w = w
proof −
  have length (rem-initial b w) = length w −−> rem-initial b w = w
  proof (induct w,simp-all,clarify)
    fix xs
    assume length (rem-initial b xs) = length xs −−> rem-initial b xs = xs
    assume f: length (rem-initial b xs) = Suc (length xs)
    with rem-initial-length [of b xs]
    show rem-initial b xs = b#xs
      by auto
  qed
  thus ?thesis
    ..
qed

lemma bv-extend-rem-initial: bv-extend (length w) b (rem-initial b w) = w
proof (induct w,simp-all,safe)
  fix xs
  assume ind: bv-extend (length xs) b (rem-initial b xs) = xs
  from rem-initial-length [of b xs]
```

have [*simp*]: *Suc* (*length xs*) − *length* (*rem-initial b xs*) = *1* + (*length xs* − *length* (*rem-initial b xs*))
by *arith*
have *bv-extend* (*Suc* (*length xs*)) *b* (*rem-initial b xs*) = *replicate* (*Suc* (*length xs*) − *length* (*rem-initial b xs*)) *b* @ (*rem-initial b xs*)
by (*simp add*: *bv-extend-def*)
also have ... = *replicate* (*1* + (*length xs* − *length* (*rem-initial b xs*))) *b* @ *rem-initial b xs*
by *simp*
also have ... = (*replicate 1 b* @ *replicate* (*length xs* − *length* (*rem-initial b xs*)) *b*) @ *rem-initial b xs*
by (*subst replicate-add*,*rule refl*)
also have ... = *b* # *bv-extend* (*length xs*) *b* (*rem-initial b xs*)
by (*auto simp add*: *bv-extend-def* [*symmetric*])
also have ... = *b* # *xs*
by (*simp add*: *ind*)
finally show *bv-extend* (*Suc* (*length xs*)) *b* (*rem-initial b xs*) = *b* # *xs*
.
qed

lemma *rem-initial-append1*:
assumes *rem-initial b xs* ~= []
shows *rem-initial b* (*xs* @ *ys*) = *rem-initial b xs* @ *ys*
proof −
have *rem-initial b xs* ~= [] −−> *rem-initial b* (*xs* @ *ys*) = *rem-initial b xs* @ *ys* (**is** *?P xs ys*)
by (*induct xs*,*auto*)
thus *?thesis*
..
qed

lemma *rem-initial-append2*:
assumes *rem-initial b xs* = []
shows *rem-initial b* (*xs* @ *ys*) = *rem-initial b ys*
proof −
have *rem-initial b xs* = [] −−> *rem-initial b* (*xs* @ *ys*) = *rem-initial b ys* (**is** *?P xs ys*)
by (*induct xs*,*auto*)
thus *?thesis*
..
qed

constdefs
norm-unsigned :: *bit list* => *bit list*
norm-unsigned == *rem-initial* **0**

lemma *norm-unsigned-Nil* [*simp*]: *norm-unsigned* [] = []
by (*simp add*: *norm-unsigned-def*)

lemma *norm-unsigned-Cons0* [*simp*]: *norm-unsigned* (**0**#*bs*) = *norm-unsigned bs*
by (*simp add*: *norm-unsigned-def*)

lemma *norm-unsigned-Cons1* [*simp*]: *norm-unsigned* (**1**#*bs*) = **1**#*bs*
by (*simp add*: *norm-unsigned-def*)

lemma *norm-unsigned-idem* [*simp*]: *norm-unsigned* (*norm-unsigned w*) = *norm-unsigned w*
by (*rule bit-list-induct* [*of* - *w*],*simp-all*)

consts
nat-to-bv-helper :: *nat* => *bit list* => *bit list*

recdef *nat-to-bv-helper measure* (λ*n*. *n*)
nat-to-bv-helper n = (%*bs*. (*if n* = *0 then bs*
else nat-to-bv-helper (*n div 2*) ((*if n mod 2* = *0 then* **0** *else* **1**)#*bs*)))

constdefs
nat-to-bv :: *nat* => *bit list*
nat-to-bv n == *nat-to-bv-helper n* []

lemma *nat-to-bv0* [*simp*]: *nat-to-bv 0* = []
by (*simp add*: *nat-to-bv-def*)

lemmas [*simp del*] = *nat-to-bv-helper.simps*

lemma *n-div-2-cases*:
assumes *zero*: (*n*::*nat*) = *0* ==> *R*
and *div* : [| *n div 2* < *n* ; *0* < *n* |] ==> *R*
shows *R*
proof (*cases n* = *0*)
assume *n* = *0*
thus *R*
by (*rule zero*)
next
assume *n* ~= *0*
hence *nn0*: *0* < *n*
by *simp*
hence *n div 2* < *n*
by *arith*
from *this* **and** *nn0*
show *R*
by (*rule div*)
qed

lemma *int-wf-ge-induct*:
assumes *base*: *P* (*k*::*int*)
and *ind* : !!*i*. (!!*j*. [| *k* ≤ *j* ; *j* < *i* |] ==> *P j*) ==> *P i*

```
  and     valid: k ≤ i
  shows         P i
proof −
  have a: ∀ j. k ≤ j ∧ j < i −−> P j
  proof (rule int-ge-induct)
    show k ≤ i
      .
  next
    show ∀ j. k ≤ j ∧ j < k −−> P j
      by auto
  next
    fix i
    assume k ≤ i
    assume a: ∀ j. k ≤ j ∧ j < i −−> P j
    have pi: P i
    proof (rule ind)
      fix j
      assume k ≤ j and j < i
      with a
      show P j
        by auto
    qed
    show ∀ j. k ≤ j ∧ j < i + 1 −−> P j
    proof auto
      fix j
      assume kj: k ≤ j
      assume ji: j ≤ i
      show P j
      proof (cases j = i)
        assume j = i
        with pi
        show P j
          by simp
      next
        assume j ~= i
        with ji
        have j < i
          by simp
        with kj and a
        show P j
          by blast
      qed
    qed
  qed
  show P i
  proof (rule ind)
    fix j
    assume k ≤ j and j < i
    with a
```

```
    show P j
      by auto
  qed
qed

lemma unfold-nat-to-bv-helper:
  nat-to-bv-helper b l = nat-to-bv-helper b [] @ l
proof −
  have ∀ l. nat-to-bv-helper b l = nat-to-bv-helper b [] @ l
  proof (induct b rule: less-induct)
    fix n
    assume ind: !!j. j < n ⟹ ∀ l. nat-to-bv-helper j l = nat-to-bv-helper j [] @ l
    show ∀ l. nat-to-bv-helper n l = nat-to-bv-helper n [] @ l
    proof
      fix l
      show nat-to-bv-helper n l = nat-to-bv-helper n [] @ l
      proof (cases n < 0)
        assume n < 0
        thus ?thesis
          by (simp add: nat-to-bv-helper.simps)
      next
        assume ~n < 0
        show ?thesis
        proof (rule n-div-2-cases [of n])
          assume [simp]: n = 0
          show ?thesis
            apply (simp only: nat-to-bv-helper.simps [of n])
            apply simp
            done
        next
          assume n2n: n div 2 < n
          assume [simp]: 0 < n
          hence n20: 0 ≤ n div 2
            by arith
          from ind [of n div 2] and n2n n20
          have ind': ∀ l. nat-to-bv-helper (n div 2) l = nat-to-bv-helper (n div 2) []
@ l
            by blast
          show ?thesis
            apply (simp only: nat-to-bv-helper.simps [of n])
            apply (cases n=0)
            apply simp
            apply (simp only: if-False)
            apply simp
            apply (subst spec [OF ind',of 0#l])
            apply (subst spec [OF ind',of 1#l])
            apply (subst spec [OF ind',of [1]])
            apply (subst spec [OF ind',of [0]])
            apply simp
```

```
        done
      qed
    qed
  qed
 qed
 thus ?thesis
  ..
qed

lemma nat-to-bv-non0 [simp]: 0 < n ==> nat-to-bv n = nat-to-bv (n div 2) @
[if n mod 2 = 0 then 0 else 1]
proof −
 assume [simp]: 0 < n
 show ?thesis
  apply (subst nat-to-bv-def [of n])
  apply (simp only: nat-to-bv-helper.simps [of n])
  apply (subst unfold-nat-to-bv-helper)
  using prems
  apply simp
  apply (subst nat-to-bv-def [of n div 2])
  apply auto
  done
qed

lemma bv-to-nat-dist-append: bv-to-nat (l1 @ l2) = bv-to-nat l1 * 2 ^ length l2
+ bv-to-nat l2
proof −
 have ∀ l2. bv-to-nat (l1 @ l2) = bv-to-nat l1 * 2 ^ length l2 + bv-to-nat l2
 proof (induct l1,simp-all)
  fix x xs
  assume ind: ∀ l2. bv-to-nat (xs @ l2) = bv-to-nat xs * 2 ^ length l2 + bv-to-nat
l2
  show ∀ l2. bitval x * 2 ^ (length xs + length l2) + bv-to-nat xs * 2 ^ length l2
= (bitval x * 2 ^ length xs + bv-to-nat xs) * 2 ^ length l2
  proof
   fix l2
   show bitval x * 2 ^ (length xs + length l2) + bv-to-nat xs * 2 ^ length l2 =
(bitval x * 2 ^ length xs + bv-to-nat xs) * 2 ^ length l2
   proof −
    have (2::nat) ^ (length xs + length l2) = 2 ^ length xs * 2 ^ length l2
     by (induct length xs,simp-all)
    hence bitval x * 2 ^ (length xs + length l2) + bv-to-nat xs * 2 ^ length l2 =
     bitval x * 2 ^ length xs * 2 ^ length l2 + bv-to-nat xs * 2 ^ length l2
     by simp
    also have ... = (bitval x * 2 ^ length xs + bv-to-nat xs) * 2 ^ length l2
     by (simp add: ring-distrib)
    finally show ?thesis .
   qed
  qed
```

```
  qed
  thus ?thesis
    ..
qed

lemma bv-nat-bv [simp]: bv-to-nat (nat-to-bv n) = n
proof (induct n rule: less-induct)
  fix n
  assume ind: !!j. j < n ⟹ bv-to-nat (nat-to-bv j) = j
  show bv-to-nat (nat-to-bv n) = n
  proof (rule n-div-2-cases [of n])
    assume [simp]: n = 0
    show ?thesis
      by simp
  next
    assume nn: n div 2 < n
    assume n0: 0 < n
    from ind and nn
    have ind': bv-to-nat (nat-to-bv (n div 2)) = n div 2
      by blast
    from n0 have n0': n ≠ 0
      by simp
    show ?thesis
      apply (subst nat-to-bv-def)
      apply (simp only: nat-to-bv-helper.simps [of n])
      apply (simp only: n0' if-False)
      apply (subst unfold-nat-to-bv-helper)
      apply (subst bv-to-nat-dist-append)
      apply (fold nat-to-bv-def)
      apply (simp add: ind' split del: split-if)
      apply (cases n mod 2 = 0)
      proof simp-all
        assume n mod 2 = 0
        with mod-div-equality [of n 2]
        show n div 2 * 2 = n
          by simp
      next
        assume n mod 2 = Suc 0
        with mod-div-equality [of n 2]
        show Suc (n div 2 * 2) = n
          by simp
      qed
  qed
qed

lemma bv-to-nat-type [simp]: bv-to-nat (norm-unsigned w) = bv-to-nat w
  by (rule bit-list-induct,simp-all)

lemma length-norm-unsigned-le [simp]: length (norm-unsigned w) ≤ length w
```

by (*rule bit-list-induct*,*simp-all*)

lemma *bv-to-nat-rew-msb*: *bv-msb w* = **1** ==> *bv-to-nat w* = *2* ^ (*length w* − *1*) + *bv-to-nat* (*tl w*)
by (*rule bit-list-cases* [*of w*],*simp-all*)

lemma *norm-unsigned-result*: *norm-unsigned xs* = [] ∨ *bv-msb* (*norm-unsigned xs*) = **1**
proof (*rule length-induct* [*of* - *xs*])
fix *xs* :: *bit list*
assume *ind*: ∀ *ys*. *length ys* < *length xs* −−> *norm-unsigned ys* = [] ∨ *bv-msb* (*norm-unsigned ys*) = **1**
show *norm-unsigned xs* = [] ∨ *bv-msb* (*norm-unsigned xs*) = **1**
proof (*rule bit-list-cases* [*of xs*],*simp-all*)
fix *bs*
assume [*simp*]: *xs* = **0**#*bs*
from *ind*
have *length bs* < *length xs* −−> *norm-unsigned bs* = [] ∨ *bv-msb* (*norm-unsigned bs*) = **1**
..
thus *norm-unsigned bs* = [] ∨ *bv-msb* (*norm-unsigned bs*) = **1**
by *simp*
qed
qed

lemma *norm-empty-bv-to-nat-zero*:
assumes *nw*: *norm-unsigned w* = []
shows *bv-to-nat w* = *0*
proof −
have *bv-to-nat w* = *bv-to-nat* (*norm-unsigned w*)
by *simp*
also have ... = *bv-to-nat* []
by (*subst nw*,*rule*)
also have ... = *0*
by *simp*
finally show *?thesis* .
qed

lemma *bv-to-nat-lower-limit*:
assumes *w0*: *0* < *bv-to-nat w*
shows *2* ^ (*length* (*norm-unsigned w*) − *1*) ≤ *bv-to-nat w*
proof −
from *w0* **and** *norm-unsigned-result* [*of w*]
have *msbw*: *bv-msb* (*norm-unsigned w*) = **1**
by (*auto simp add*: *norm-empty-bv-to-nat-zero*)
have *2* ^ (*length* (*norm-unsigned w*) − *1*) ≤ *bv-to-nat* (*norm-unsigned w*)
by (*subst bv-to-nat-rew-msb* [*OF msbw*],*simp*)
thus *?thesis*
by *simp*

qed

lemmas [*simp del*] = *nat-to-bv-non0*

lemma *norm-unsigned-length* [*intro*!]: *length* (*norm-unsigned w*) ≤ *length w*
by (*subst norm-unsigned-def*,*rule rem-initial-length*)

lemma *norm-unsigned-equal*: *length* (*norm-unsigned w*) = *length w* ==> *norm-unsigned w* = *w*
by (*simp add*: *norm-unsigned-def*,*rule rem-initial-equal*)

lemma *bv-extend-norm-unsigned*: *bv-extend* (*length w*) **0** (*norm-unsigned w*) = *w*
by (*simp add*: *norm-unsigned-def*,*rule bv-extend-rem-initial*)

lemma *norm-unsigned-append1* [*simp*]: *norm-unsigned xs* ≠ [] ==> *norm-unsigned* (*xs* @ *ys*) = *norm-unsigned xs* @ *ys*
by (*simp add*: *norm-unsigned-def*,*rule rem-initial-append1*)

lemma *norm-unsigned-append2* [*simp*]: *norm-unsigned xs* = [] ==> *norm-unsigned* (*xs* @ *ys*) = *norm-unsigned ys*
by (*simp add*: *norm-unsigned-def*,*rule rem-initial-append2*)

lemma *bv-to-nat-zero-imp-empty* [*rule-format*]:
bv-to-nat w = *0* ⟶ *norm-unsigned w* = []
by (*rule bit-list-induct* [*of* - *w*],*simp-all*)

lemma *bv-to-nat-nzero-imp-nempty*:
assumes *bv-to-nat w* ≠ *0*
shows *norm-unsigned w* ≠ []
proof −
have *bv-to-nat w* ≠ *0* −−> *norm-unsigned w* ≠ []
by (*rule bit-list-induct* [*of* - *w*],*simp-all*)
thus *?thesis*
..
qed

lemma *nat-helper1*:
assumes *ass*: *nat-to-bv* (*bv-to-nat w*) = *norm-unsigned w*
shows *nat-to-bv* (*2* ∗ *bv-to-nat w* + *bitval x*) = *norm-unsigned* (*w* @ [*x*])
proof (*cases x*)
assume [*simp*]: *x* = **1**
show *?thesis*
apply (*simp add*: *nat-to-bv-non0*)
apply *safe*
proof −
fix *q*
assume *Suc* (*2* ∗ *bv-to-nat w*) = *2* ∗ *q*
hence *orig*: (*2* ∗ *bv-to-nat w* + *1*) *mod 2* = *2* ∗ *q mod 2* (**is** *?lhs* = *?rhs*)
by *simp*

```
    have ?lhs = (1 + 2 * bv-to-nat w) mod 2
      by (simp add: add-commute)
    also have ... = 1
      by (subst mod-add1-eq) simp
    finally have eq1: ?lhs = 1 .
    have ?rhs  = 0
      by simp
    with orig and eq1
   show nat-to-bv (Suc (2 * bv-to-nat w) div 2) @ [0] = norm-unsigned (w @ [1])
      by simp
  next
     have nat-to-bv (Suc (2 * bv-to-nat w) div 2) @ [1] = nat-to-bv ((1 + 2 *
bv-to-nat w) div 2) @ [1]
      by (simp add: add-commute)
    also have ... = nat-to-bv (bv-to-nat w) @ [1]
      by (subst div-add1-eq,simp)
    also have ... = norm-unsigned w @ [1]
      by (subst ass,rule refl)
    also have ... = norm-unsigned (w @ [1])
      by (cases norm-unsigned w,simp-all)
    finally show nat-to-bv (Suc (2 * bv-to-nat w) div 2) @ [1] = norm-unsigned
(w @ [1])
      .
  qed
next
  assume [simp]: x = 0
  show ?thesis
  proof (cases bv-to-nat w = 0)
    assume bv-to-nat w = 0
    thus ?thesis
      by (simp add: bv-to-nat-zero-imp-empty)
  next
    assume bv-to-nat w ≠ 0
    thus ?thesis
      apply simp
      apply (subst nat-to-bv-non0)
      apply simp
      apply auto
      apply (subst ass)
      apply (cases norm-unsigned w)
      apply (simp-all add: norm-empty-bv-to-nat-zero)
      done
  qed
qed

lemma nat-helper2: nat-to-bv (2 ^ length xs + bv-to-nat xs) = 1 # xs
proof −
  have ∀ xs. nat-to-bv (2 ^ length (rev xs) + bv-to-nat (rev xs)) = 1 # (rev xs)
(is ∀ xs. ?P xs)
```

```
proof
  fix xs
  show ?P xs
  proof (rule length-induct [of - xs])
    fix xs :: bit list
    assume ind: ∀ ys. length ys < length xs −−> ?P ys
    show ?P xs
    proof (cases xs)
      assume [simp]: xs = []
      show ?thesis
        by (simp add: nat-to-bv-non0)
    next
      fix y ys
      assume [simp]: xs = y # ys
      show ?thesis
        apply simp
        apply (subst bv-to-nat-dist-append)
        apply simp
      proof −
       have nat-to-bv (2 * 2 ^ length ys + (bv-to-nat (rev ys) * 2 + bitval y)) =
          nat-to-bv (2 * (2 ^ length ys + bv-to-nat (rev ys)) + bitval y)
          by (simp add: add-ac mult-ac)
        also have ... = nat-to-bv (2 * (bv-to-nat (1#rev ys)) + bitval y)
          by simp
        also have ... = norm-unsigned (1#rev ys) @ [y]
        proof −
          from ind
          have nat-to-bv (2 ^ length (rev ys) + bv-to-nat (rev ys)) = 1 # rev ys
            by auto
         hence [simp]: nat-to-bv (2 ^ length ys + bv-to-nat (rev ys)) = 1 # rev ys
            by simp
          show ?thesis
            apply (subst nat-helper1)
            apply simp-all
            done
        qed
        also have ... = (1#rev ys) @ [y]
          by simp
        also have ... = 1 # rev ys @ [y]
          by simp
         finally show nat-to-bv (2 * 2 ^ length ys + (bv-to-nat (rev ys) * 2 +
bitval y)) = 1 # rev ys @ [y]
          .
      qed
    qed
  qed
  hence nat-to-bv (2 ^ length (rev (rev xs)) + bv-to-nat (rev (rev xs))) = 1 # rev
(rev xs)
```

```
    ..
  thus ?thesis
    by simp
qed

lemma nat-bv-nat [simp]: nat-to-bv (bv-to-nat w) = norm-unsigned w
proof (rule bit-list-induct [of - w],simp-all)
  fix xs
  assume nat-to-bv (bv-to-nat xs) = norm-unsigned xs
  have bv-to-nat xs = bv-to-nat (norm-unsigned xs)
    by simp
  have bv-to-nat xs < 2 ^ length xs
    by (rule bv-to-nat-upper-range)
  show nat-to-bv (2 ^ length xs + bv-to-nat xs) = 1 # xs
    by (rule nat-helper2)
qed

lemma bv-to-nat-qinj:
  assumes one: bv-to-nat xs = bv-to-nat ys
  and     len: length xs = length ys
  shows        xs = ys
proof -
  from one
  have nat-to-bv (bv-to-nat xs) = nat-to-bv (bv-to-nat ys)
    by simp
  hence xsys: norm-unsigned xs = norm-unsigned ys
    by simp
  have xs = bv-extend (length xs) 0 (norm-unsigned xs)
    by (simp add: bv-extend-norm-unsigned)
  also have ... = bv-extend (length ys) 0 (norm-unsigned ys)
    by (simp add: xsys len)
  also have ... = ys
    by (simp add: bv-extend-norm-unsigned)
  finally show ?thesis .
qed

lemma norm-unsigned-nat-to-bv [simp]:
  norm-unsigned (nat-to-bv n) = nat-to-bv n
proof -
  have norm-unsigned (nat-to-bv n) = nat-to-bv (bv-to-nat (norm-unsigned (nat-to-bv
n)))
    by (subst nat-bv-nat,simp)
  also have ... = nat-to-bv n
    by simp
  finally show ?thesis .
qed

lemma length-nat-to-bv-upper-limit:
  assumes nk: n ≤ 2 ^ k - 1
```

```
  shows       length (nat-to-bv n) ≤ k
proof (cases n = 0)
  case True
  thus ?thesis
    by (simp add: nat-to-bv-def nat-to-bv-helper.simps)
next
  case False
  hence n0: 0 < n by simp
  show ?thesis
  proof (rule ccontr)
    assume ~ length (nat-to-bv n) ≤ k
    hence k < length (nat-to-bv n)
      by simp
    hence k ≤ length (nat-to-bv n) - 1
      by arith
    hence (2::nat) ^ k ≤ 2 ^ (length (nat-to-bv n) - 1)
      by simp
    also have ... = 2 ^ (length (norm-unsigned (nat-to-bv n)) - 1)
      by simp
    also have ... ≤ bv-to-nat (nat-to-bv n)
      by (rule bv-to-nat-lower-limit,simp add: n0)
    also have ... = n
      by simp
    finally have 2 ^ k ≤ n .
    with n0
    have 2 ^ k - 1 < n
      by arith
    with nk
    show False
      by simp
  qed
qed

lemma length-nat-to-bv-lower-limit:
  assumes nk: 2 ^ k ≤ n
  shows       k < length (nat-to-bv n)
proof (rule ccontr)
  assume ~ k < length (nat-to-bv n)
  hence lnk: length (nat-to-bv n) ≤ k
    by simp
  have n = bv-to-nat (nat-to-bv n)
    by simp
  also have ... < 2 ^ length (nat-to-bv n)
    by (rule bv-to-nat-upper-range)
  also from lnk have ... ≤ 2 ^ k
    by simp
  finally have n < 2 ^ k .
  with nk
  show False
```

by *simp*
qed

16.4 Unsigned Arithmetic Operations

constdefs
bv-add :: [*bit list*, *bit list*] => *bit list*
bv-add w1 w2 == *nat-to-bv* (*bv-to-nat w1* + *bv-to-nat w2*)

lemma *bv-add-type1* [*simp*]: *bv-add* (*norm-unsigned w1*) *w2* = *bv-add w1 w2*
by (*simp add*: *bv-add-def*)

lemma *bv-add-type2* [*simp*]: *bv-add w1* (*norm-unsigned w2*) = *bv-add w1 w2*
by (*simp add*: *bv-add-def*)

lemma *bv-add-returntype* [*simp*]: *norm-unsigned* (*bv-add w1 w2*) = *bv-add w1 w2*
by (*simp add*: *bv-add-def*)

lemma *bv-add-length*: *length* (*bv-add w1 w2*) ≤ *Suc* (*max* (*length w1*) (*length w2*))
proof (*unfold bv-add-def*,*rule length-nat-to-bv-upper-limit*)
from *bv-to-nat-upper-range* [*of w1*] **and** *bv-to-nat-upper-range* [*of w2*]
have *bv-to-nat w1* + *bv-to-nat w2* ≤ (2 ^ *length w1* − 1) + (2 ^ *length w2* − 1)
by *arith*
also have ... ≤ *max* (2 ^ *length w1* − 1) (2 ^ *length w2* − 1) + *max* (2 ^ *length w1* − 1) (2 ^ *length w2* − 1)
by (*rule add-mono*,*safe intro*!: *le-maxI1 le-maxI2*)
also have ... = 2 ∗ *max* (2 ^ *length w1* − 1) (2 ^ *length w2* − 1)
by *simp*
also have ... ≤ 2 ^ *Suc* (*max* (*length w1*) (*length w2*)) − 2
proof (*cases length w1* ≤ *length w2*)
assume *w1w2*: *length w1* ≤ *length w2*
hence (2::*nat*) ^ *length w1* ≤ 2 ^ *length w2*
by *simp*
hence (2::*nat*) ^ *length w1* − 1 ≤ 2 ^ *length w2* − 1
by *arith*
with *w1w2* **show** *?thesis*
by (*simp add*: *diff-mult-distrib2 split*: *split-max*)
next
assume [*simp*]: ~ (*length w1* ≤ *length w2*)
have ~ ((2::*nat*) ^ *length w1* − 1 ≤ 2 ^ *length w2* − 1)
proof
assume (2::*nat*) ^ *length w1* − 1 ≤ 2 ^ *length w2* − 1
hence ((2::*nat*) ^ *length w1* − 1) + 1 ≤ (2 ^ *length w2* − 1) + 1
by (*rule add-right-mono*)
hence (2::*nat*) ^ *length w1* ≤ 2 ^ *length w2*
by *simp*
hence *length w1* ≤ *length w2*
by *simp*
thus *False*

```
      by simp
    qed
    thus ?thesis
      by (simp add: diff-mult-distrib2 split: split-max)
  qed
  finally show bv-to-nat w1 + bv-to-nat w2 ≤ 2 ^ Suc (max (length w1) (length
w2)) − 1
    by arith
qed

constdefs
  bv-mult :: [bit list, bit list ] => bit list
  bv-mult w1 w2 == nat-to-bv (bv-to-nat w1 ∗ bv-to-nat w2)

lemma bv-mult-type1 [simp]: bv-mult (norm-unsigned w1) w2 = bv-mult w1 w2
  by (simp add: bv-mult-def)

lemma bv-mult-type2 [simp]: bv-mult w1 (norm-unsigned w2) = bv-mult w1 w2
  by (simp add: bv-mult-def)

lemma bv-mult-returntype [simp]: norm-unsigned (bv-mult w1 w2) = bv-mult w1
w2
  by (simp add: bv-mult-def)

lemma bv-mult-length: length (bv-mult w1 w2) ≤ length w1 + length w2
proof (unfold bv-mult-def,rule length-nat-to-bv-upper-limit)
  from bv-to-nat-upper-range [of w1] and bv-to-nat-upper-range [of w2]
  have h: bv-to-nat w1 ≤ 2 ^ length w1 − 1 ∧ bv-to-nat w2 ≤ 2 ^ length w2 − 1
    by arith
  have bv-to-nat w1 ∗ bv-to-nat w2 ≤ (2 ^ length w1 − 1) ∗ (2 ^ length w2 − 1)
    apply (cut-tac h)
    apply (rule mult-mono)
    apply auto
    done
  also have ... < 2 ^ length w1 ∗ 2 ^ length w2
    by (rule mult-strict-mono,auto)
  also have ... = 2 ^ (length w1 + length w2)
    by (simp add: power-add)
  finally show bv-to-nat w1 ∗ bv-to-nat w2 ≤ 2 ^ (length w1 + length w2) − 1
    by arith
qed
```

16.5 Signed Vectors

```
consts
  norm-signed :: bit list => bit list

primrec
  norm-signed-Nil: norm-signed [] = []
```

norm-signed-Cons: *norm-signed* (*b*#*bs*) = (*case b of* **0** => *if norm-unsigned bs* = [] *then* [] *else b*#*norm-unsigned bs* | **1** => *b*#*rem-initial b bs*)

lemma *norm-signed0* [*simp*]: *norm-signed* [**0**] = []
by *simp*

lemma *norm-signed1* [*simp*]: *norm-signed* [**1**] = [**1**]
by *simp*

lemma *norm-signed01* [*simp*]: *norm-signed* (**0**#**1**#*xs*) = **0**#**1**#*xs*
by *simp*

lemma *norm-signed00* [*simp*]: *norm-signed* (**0**#**0**#*xs*) = *norm-signed* (**0**#*xs*)
by *simp*

lemma *norm-signed10* [*simp*]: *norm-signed* (**1**#**0**#*xs*) = **1**#**0**#*xs*
by *simp*

lemma *norm-signed11* [*simp*]: *norm-signed* (**1**#**1**#*xs*) = *norm-signed* (**1**#*xs*)
by *simp*

lemmas [*simp del*] = *norm-signed-Cons*

constdefs
int-to-bv :: *int* => *bit list*
int-to-bv n == *if* 0 ≤ *n*
then norm-signed (**0**#*nat-to-bv* (*nat n*))
else norm-signed (*bv-not* (**0**#*nat-to-bv* (*nat* (−*n*− 1)))))

lemma *int-to-bv-ge0* [*simp*]: 0 ≤ *n* ==> *int-to-bv n* = *norm-signed* (**0** # *nat-to-bv* (*nat n*))
by (*simp add*: *int-to-bv-def*)

lemma *int-to-bv-lt0* [*simp*]: *n* < 0 ==> *int-to-bv n* = *norm-signed* (*bv-not* (**0**#*nat-to-bv* (*nat* (−*n*− 1))))
by (*simp add*: *int-to-bv-def*)

lemma *norm-signed-idem* [*simp*]: *norm-signed* (*norm-signed w*) = *norm-signed w*
proof (*rule bit-list-induct* [*of* - *w*],*simp-all*)
fix *xs*
assume *norm-signed* (*norm-signed xs*) = *norm-signed xs*
show *norm-signed* (*norm-signed* (**0**#*xs*)) = *norm-signed* (**0**#*xs*)
proof (*rule bit-list-cases* [*of xs*],*simp-all*)
fix *ys*
assume [*symmetric*,*simp*]: *xs* = **0**#*ys*
show *norm-signed* (*norm-signed* (**0**#*ys*)) = *norm-signed* (**0**#*ys*)
by *simp*
qed
next

```
  fix xs
  assume norm-signed (norm-signed xs) = norm-signed xs
  show norm-signed (norm-signed (1#xs)) = norm-signed (1#xs)
  proof (rule bit-list-cases [of xs],simp-all)
    fix ys
    assume [symmetric,simp]: xs = 1#ys
    show norm-signed (norm-signed (1#ys)) = norm-signed (1#ys)
      by simp
  qed
qed

constdefs
  bv-to-int :: bit list => int
  bv-to-int w == case bv-msb w of 0 => int (bv-to-nat w) | 1 => - int (bv-to-nat
(bv-not w) + 1)

lemma bv-to-int-Nil [simp]: bv-to-int [] = 0
  by (simp add: bv-to-int-def)

lemma bv-to-int-Cons0 [simp]: bv-to-int (0#bs) = int (bv-to-nat bs)
  by (simp add: bv-to-int-def)

lemma bv-to-int-Cons1 [simp]: bv-to-int (1#bs) = - int (bv-to-nat (bv-not bs) +
1)
  by (simp add: bv-to-int-def)

lemma bv-to-int-type [simp]: bv-to-int (norm-signed w) = bv-to-int w
proof (rule bit-list-induct [of - w],simp-all)
  fix xs
  assume ind: bv-to-int (norm-signed xs) = bv-to-int xs
  show bv-to-int (norm-signed (0#xs)) = int (bv-to-nat xs)
  proof (rule bit-list-cases [of xs],simp-all)
    fix ys
    assume [simp]: xs = 0#ys
    from ind
    show bv-to-int (norm-signed (0#ys)) = int (bv-to-nat ys)
      by simp
  qed
next
  fix xs
  assume ind: bv-to-int (norm-signed xs) = bv-to-int xs
  show bv-to-int (norm-signed (1#xs)) = -1 - int (bv-to-nat (bv-not xs))
  proof (rule bit-list-cases [of xs],simp-all)
    fix ys
    assume [simp]: xs = 1#ys
    from ind
    show bv-to-int (norm-signed (1#ys)) = -1 - int (bv-to-nat (bv-not ys))
      by simp
  qed
```

qed

lemma *bv-to-int-upper-range*: *bv-to-int w < 2 ^ (length w − 1)*
proof (*rule bit-list-cases* [*of w*],*simp-all*)
fix *bs*
from *bv-to-nat-upper-range*
show *int* (*bv-to-nat bs*) *< 2 ^ length bs*
by (*simp add*: *int-nat-two-exp*)
next
fix *bs*
have *−1 − int* (*bv-to-nat* (*bv-not bs*)) *≤ 0*
by *simp*
also have *... < 2 ^ length bs*
by (*induct bs*,*simp-all*)
finally show *−1 − int* (*bv-to-nat* (*bv-not bs*)) *< 2 ^ length bs*
.
qed

lemma *bv-to-int-lower-range*: *− (2 ^ (length w − 1)) ≤ bv-to-int w*
proof (*rule bit-list-cases* [*of w*],*simp-all*)
fix *bs* :: *bit list*
have *− (2 ^ length bs) ≤* (*0*::*int*)
by (*induct bs*,*simp-all*)
also have *... ≤ int* (*bv-to-nat bs*)
by *simp*
finally show *− (2 ^ length bs) ≤ int* (*bv-to-nat bs*)
.
next
fix *bs*
from *bv-to-nat-upper-range* [*of bv-not bs*]
show *− (2 ^ length bs) ≤ −1 − int* (*bv-to-nat* (*bv-not bs*))
by (*simp add*: *int-nat-two-exp*)
qed

lemma *int-bv-int* [*simp*]: *int-to-bv* (*bv-to-int w*) = *norm-signed w*
proof (*rule bit-list-cases* [*of w*],*simp*)
fix *xs*
assume [*simp*]: *w* = **0**#*xs*
show *?thesis*
apply *simp*
apply (*subst norm-signed-Cons* [*of* **0** *xs*])
apply *simp*
using *norm-unsigned-result* [*of xs*]
apply *safe*
apply (*rule bit-list-cases* [*of norm-unsigned xs*])
apply *simp-all*
done
next
fix *xs*

```
  assume [simp]: w = 1#xs
  show ?thesis
    apply (simp del: int-to-bv-lt0)
    apply (rule bit-list-induct [of - xs])
    apply simp
    apply (subst int-to-bv-lt0)
    apply (subgoal-tac - int (bv-to-nat (bv-not (0 # bs))) + -1 < 0 + 0)
    apply simp
    apply (rule add-le-less-mono)
    apply simp
    apply simp
    apply (simp del: bv-to-nat1 bv-to-nat-helper)
    apply simp
    done
qed

lemma bv-int-bv [simp]: bv-to-int (int-to-bv i) = i
  by (cases 0 ≤ i,simp-all)

lemma bv-msb-norm [simp]: bv-msb (norm-signed w) = bv-msb w
  by (rule bit-list-cases [of w],simp-all add: norm-signed-Cons)

lemma norm-signed-length: length (norm-signed w) ≤ length w
  apply (cases w,simp-all)
  apply (subst norm-signed-Cons)
  apply (case-tac a,simp-all)
  apply (rule rem-initial-length)
  done

lemma norm-signed-equal: length (norm-signed w) = length w ==> norm-signed
w = w
proof (rule bit-list-cases [of w],simp-all)
  fix xs
  assume length (norm-signed (0#xs)) = Suc (length xs)
  thus norm-signed (0#xs) = 0#xs
    apply (simp add: norm-signed-Cons)
    apply safe
    apply simp-all
    apply (rule norm-unsigned-equal)
    apply assumption
    done
next
  fix xs
  assume length (norm-signed (1#xs)) = Suc (length xs)
  thus norm-signed (1#xs) = 1#xs
    apply (simp add: norm-signed-Cons)
    apply (rule rem-initial-equal)
    apply assumption
    done
```

qed

lemma *bv-extend-norm-signed*: *bv-msb w = b ==> bv-extend (length w) b (norm-signed w) = w*
proof (*rule bit-list-cases* [*of w*],*simp-all*)
fix *xs*
show *bv-extend* (*Suc* (*length xs*)) **0** (*norm-signed* (**0**#*xs*)) = **0**#*xs*
proof (*simp add*: *norm-signed-list-def*,*auto*)
assume *norm-unsigned xs* = []
hence *xx*: *rem-initial* **0** *xs* = []
by (*simp add*: *norm-unsigned-def*)
have *bv-extend* (*Suc* (*length xs*)) **0** (**0**#*rem-initial* **0** *xs*) = **0**#*xs*
apply (*simp add*: *bv-extend-def replicate-app-Cons-same*)
apply (*fold bv-extend-def*)
apply (*rule bv-extend-rem-initial*)
done
thus *bv-extend* (*Suc* (*length xs*)) **0** [**0**] = **0**#*xs*
by (*simp add*: *xx*)
next
show *bv-extend* (*Suc* (*length xs*)) **0** (**0**#*norm-unsigned xs*) = **0**#*xs*
apply (*simp add*: *norm-unsigned-def*)
apply (*simp add*: *bv-extend-def replicate-app-Cons-same*)
apply (*fold bv-extend-def*)
apply (*rule bv-extend-rem-initial*)
done
qed
next
fix *xs*
show *bv-extend* (*Suc* (*length xs*)) **1** (*norm-signed* (**1**#*xs*)) = **1**#*xs*
apply (*simp add*: *norm-signed-Cons*)
apply (*simp add*: *bv-extend-def replicate-app-Cons-same*)
apply (*fold bv-extend-def*)
apply (*rule bv-extend-rem-initial*)
done
qed

lemma *bv-to-int-qinj*:
assumes *one*: *bv-to-int xs = bv-to-int ys*
and *len*: *length xs = length ys*
shows *xs = ys*
proof −
from *one*
have *int-to-bv* (*bv-to-int xs*) = *int-to-bv* (*bv-to-int ys*)
by *simp*
hence *xsys*: *norm-signed xs = norm-signed ys*
by *simp*
hence *xsys'*: *bv-msb xs = bv-msb ys*
proof −
have *bv-msb xs = bv-msb* (*norm-signed xs*)

```
    by simp
  also have ... = bv-msb (norm-signed ys)
    by (simp add: xsys)
  also have ... = bv-msb ys
    by simp
  finally show ?thesis .
 qed
 have xs = bv-extend (length xs) (bv-msb xs) (norm-signed xs)
  by (simp add: bv-extend-norm-signed)
 also have ... = bv-extend (length ys) (bv-msb ys) (norm-signed ys)
  by (simp add: xsys xsys' len)
 also have ... = ys
  by (simp add: bv-extend-norm-signed)
 finally show ?thesis .
qed

lemma int-to-bv-returntype [simp]: norm-signed (int-to-bv w) = int-to-bv w
 by (simp add: int-to-bv-def)

lemma bv-to-int-msb0: 0 ≤ bv-to-int w1 ==> bv-msb w1 = 0
 by (rule bit-list-cases,simp-all)

lemma bv-to-int-msb1: bv-to-int w1 < 0 ==> bv-msb w1 = 1
 by (rule bit-list-cases,simp-all)

lemma bv-to-int-lower-limit-gt0:
 assumes w0: 0 < bv-to-int w
 shows       2 ^ (length (norm-signed w) - 2) ≤ bv-to-int w
proof -
 from w0
 have 0 ≤ bv-to-int w
  by simp
 hence [simp]: bv-msb w = 0
  by (rule bv-to-int-msb0)
 have 2 ^ (length (norm-signed w) - 2) ≤ bv-to-int (norm-signed w)
 proof (rule bit-list-cases [of w])
  assume w = []
  with w0
  show ?thesis
   by simp
 next
  fix w'
  assume weq: w = 0 # w'
  thus ?thesis
  proof (simp add: norm-signed-Cons,safe)
   assume norm-unsigned w' = []
   with weq and w0
   show False
    by (simp add: norm-empty-bv-to-nat-zero)
```

```
    next
      assume w'0: norm-unsigned w' ≠ []
      have 0 < bv-to-nat w'
      proof (rule ccontr)
        assume ~ (0 < bv-to-nat w')
        hence bv-to-nat w' = 0
          by arith
        hence norm-unsigned w' = []
          by (simp add: bv-to-nat-zero-imp-empty)
        with w'0
        show False
          by simp
      qed
      with bv-to-nat-lower-limit [of w']
      show 2 ^ (length (norm-unsigned w') - Suc 0) ≤ int (bv-to-nat w')
        by (simp add: int-nat-two-exp)
    qed
  next
    fix w'
    assume w = 1 # w'
    from w0
    have bv-msb w = 0
      by simp
    with prems
    show ?thesis
      by simp
  qed
  also have ... = bv-to-int w
    by simp
  finally show ?thesis .
qed

lemma norm-signed-result: norm-signed w = [] ∨ norm-signed w = [1] ∨ bv-msb
(norm-signed w) ≠ bv-msb (tl (norm-signed w))
  apply (rule bit-list-cases [of w],simp-all)
  apply (case-tac bs,simp-all)
  apply (case-tac a,simp-all)
  apply (simp add: norm-signed-Cons)
  apply safe
  apply simp
proof -
  fix l
  assume msb: 0 = bv-msb (norm-unsigned l)
  assume norm-unsigned l ≠ []
  with norm-unsigned-result [of l]
  have bv-msb (norm-unsigned l) = 1
    by simp
  with msb
  show False
```

```
    by simp
next
  fix xs
  assume p: 1 = bv-msb (tl (norm-signed (1 # xs)))
  have 1 ≠ bv-msb (tl (norm-signed (1 # xs)))
    by (rule bit-list-induct [of - xs],simp-all)
  with p
  show False
    by simp
qed

lemma bv-to-int-upper-limit-lem1:
  assumes w0: bv-to-int w < -1
  shows       bv-to-int w < - (2 ^ (length (norm-signed w) - 2))
proof -
  from w0
  have bv-to-int w < 0
    by simp
  hence msbw [simp]: bv-msb w = 1
    by (rule bv-to-int-msb1)
  have bv-to-int w = bv-to-int (norm-signed w)
    by simp
  also from norm-signed-result [of w]
  have ... < - (2 ^ (length (norm-signed w) - 2))
  proof (safe)
    assume norm-signed w = []
    hence bv-to-int (norm-signed w) = 0
      by simp
    with w0
    show ?thesis
      by simp
  next
    assume norm-signed w = [1]
    hence bv-to-int (norm-signed w) = -1
      by simp
    with w0
    show ?thesis
      by simp
  next
    assume bv-msb (norm-signed w) ≠ bv-msb (tl (norm-signed w))
    hence msb-tl: 1 ≠ bv-msb (tl (norm-signed w))
      by simp
    show bv-to-int (norm-signed w) < - (2 ^ (length (norm-signed w) - 2))
    proof (rule bit-list-cases [of norm-signed w])
      assume norm-signed w = []
      hence bv-to-int (norm-signed w) = 0
        by simp
      with w0
      show ?thesis
```

```
      by simp
    next
      fix w'
      assume nw: norm-signed w = 0 # w'
      from msbw
      have bv-msb (norm-signed w) = 1
        by simp
      with nw
      show ?thesis
        by simp
    next
      fix w'
      assume weq: norm-signed w = 1 # w'
      show ?thesis
      proof (rule bit-list-cases [of w'])
        assume w'eq: w' = []
        from w0
        have bv-to-int (norm-signed w) < -1
          by simp
        with w'eq and weq
        show ?thesis
          by simp
      next
        fix w''
        assume w'eq: w' = 0 # w''
        show ?thesis
          apply (simp add: weq w'eq)
          apply (subgoal-tac - int (bv-to-nat (bv-not w'')) + -1 < 0 + 0)
          apply (simp add: int-nat-two-exp)
          apply (rule add-le-less-mono)
          apply simp-all
          done
      next
        fix w''
        assume w'eq: w' = 1 # w''
        with weq and msb-tl
        show ?thesis
          by simp
      qed
    qed
  qed
  finally show ?thesis .
qed

lemma length-int-to-bv-upper-limit-gt0:
  assumes w0: 0 < i
  and     wk: i ≤ 2 ^ (k - 1) - 1
  shows       length (int-to-bv i) ≤ k
proof (rule ccontr)
```

```
  from w0 wk
  have k1: 1 < k
    by (cases k − 1,simp-all,arith)
  assume ~ length (int-to-bv i) ≤ k
  hence k < length (int-to-bv i)
    by simp
  hence k ≤ length (int-to-bv i) − 1
    by arith
  hence a: k − 1 ≤ length (int-to-bv i) − 2
    by arith
  hence (2::int) ^ (k − 1) ≤ 2 ^ (length (int-to-bv i) − 2) by simp
  also have ... ≤ i
  proof −
    have 2 ^ (length (norm-signed (int-to-bv i)) − 2) ≤ bv-to-int (int-to-bv i)
    proof (rule bv-to-int-lower-limit-gt0)
      from w0
      show 0 < bv-to-int (int-to-bv i)
        by simp
    qed
    thus ?thesis
      by simp
  qed
  finally have 2 ^ (k − 1) ≤ i .
  with wk
  show False
    by simp
qed

lemma pos-length-pos:
  assumes i0: 0 < bv-to-int w
  shows       0 < length w
proof −
  from norm-signed-result [of w]
  have 0 < length (norm-signed w)
  proof (auto)
    assume ii: norm-signed w = []
    have bv-to-int (norm-signed w) = 0
      by (subst ii,simp)
    hence bv-to-int w = 0
      by simp
    with i0
    show False
      by simp
  next
    assume ii: norm-signed w = []
    assume jj: bv-msb w ≠ 0
    have 0 = bv-msb (norm-signed w)
      by (subst ii,simp)
    also have ... ≠ 0
```

```
      by (simp add: jj)
    finally show False by simp
  qed
  also have ... ≤ length w
    by (rule norm-signed-length)
  finally show ?thesis
    .
qed

lemma neg-length-pos:
  assumes i0: bv-to-int w < -1
  shows       0 < length w
proof -
  from norm-signed-result [of w]
  have 0 < length (norm-signed w)
  proof (auto)
    assume ii: norm-signed w = []
    have bv-to-int (norm-signed w) = 0
      by (subst ii,simp)
    hence bv-to-int w = 0
      by simp
    with i0
    show False
      by simp
  next
    assume ii: norm-signed w = []
    assume jj: bv-msb w ≠ 0
    have 0 = bv-msb (norm-signed w)
      by (subst ii,simp)
    also have ... ≠ 0
      by (simp add: jj)
    finally show False by simp
  qed
  also have ... ≤ length w
    by (rule norm-signed-length)
  finally show ?thesis
    .
qed

lemma length-int-to-bv-lower-limit-gt0:
  assumes wk: 2 ^ (k - 1) ≤ i
  shows       k < length (int-to-bv i)
proof (rule ccontr)
  have 0 < (2::int) ^ (k - 1)
    by (rule zero-less-power,simp)
  also have ... ≤ i
    by (rule wk)
  finally have i0: 0 < i
    .
```

```
  have lii0: 0 < length (int-to-bv i)
    apply (rule pos-length-pos)
    apply (simp,rule i0)
    done
  assume ~ k < length (int-to-bv i)
  hence length (int-to-bv i) ≤ k
    by simp
  with lii0
  have a: length (int-to-bv i) − 1 ≤ k − 1
    by arith
  have i < 2 ^ (length (int-to-bv i) − 1)
  proof −
    have i = bv-to-int (int-to-bv i)
      by simp
    also have ... < 2 ^ (length (int-to-bv i) − 1)
      by (rule bv-to-int-upper-range)
    finally show ?thesis .
  qed
  also have (2::int) ^ (length (int-to-bv i) − 1) ≤ 2 ^ (k − 1) using a
        by simp
  finally have i < 2 ^ (k − 1) .
  with wk
  show False
    by simp
qed

lemma length-int-to-bv-upper-limit-lem1:
  assumes w1: i < −1
  and     wk: − (2 ^ (k − 1)) ≤ i
  shows       length (int-to-bv i) ≤ k
proof (rule ccontr)
  from w1 wk
  have k1: 1 < k
    by (cases k − 1,simp-all,arith)
  assume ~ length (int-to-bv i) ≤ k
  hence k < length (int-to-bv i)
    by simp
  hence k ≤ length (int-to-bv i) − 1
    by arith
  hence a: k − 1 ≤ length (int-to-bv i) − 2
    by arith
  have i < − (2 ^ (length (int-to-bv i) − 2))
  proof −
    have i = bv-to-int (int-to-bv i)
      by simp
    also have ... < − (2 ^ (length (norm-signed (int-to-bv i)) − 2))
      by (rule bv-to-int-upper-limit-lem1,simp,rule w1)
    finally show ?thesis by simp
  qed
```

```
  also have ... ≤ −(2 ^ (k − 1))
  proof −
    have (2::int) ^ (k − 1) ≤ 2 ^ (length (int-to-bv i) − 2) using a
      by simp
    thus ?thesis
      by simp
  qed
  finally have i < −(2 ^ (k − 1)) .
  with wk
  show False
    by simp
qed

lemma length-int-to-bv-lower-limit-lem1:
  assumes wk: i < −(2 ^ (k − 1))
  shows       k < length (int-to-bv i)
proof (rule ccontr)
  from wk
  have i ≤ −(2 ^ (k − 1)) − 1
    by simp
  also have ... < −1
  proof −
    have 0 < (2::int) ^ (k − 1)
      by (rule zero-less-power,simp)
    hence −((2::int) ^ (k − 1)) < 0
      by simp
    thus ?thesis by simp
  qed
  finally have i1: i < −1 .
  have lii0: 0 < length (int-to-bv i)
    apply (rule neg-length-pos)
    apply (simp,rule i1)
    done
  assume ~ k < length (int-to-bv i)
  hence length (int-to-bv i) ≤ k
    by simp
  with lii0
  have a: length (int-to-bv i) − 1 ≤ k − 1
    by arith
  hence (2::int) ^ (length (int-to-bv i) − 1) ≤ 2 ^ (k − 1) by simp
  hence −((2::int) ^ (k − 1)) ≤ − (2 ^ (length (int-to-bv i) − 1))
    by simp
  also have ... ≤ i
  proof −
    have − (2 ^ (length (int-to-bv i) − 1)) ≤ bv-to-int (int-to-bv i)
      by (rule bv-to-int-lower-range)
    also have ... = i
      by simp
    finally show ?thesis .
```

qed
finally have $-(2 \hat{} (k - 1)) \leq i$ **.**
with *wk*
show *False*
by *simp*
qed

16.6 Signed Arithmetic Operations

16.6.1 Conversion from unsigned to signed

constdefs
utos :: *bit list* => *bit list*
utos w == *norm-signed* (**0** # *w*)

lemma *utos-type* [*simp*]: *utos* (*norm-unsigned w*) = *utos w*
by (*simp add*: *utos-def norm-signed-Cons*)

lemma *utos-returntype* [*simp*]: *norm-signed* (*utos w*) = *utos w*
by (*simp add*: *utos-def*)

lemma *utos-length*: *length* (*utos w*) ≤ *Suc* (*length w*)
by (*simp add*: *utos-def norm-signed-Cons*)

lemma *bv-to-int-utos*: *bv-to-int* (*utos w*) = *int* (*bv-to-nat w*)
proof (*simp add*: *utos-def norm-signed-Cons*,*safe*)
assume *norm-unsigned w* = []
hence *bv-to-nat* (*norm-unsigned w*) = *0*
by *simp*
thus *bv-to-nat w* = *0*
by *simp*
qed

16.6.2 Unary minus

constdefs
bv-uminus :: *bit list* => *bit list*
bv-uminus w == *int-to-bv* (− *bv-to-int w*)

lemma *bv-uminus-type* [*simp*]: *bv-uminus* (*norm-signed w*) = *bv-uminus w*
by (*simp add*: *bv-uminus-def*)

lemma *bv-uminus-returntype* [*simp*]: *norm-signed* (*bv-uminus w*) = *bv-uminus w*
by (*simp add*: *bv-uminus-def*)

lemma *bv-uminus-length*: *length* (*bv-uminus w*) ≤ *Suc* (*length w*)
proof −
have *1* < −*bv-to-int w* ∨ −*bv-to-int w* = *1* ∨ −*bv-to-int w* = *0* ∨ −*bv-to-int w* = −*1* ∨ −*bv-to-int w* < −*1*
by *arith*

thus *?thesis*
proof *safe*
assume *p*: *1 < − bv-to-int w*
have *lw*: *0 < length w*
apply (*rule neg-length-pos*)
using *p*
apply *simp*
done
show *?thesis*
proof (*simp add*: *bv-uminus-def*,*rule length-int-to-bv-upper-limit-gt0*,*simp-all*)
from *prems*
show *bv-to-int w < 0*
by *simp*
next
have *−(2^(length w − 1)) ≤ bv-to-int w*
by (*rule bv-to-int-lower-range*)
hence *− bv-to-int w ≤ 2^(length w − 1)*
by *simp*
also from *lw* **have** *... < 2 ^ length w*
by *simp*
finally show *− bv-to-int w < 2 ^ length w*
by *simp*
qed
next
assume *p*: *− bv-to-int w = 1*
hence *lw*: *0 < length w*
by (*cases w*,*simp-all*)
from *p*
show *?thesis*
apply (*simp add*: *bv-uminus-def*)
using *lw*
apply (*simp* (*no-asm*) *add*: *nat-to-bv-non0*)
done
next
assume *− bv-to-int w = 0*
thus *?thesis*
by (*simp add*: *bv-uminus-def*)
next
assume *p*: *− bv-to-int w = −1*
thus *?thesis*
by (*simp add*: *bv-uminus-def*)
next
assume *p*: *− bv-to-int w < −1*
show *?thesis*
apply (*simp add*: *bv-uminus-def*)
apply (*rule length-int-to-bv-upper-limit-lem1*)
apply (*rule p*)
apply *simp*
proof *−*

```
    have bv-to-int w < 2 ^ (length w - 1)
      by (rule bv-to-int-upper-range)
    also have ... ≤ 2 ^ length w by simp
    finally show bv-to-int w ≤ 2 ^ length w
      by simp
  qed
 qed
qed

lemma bv-uminus-length-utos: length (bv-uminus (utos w)) ≤ Suc (length w)
proof -
 have -bv-to-int (utos w) = 0 ∨ -bv-to-int (utos w) = -1 ∨ -bv-to-int (utos
w) < -1
  apply (simp add: bv-to-int-utos)
  by arith
 thus ?thesis
 proof safe
  assume -bv-to-int (utos w) = 0
  thus ?thesis
   by (simp add: bv-uminus-def)
 next
  assume -bv-to-int (utos w) = -1
  thus ?thesis
   by (simp add: bv-uminus-def)
 next
  assume p: -bv-to-int (utos w) < -1
  show ?thesis
   apply (simp add: bv-uminus-def)
   apply (rule length-int-to-bv-upper-limit-lem1)
   apply (rule p)
   apply (simp add: bv-to-int-utos)
   using bv-to-nat-upper-range [of w]
   apply (simp add: int-nat-two-exp)
   done
 qed
qed

constdefs
 bv-sadd :: [bit list, bit list ] => bit list
 bv-sadd w1 w2 == int-to-bv (bv-to-int w1 + bv-to-int w2)

lemma bv-sadd-type1 [simp]: bv-sadd (norm-signed w1) w2 = bv-sadd w1 w2
 by (simp add: bv-sadd-def)

lemma bv-sadd-type2 [simp]: bv-sadd w1 (norm-signed w2) = bv-sadd w1 w2
 by (simp add: bv-sadd-def)

lemma bv-sadd-returntype [simp]: norm-signed (bv-sadd w1 w2) = bv-sadd w1 w2
 by (simp add: bv-sadd-def)
```

lemma *adder-helper*:
assumes *lw*: *0 < max (length w1) (length w2)*
shows *((2::int) ^ (length w1 − 1)) + (2 ^ (length w2 − 1)) ≤ 2 ^ max (length w1) (length w2)*
proof −
have *((2::int) ^ (length w1 − 1)) + (2 ^ (length w2 − 1)) ≤ 2 ^ (max (length w1) (length w2) − 1) + 2 ^ (max (length w1) (length w2) − 1)*
apply *(cases length w1 ≤ length w2)*
apply *(auto simp add: max-def)*
apply *arith*
apply *arith*
done
also have *... = 2 ^ max (length w1) (length w2)*
proof −
from *lw*
show *?thesis*
apply *simp*
apply *(subst power-Suc [symmetric])*
apply *(simp del: power.simps)*
done
qed
finally show *?thesis* **.**
qed

lemma *bv-sadd-length*: *length (bv-sadd w1 w2) ≤ Suc (max (length w1) (length w2))*
proof −
let *?Q = bv-to-int w1 + bv-to-int w2*

have *helper*: *?Q ≠ 0 ==> 0 < max (length w1) (length w2)*
proof −
assume *p*: *?Q ≠ 0*
show *0 < max (length w1) (length w2)*
proof *(simp add: less-max-iff-disj,rule)*
assume *[simp]*: *w1 = []*
show *w2 ≠ []*
proof *(rule ccontr,simp)*
assume *[simp]*: *w2 = []*
from *p*
show *False*
by *simp*
qed
qed
qed

have *0 < ?Q ∨ ?Q = 0 ∨ ?Q = −1 ∨ ?Q < −1*
by *arith*
thus *?thesis*

proof *safe*
assume *?Q = 0*
thus *?thesis*
by (*simp add*: *bv-sadd-def*)
next
assume *?Q = −1*
thus *?thesis*
by (*simp add*: *bv-sadd-def*)
next
assume *p*: *0 < ?Q*
show *?thesis*
apply (*simp add*: *bv-sadd-def*)
apply (*rule length-int-to-bv-upper-limit-gt0*)
apply (*rule p*)
proof *simp*
from *bv-to-int-upper-range* [*of w2*]
have *bv-to-int w2 ≤ 2 ^ (length w2 − 1)*
by *simp*
with *bv-to-int-upper-range* [*of w1*]
have *bv-to-int w1 + bv-to-int w2 < (2 ^ (length w1 − 1)) + (2 ^ (length w2 − 1))*
by (*rule zadd-zless-mono*)
also have *... ≤ 2 ^ max (length w1) (length w2)*
apply (*rule adder-helper*)
apply (*rule helper*)
using *p*
apply *simp*
done
finally show *?Q < 2 ^ max (length w1) (length w2)*
.
qed
next
assume *p*: *?Q < −1*
show *?thesis*
apply (*simp add*: *bv-sadd-def*)
apply (*rule length-int-to-bv-upper-limit-lem1*,*simp-all*)
apply (*rule p*)
proof −
have *(2 ^ (length w1 − 1)) + 2 ^ (length w2 − 1) ≤ (2::int) ^ max (length w1) (length w2)*
apply (*rule adder-helper*)
apply (*rule helper*)
using *p*
apply *simp*
done
hence *−((2::int) ^ max (length w1) (length w2)) ≤ − (2 ^ (length w1 − 1)) + −(2 ^ (length w2 − 1))*
by *simp*
also have *− (2 ^ (length w1 − 1)) + −(2 ^ (length w2 − 1)) ≤ ?Q*

```
      apply (rule add-mono)
      apply (rule bv-to-int-lower-range [of w1])
      apply (rule bv-to-int-lower-range [of w2])
      done
    finally show - (2^max (length w1) (length w2)) ≤ ?Q .
  qed
 qed
qed

constdefs
  bv-sub :: [bit list, bit list] => bit list
  bv-sub w1 w2 == bv-sadd w1 (bv-uminus w2)

lemma bv-sub-type1 [simp]: bv-sub (norm-signed w1) w2 = bv-sub w1 w2
  by (simp add: bv-sub-def)

lemma bv-sub-type2 [simp]: bv-sub w1 (norm-signed w2) = bv-sub w1 w2
  by (simp add: bv-sub-def)

lemma bv-sub-returntype [simp]: norm-signed (bv-sub w1 w2) = bv-sub w1 w2
  by (simp add: bv-sub-def)

lemma bv-sub-length: length (bv-sub w1 w2) ≤ Suc (max (length w1) (length w2))
proof (cases bv-to-int w2 = 0)
  assume p: bv-to-int w2 = 0
  show ?thesis
  proof (simp add: bv-sub-def bv-sadd-def bv-uminus-def p)
    have length (norm-signed w1) ≤ length w1
      by (rule norm-signed-length)
    also have ... ≤ max (length w1) (length w2)
      by (rule le-maxI1)
    also have ... ≤ Suc (max (length w1) (length w2))
      by arith
    finally show length (norm-signed w1) ≤ Suc (max (length w1) (length w2))
      .
  qed
next
  assume bv-to-int w2 ≠ 0
  hence 0 < length w2
    by (cases w2,simp-all)
  hence lmw: 0 < max (length w1) (length w2)
    by arith

  let ?Q = bv-to-int w1 - bv-to-int w2

  have 0 < ?Q ∨ ?Q = 0 ∨ ?Q = -1 ∨ ?Q < -1
    by arith
  thus ?thesis
  proof safe
```

assume *?Q = 0*
thus *?thesis*
by (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def*)
next
assume *?Q = −1*
thus *?thesis*
by (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def*)
next
assume *p*: *0 < ?Q*
show *?thesis*
apply (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def*)
apply (*rule length-int-to-bv-upper-limit-gt0*)
apply (*rule p*)
proof *simp*
from *bv-to-int-lower-range* [*of w2*]
have *v2*: *− bv-to-int w2 ≤ 2 ^ (length w2 − 1)*
by *simp*
have *bv-to-int w1 + − bv-to-int w2 < (2 ^ (length w1 − 1)) + (2 ^ (length w2 − 1))*
apply (*rule zadd-zless-mono*)
apply (*rule bv-to-int-upper-range* [*of w1*])
apply (*rule v2*)
done
also have *... ≤ 2 ^ max (length w1) (length w2)*
apply (*rule adder-helper*)
apply (*rule lmw*)
done
finally show *?Q < 2 ^ max (length w1) (length w2)*
by *simp*
qed
next
assume *p*: *?Q < −1*
show *?thesis*
apply (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def*)
apply (*rule length-int-to-bv-upper-limit-lem1*)
apply (*rule p*)
proof *simp*
have *(2 ^ (length w1 − 1)) + 2 ^ (length w2 − 1) ≤ (2::int) ^ max (length w1) (length w2)*
apply (*rule adder-helper*)
apply (*rule lmw*)
done
hence *−((2::int) ^ max (length w1) (length w2)) ≤ − (2 ^ (length w1 − 1)) + −(2 ^ (length w2 − 1))*
by *simp*
also have *− (2 ^ (length w1 − 1)) + −(2 ^ (length w2 − 1)) ≤ bv-to-int w1 + −bv-to-int w2*
apply (*rule add-mono*)
apply (*rule bv-to-int-lower-range* [*of w1*])

```
      using bv-to-int-upper-range [of w2]
      apply simp
      done
    finally show - (2^max (length w1) (length w2)) ≤ ?Q
      by simp
  qed
 qed
qed

constdefs
  bv-smult :: [bit list, bit list] => bit list
  bv-smult w1 w2 == int-to-bv (bv-to-int w1 * bv-to-int w2)

lemma bv-smult-type1 [simp]: bv-smult (norm-signed w1) w2 = bv-smult w1 w2
  by (simp add: bv-smult-def)

lemma bv-smult-type2 [simp]: bv-smult w1 (norm-signed w2) = bv-smult w1 w2
  by (simp add: bv-smult-def)

lemma bv-smult-returntype [simp]: norm-signed (bv-smult w1 w2) = bv-smult w1
w2
  by (simp add: bv-smult-def)

lemma bv-smult-length: length (bv-smult w1 w2) ≤ length w1 + length w2
proof -
  let ?Q = bv-to-int w1 * bv-to-int w2

  have lmw: ?Q ≠ 0 ==> 0 < length w1 ∧ 0 < length w2
    by auto

  have 0 < ?Q ∨ ?Q = 0 ∨ ?Q = -1 ∨ ?Q < -1
    by arith
  thus ?thesis
  proof (safe dest!: iffD1 [OF mult-eq-0-iff])
    assume bv-to-int w1 = 0
    thus ?thesis
      by (simp add: bv-smult-def)
  next
    assume bv-to-int w2 = 0
    thus ?thesis
      by (simp add: bv-smult-def)
  next
    assume p: ?Q = -1
    show ?thesis
      apply (simp add: bv-smult-def p)
      apply (cut-tac lmw)
      apply arith
      using p
      apply simp
```

```
    done
  next
    assume p: 0 < ?Q
    thus ?thesis
    proof (simp add: zero-less-mult-iff,safe)
      assume bi1: 0 < bv-to-int w1
      assume bi2: 0 < bv-to-int w2
      show ?thesis
        apply (simp add: bv-smult-def)
        apply (rule length-int-to-bv-upper-limit-gt0)
        apply (rule p)
      proof simp
        have ?Q < 2 ^ (length w1 - 1) * 2 ^ (length w2 - 1)
          apply (rule mult-strict-mono)
          apply (rule bv-to-int-upper-range)
          apply (rule bv-to-int-upper-range)
          apply (rule zero-less-power)
          apply simp
          using bi2
          apply simp
          done
        also have ... ≤ 2 ^ (length w1 + length w2 - Suc 0)
          apply simp
          apply (subst zpower-zadd-distrib [symmetric])
          apply simp
          apply arith
          done
        finally show ?Q < 2 ^ (length w1 + length w2 - Suc 0)
          .
      qed
    next
      assume bi1: bv-to-int w1 < 0
      assume bi2: bv-to-int w2 < 0
      show ?thesis
        apply (simp add: bv-smult-def)
        apply (rule length-int-to-bv-upper-limit-gt0)
        apply (rule p)
      proof simp
       have -bv-to-int w1 * -bv-to-int w2 ≤ 2 ^ (length w1 - 1) * 2 ^(length w2
- 1)
          apply (rule mult-mono)
          using bv-to-int-lower-range [of w1]
          apply simp
          using bv-to-int-lower-range [of w2]
          apply simp
          apply (rule zero-le-power,simp)
          using bi2
          apply simp
          done
```

```
      hence ?Q ≤ 2 ^ (length w1 − 1) ∗ 2 ^(length w2 − 1)
        by simp
      also have ... < 2 ^ (length w1 + length w2 − Suc 0)
        apply simp
        apply (subst zpower-zadd-distrib [symmetric])
        apply simp
        apply (cut-tac lmw)
        apply arith
        apply (cut-tac p)
        apply arith
        done
      finally show ?Q < 2 ^ (length w1 + length w2 − Suc 0) .
    qed
  qed
 next
  assume p: ?Q < −1
  show ?thesis
    apply (subst bv-smult-def)
    apply (rule length-int-to-bv-upper-limit-lem1)
    apply (rule p)
  proof simp
    have (2::int) ^ (length w1 − 1) ∗ 2 ^(length w2 − 1) ≤ 2 ^ (length w1 +
length w2 − Suc 0)
      apply simp
      apply (subst zpower-zadd-distrib [symmetric])
      apply simp
      apply (cut-tac lmw)
      apply arith
      apply (cut-tac p)
      apply arith
      done
    hence −((2::int) ^ (length w1 + length w2 − Suc 0)) ≤ −(2^(length w1 −
1) ∗ 2 ^ (length w2 − 1))
      by simp
    also have ... ≤ ?Q
    proof −
      from p
      have q: bv-to-int w1 ∗ bv-to-int w2 < 0
        by simp
      thus ?thesis
      proof (simp add: mult-less-0-iff,safe)
        assume bi1: 0 < bv-to-int w1
        assume bi2: bv-to-int w2 < 0
         have −bv-to-int w2 ∗ bv-to-int w1 ≤ ((2::int)^(length w2 − 1)) ∗ (2 ^
(length w1 − 1))
          apply (rule mult-mono)
          using bv-to-int-lower-range [of w2]
          apply simp
          using bv-to-int-upper-range [of w1]
```

```
          apply simp
          apply (rule zero-le-power,simp)
          using bi1
          apply simp
          done
        hence -?Q ≤ ((2::int) ^ (length w1 - 1)) * (2 ^ (length w2 - 1))
          by (simp add: zmult-ac)
        thus -(((2::int) ^ (length w1 - Suc 0)) * (2 ^ (length w2 - Suc 0))) ≤
?Q
          by simp
      next
        assume bi1: bv-to-int w1 < 0
        assume bi2: 0 < bv-to-int w2
        have -bv-to-int w1 * bv-to-int w2 ≤ ((2::int) ^ (length w1 - 1)) * (2 ^
(length w2 - 1))
          apply (rule mult-mono)
          using bv-to-int-lower-range [of w1]
          apply simp
          using bv-to-int-upper-range [of w2]
          apply simp
          apply (rule zero-le-power,simp)
          using bi2
          apply simp
          done
        hence -?Q ≤ ((2::int) ^ (length w1 - 1)) * (2 ^ (length w2 - 1))
          by (simp add: zmult-ac)
        thus -(((2::int) ^ (length w1 - Suc 0)) * (2 ^ (length w2 - Suc 0))) ≤
?Q
          by simp
      qed
    qed
    finally show -(2 ^ (length w1 + length w2 - Suc 0)) ≤ ?Q
      .
  qed
 qed
qed

lemma bv-msb-one: bv-msb w = 1 ==> 0 < bv-to-nat w
 by (cases w,simp-all)

lemma bv-smult-length-utos: length (bv-smult (utos w1) w2) ≤ length w1 + length
w2
proof -
 let ?Q = bv-to-int (utos w1) * bv-to-int w2

 have lmw: ?Q ≠ 0 ==> 0 < length (utos w1) ∧ 0 < length w2
  by auto

 have 0 < ?Q ∨ ?Q = 0 ∨ ?Q = -1 ∨ ?Q < -1
```

```
  by arith
thus ?thesis
proof (safe dest!: iffD1 [OF mult-eq-0-iff])
  assume bv-to-int (utos w1) = 0
  thus ?thesis
    by (simp add: bv-smult-def)
next
  assume bv-to-int w2 = 0
  thus ?thesis
    by (simp add: bv-smult-def)
next
  assume p: 0 < ?Q
  thus ?thesis
  proof (simp add: zero-less-mult-iff,safe)
    assume biw2: 0 < bv-to-int w2
    show ?thesis
      apply (simp add: bv-smult-def)
      apply (rule length-int-to-bv-upper-limit-gt0)
      apply (rule p)
    proof simp
      have ?Q < 2 ^ length w1 * 2 ^ (length w2 - 1)
        apply (rule mult-strict-mono)
        apply (simp add: bv-to-int-utos int-nat-two-exp)
        apply (rule bv-to-nat-upper-range)
        apply (rule bv-to-int-upper-range)
        apply (rule zero-less-power,simp)
        using biw2
        apply simp
        done
      also have ... ≤ 2 ^ (length w1 + length w2 - Suc 0)
        apply simp
        apply (subst zpower-zadd-distrib [symmetric])
        apply simp
        apply (cut-tac lmw)
        apply arith
        using p
        apply auto
        done
      finally show ?Q < 2 ^ (length w1 + length w2 - Suc 0)
        .
    qed
  next
    assume bv-to-int (utos w1) < 0
    thus ?thesis
      by (simp add: bv-to-int-utos)
  qed
next
  assume p: ?Q = -1
  thus ?thesis
```

apply (*simp add*: *bv-smult-def*)
apply (*cut-tac lmw*)
apply *arith*
apply *simp*
done
next
assume *p*: *?Q* < −*1*
show *?thesis*
apply (*subst bv-smult-def*)
apply (*rule length-int-to-bv-upper-limit-lem1*)
apply (*rule p*)
proof *simp*
have (*2*::*int*) ^ *length w1* ∗ *2* ^(*length w2* − *1*) ≤ *2* ^ (*length w1* + *length w2* − *Suc 0*)
apply *simp*
apply (*subst zpower-zadd-distrib* [*symmetric*])
apply *simp*
apply (*cut-tac lmw*)
apply *arith*
apply (*cut-tac p*)
apply *arith*
done
hence −((*2*::*int*) ^ (*length w1* + *length w2* − *Suc 0*)) ≤ −(*2* ^ *length w1* ∗ *2* ^ (*length w2* − *1*))
by *simp*
also have ... ≤ *?Q*
proof −
from *p*
have *q*: *bv-to-int* (*utos w1*) ∗ *bv-to-int w2* < *0*
by *simp*
thus *?thesis*
proof (*simp add*: *mult-less-0-iff*,*safe*)
assume *bi1*: *0* < *bv-to-int* (*utos w1*)
assume *bi2*: *bv-to-int w2* < *0*
have −*bv-to-int w2* ∗ *bv-to-int* (*utos w1*) ≤ ((*2*::*int*) ^(*length w2* − *1*)) ∗ (*2* ^ *length w1*)
apply (*rule mult-mono*)
using *bv-to-int-lower-range* [*of w2*]
apply *simp*
apply (*simp add*: *bv-to-int-utos*)
using *bv-to-nat-upper-range* [*of w1*]
apply (*simp add*: *int-nat-two-exp*)
apply (*rule zero-le-power*,*simp*)
using *bi1*
apply *simp*
done
hence −*?Q* ≤ ((*2*::*int*) ^*length w1*) ∗ (*2* ^ (*length w2* − *1*))
by (*simp add*: *zmult-ac*)
thus −(((*2*::*int*) ^*length w1*) ∗ (*2* ^ (*length w2* − *Suc 0*))) ≤ *?Q*

```
        by simp
      next
        assume bi1: bv-to-int (utos w1) < 0
        thus -(((2::int)^length w1) * (2 ^ (length w2 - Suc 0))) ≤ ?Q
          by (simp add: bv-to-int-utos)
      qed
    qed
    finally show -(2 ^ (length w1 + length w2 - Suc 0)) ≤ ?Q
      .
  qed
 qed
qed

lemma bv-smult-sym: bv-smult w1 w2 = bv-smult w2 w1
 by (simp add: bv-smult-def zmult-ac)
```

16.7 Structural operations

```
constdefs
  bv-select :: [bit list,nat] => bit
  bv-select w i == w ! (length w - 1 - i)
  bv-chop :: [bit list,nat] => bit list * bit list
  bv-chop w i == let len = length w in (take (len - i) w,drop (len - i) w)
  bv-slice :: [bit list,nat*nat] => bit list
  bv-slice w == λ(b,e). fst (bv-chop (snd (bv-chop w (b+1))) e)

lemma bv-select-rev:
 assumes notnull: n < length w
 shows           bv-select w n = rev w ! n
proof -
 have ∀ n. n < length w --> bv-select w n = rev w ! n
 proof (rule length-induct [of - w],auto simp add: bv-select-def)
   fix xs :: bit list
   fix n
   assume ind: ∀ ys::bit list. length ys < length xs --> (∀ n. n < length ys -->
ys ! (length ys - Suc n) = rev ys ! n)
   assume notx: n < length xs
   show xs ! (length xs - Suc n) = rev xs ! n
   proof (cases xs)
     assume xs = []
     with notx
     show ?thesis
       by simp
   next
     fix y ys
     assume [simp]: xs = y # ys
     show ?thesis
     proof (auto simp add: nth-append)
       assume noty: n < length ys
```

```
      from spec [OF ind,of ys]
      have ∀ n. n < length ys −−> ys ! (length ys − Suc n) = rev ys ! n
        by simp
      hence n < length ys −−> ys ! (length ys − Suc n) = rev ys ! n
        ..
      hence ys ! (length ys − Suc n) = rev ys ! n
        ..
      thus (y # ys) ! (length ys − n) = rev ys ! n
        by (simp add: nth-Cons' noty linorder-not-less [symmetric])
    next
      assume ~ n < length ys
      hence x: length ys ≤ n
        by simp
      from notx
      have n < Suc (length ys)
        by simp
      hence n ≤ length ys
        by simp
      with x
      have length ys = n
        by simp
      thus y = [y] ! (n − length ys)
        by simp
    qed
  qed
 qed
 hence n < length w −−> bv-select w n = rev w ! n
  ..
 thus ?thesis
  ..
qed

lemma bv-chop-append: bv-chop (w1 @ w2) (length w2) = (w1,w2)
 by (simp add: bv-chop-def Let-def)

lemma append-bv-chop-id: fst (bv-chop w l) @ snd (bv-chop w l) = w
 by (simp add: bv-chop-def Let-def)

lemma bv-chop-length-fst [simp]: length (fst (bv-chop w i)) = length w − i
 by (simp add: bv-chop-def Let-def,arith)

lemma bv-chop-length-snd [simp]: length (snd (bv-chop w i)) = min i (length w)
 by (simp add: bv-chop-def Let-def,arith)

lemma bv-slice-length [simp]: [| j ≤ i; i < length w |] ==> length (bv-slice w
(i,j)) = i − j + 1
 by (auto simp add: bv-slice-def,arith)

constdefs
```

length-nat :: *nat* => *nat*
length-nat x == *LEAST n. x* < *2* ^ *n*

lemma *length-nat*: *length* (*nat-to-bv n*) = *length-nat n*
apply (*simp add*: *length-nat-def*)
apply (*rule Least-equality* [*symmetric*])
prefer *2*
apply (*rule length-nat-to-bv-upper-limit*)
apply *arith*
apply (*rule ccontr*)
proof −
assume ~ *n* < *2* ^ *length* (*nat-to-bv n*)
hence *2* ^ *length* (*nat-to-bv n*) ≤ *n*
by *simp*
hence *length* (*nat-to-bv n*) < *length* (*nat-to-bv n*)
by (*rule length-nat-to-bv-lower-limit*)
thus *False*
by *simp*
qed

lemma *length-nat-0* [*simp*]: *length-nat 0* = *0*
by (*simp add*: *length-nat-def Least-equality*)

lemma *length-nat-non0*:
assumes *n0*: *0* < *n*
shows *length-nat n* = *Suc* (*length-nat* (*n div 2*))
apply (*simp add*: *length-nat* [*symmetric*])
apply (*subst nat-to-bv-non0* [*of n*])
apply (*simp-all add*: *n0*)
done

constdefs
length-int :: *int* => *nat*
length-int x == *if 0* < *x then Suc* (*length-nat* (*nat x*)) *else if x* = *0 then 0 else Suc* (*length-nat* (*nat* (−*x* − *1*)))

lemma *length-int*: *length* (*int-to-bv i*) = *length-int i*
proof (*cases 0* < *i*)
assume *i0*: *0* < *i*
hence *length* (*int-to-bv i*) = *length* (*norm-signed* (**0** # *norm-unsigned* (*nat-to-bv* (*nat i*))))
by *simp*
also from *norm-unsigned-result* [*of nat-to-bv* (*nat i*)]
have ... = *Suc* (*length-nat* (*nat i*))
apply *safe*
apply (*simp del*: *norm-unsigned-nat-to-bv*)
apply (*drule norm-empty-bv-to-nat-zero*)
using *prems*
apply *simp*

```
    apply arith
    apply (cases norm-unsigned (nat-to-bv (nat i)))
    apply (drule norm-empty-bv-to-nat-zero [of nat-to-bv (nat i)])
    using prems
    apply simp
    apply simp
    using prems
    apply (simp add: length-nat [symmetric])
    done
  finally show ?thesis
    using i0
    by (simp add: length-int-def)
next
  assume ~ 0 < i
  hence i0: i ≤ 0
    by simp
  show ?thesis
  proof (cases i = 0)
    assume i = 0
    thus ?thesis
      by (simp add: length-int-def)
  next
    assume i ≠ 0
    with i0
    have i0: i < 0
      by simp
    hence length (int-to-bv i) = length (norm-signed (1 # bv-not (norm-unsigned
(nat-to-bv (nat (− i) − 1)))))
      by (simp add: int-to-bv-def nat-diff-distrib)
    also from norm-unsigned-result [of nat-to-bv (nat (− i) − 1)]
    have ... = Suc (length-nat (nat (− i) − 1))
      apply safe
      apply (simp del: norm-unsigned-nat-to-bv)
      apply (drule norm-empty-bv-to-nat-zero [of nat-to-bv (nat (−i) − Suc 0)])
      using prems
      apply simp
      apply (cases − i − 1 = 0)
      apply simp
      apply (simp add: length-nat [symmetric])
      apply (cases norm-unsigned (nat-to-bv (nat (− i) − 1)))
      apply simp
      apply simp
      done
    finally
    show ?thesis
      using i0
      by (simp add: length-int-def nat-diff-distrib del: int-to-bv-lt0)
  qed
qed
```

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

lemma *length-int-gt0*: *0 < i ==> length-int i = Suc* (*length-nat* (*nat i*))
by (*simp add*: *length-int-def*)

lemma *length-int-lt0*: *i < 0 ==> length-int i = Suc* (*length-nat* (*nat* (*− i*) *− 1*))
by (*simp add*: *length-int-def nat-diff-distrib*)

lemma *bv-chopI*: [| *w = w1 @ w2* ; *i = length w2* |] *==> bv-chop w i* = (*w1*,*w2*)
by (*simp add*: *bv-chop-def Let-def*)

lemma *bv-sliceI*: [| *j ≤ i* ; *i < length w* ; *w = w1 @ w2 @ w3* ; *Suc i = length w2 + j* ; *j = length w3* |] *==> bv-slice w* (*i*,*j*) = *w2*
apply (*simp add*: *bv-slice-def*)
apply (*subst bv-chopI* [*of w1 @ w2 @ w3 w1 w2 @ w3*])
apply *simp*
apply *simp*
apply *simp*
apply (*subst bv-chopI* [*of w2 @ w3 w2 w3*],*simp-all*)
done

lemma *bv-slice-bv-slice*:
assumes *ki*: *k ≤ i*
and *ij*: *i ≤ j*
and *jl*: *j ≤ l*
and *lw*: *l < length w*
shows *bv-slice w* (*j*,*i*) = *bv-slice* (*bv-slice w* (*l*,*k*)) (*j−k*,*i−k*)
proof −
def *w1* == *fst* (*bv-chop w* (*Suc l*))
def *w2* == *fst* (*bv-chop* (*snd* (*bv-chop w* (*Suc l*))) (*Suc j*))
def *w3* == *fst* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop w* (*Suc l*))) (*Suc j*))) *i*)
def *w4* == *fst* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop w* (*Suc l*))) (*Suc j*))) *i*)) *k*)
def *w5* == *snd* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop w* (*Suc l*))) (*Suc j*))) *i*)) *k*)

note *w-defs* = *w1-def w2-def w3-def w4-def w5-def*

have *w-def*: *w = w1 @ w2 @ w3 @ w4 @ w5*
by (*simp add*: *w-defs append-bv-chop-id*)

from *ki ij jl lw*
show *?thesis*
apply (*subst bv-sliceI* [**where** *?j = i* **and** *?i = j* **and** *?w = w* **and** *?w1.0 = w1 @ w2* **and** *?w2.0 = w3* **and** *?w3.0 = w4 @ w5*])
apply *simp-all*
apply (*rule w-def*)

apply (*simp add*: *w-defs min-def*)
apply (*simp add*: *w-defs min-def*)
apply (*subst bv-sliceI* [**where** *?j* = *k* **and** *?i* = *l* **and** *?w* = *w* **and** *?w1.0* = *w1* **and** *?w2.0* = *w2* @ *w3* @ *w4* **and** *?w3.0* = *w5*])
apply *simp-all*
apply (*rule w-def*)
apply (*simp add*: *w-defs min-def*)
apply (*simp add*: *w-defs min-def*)
apply (*subst bv-sliceI* [**where** *?j* = *i*−*k* **and** *?i* = *j*−*k* **and** *?w* = *w2* @ *w3* @ *w4* **and** *?w1.0* = *w2* **and** *?w2.0* = *w3* **and** *?w3.0* = *w4*])
apply *simp-all*
apply (*simp-all add*: *w-defs min-def*)
apply *arith+*
done
qed

lemma *bv-to-nat-extend* [*simp*]: *bv-to-nat* (*bv-extend n* **0** *w*) = *bv-to-nat w*
apply (*simp add*: *bv-extend-def*)
apply (*subst bv-to-nat-dist-append*)
apply *simp*
apply (*induct n* − *length w*,*simp-all*)
done

lemma *bv-msb-extend-same* [*simp*]: *bv-msb w* = *b* ==> *bv-msb* (*bv-extend n b w*) = *b*
apply (*simp add*: *bv-extend-def*)
apply (*induct n* − *length w*,*simp-all*)
done

lemma *bv-to-int-extend* [*simp*]:
assumes *a*: *bv-msb w* = *b*
shows *bv-to-int* (*bv-extend n b w*) = *bv-to-int w*
proof (*cases bv-msb w*)
assume [*simp*]: *bv-msb w* = **0**
with *a* **have** [*simp*]: *b* = **0**
by *simp*
show *?thesis*
by (*simp add*: *bv-to-int-def*)
next
assume [*simp*]: *bv-msb w* = **1**
with *a* **have** [*simp*]: *b* = **1**
by *simp*
show *?thesis*
apply (*simp add*: *bv-to-int-def*)
apply (*simp add*: *bv-extend-def*)
apply (*induct n* − *length w*,*simp-all*)
done
qed

lemma *length-nat-mono* [*simp*]: $x \leq y \Longrightarrow$ *length-nat* $x \leq$ *length-nat* y
proof (*rule ccontr*)
  assume *xy*: $x \leq y$
  assume $\sim$ *length-nat* $x \leq$ *length-nat* y
  hence *lxly*: *length-nat* $y <$ *length-nat* x
    by *simp*
  hence *length-nat* $y <$ (*LEAST n*. $x < 2 \hat{} n$)
    by (*simp add*: *length-nat-def*)
  hence $\sim x < 2 \hat{}$ *length-nat* y
    by (*rule not-less-Least*)
  hence *xx*: $2 \hat{}$ *length-nat* $y \leq x$
    by *simp*
  have *yy*: $y < 2 \hat{}$ *length-nat* y
    apply (*simp add*: *length-nat-def*)
    apply (*rule LeastI*)
    apply (*subgoal-tac* $y < 2 \hat{} y$,*assumption*)
    apply (*cases* $0 \leq y$)
    apply (*induct* y,*simp-all*)
    done
  with *xx*
  have $y < x$ **by** *simp*
  with *xy*
  show *False*
    by *simp*
qed

lemma *length-nat-mono-int* [*simp*]: $x \leq y \Longrightarrow$ *length-nat* $x \leq$ *length-nat* y
  apply (*rule length-nat-mono*)
  apply *arith*
  done

lemma *length-nat-pos* [*simp*,*intro*!]: $0 < x \Longrightarrow 0 <$ *length-nat* x
  by (*simp add*: *length-nat-non0*)

lemma *length-int-mono-gt0*: [| $0 \leq x$; $x \leq y$ |] $\Longrightarrow$ *length-int* $x \leq$ *length-int* y
  by (*cases* $x = 0$,*simp-all add*: *length-int-gt0 nat-le-eq-zle*)

lemma *length-int-mono-lt0*: [| $x \leq y$; $y \leq 0$ |] $\Longrightarrow$ *length-int* $y \leq$ *length-int* x
  apply (*cases* $y = 0$,*simp-all add*: *length-int-lt0*)
  apply (*subgoal-tac nat* $(-y) -$ *Suc* $0 \leq$ *nat* $(-x) -$ *Suc* 0)
  apply (*simp add*: *length-nat-mono*)
  apply *arith*
  done

lemmas [*simp*] = *length-nat-non0*

lemma *nat-to-bv* (*number-of Numeral.Pls*) = []
  by *simp*

consts
fast-bv-to-nat-helper :: [*bit list*, *bin*] => *bin*

primrec
fast-bv-to-nat-Nil: *fast-bv-to-nat-helper* [] *bin* = *bin*
fast-bv-to-nat-Cons: *fast-bv-to-nat-helper* (*b*#*bs*) *bin* = *fast-bv-to-nat-helper bs* (*bin BIT* (*bit-case bit.B0 bit.B1 b*))

lemma *fast-bv-to-nat-Cons0*: *fast-bv-to-nat-helper* (**0**#*bs*) *bin* = *fast-bv-to-nat-helper bs* (*bin BIT bit.B0*)
by *simp*

lemma *fast-bv-to-nat-Cons1*: *fast-bv-to-nat-helper* (**1**#*bs*) *bin* = *fast-bv-to-nat-helper bs* (*bin BIT bit.B1*)
by *simp*

lemma *fast-bv-to-nat-def*: *bv-to-nat bs* == *number-of* (*fast-bv-to-nat-helper bs Numeral.Pls*)
proof (*simp add*: *bv-to-nat-def*)
have ∀ *bin*. ¬ (*neg* (*number-of bin* :: *int*)) ⟶ (*foldl* (%*bn b*. *2* ∗ *bn* + *bitval b*) (*number-of bin*) *bs*) = *number-of* (*fast-bv-to-nat-helper bs bin*)
apply (*induct bs*,*simp*)
apply (*case-tac a*,*simp-all*)
done
thus *foldl* (λ*bn b*. *2* ∗ *bn* + *bitval b*) *0 bs* ≡ *number-of* (*fast-bv-to-nat-helper bs Numeral.Pls*)
by (*simp del*: *nat-numeral-0-eq-0 add*: *nat-numeral-0-eq-0* [*symmetric*])
qed

declare *fast-bv-to-nat-Cons* [*simp del*]
declare *fast-bv-to-nat-Cons0* [*simp*]
declare *fast-bv-to-nat-Cons1* [*simp*]

setup *setup-word*

declare *bv-to-nat1* [*simp del*]
declare *bv-to-nat-helper* [*simp del*]

constdefs
bv-mapzip :: [*bit* => *bit* => *bit*,*bit list*, *bit list*] => *bit list*
bv-mapzip f w1 w2 == *let g* = *bv-extend* (*max* (*length w1*) (*length w2*)) **0**
in map (*split f*) (*zip* (*g w1*) (*g w2*))

lemma *bv-length-bv-mapzip* [*simp*]: *length* (*bv-mapzip f w1 w2*) = *max* (*length w1*) (*length w2*)
by (*simp add*: *bv-mapzip-def Let-def split*: *split-max*)

lemma *bv-mapzip-Nil* [*simp*]: *bv-mapzip f* [] [] = []
by (*simp add*: *bv-mapzip-def Let-def*)

lemma *bv-mapzip-Cons* [*simp*]: *length w1 = length w2 ==> bv-mapzip f* (*x#w1*) (*y#w2*) = *f x y # bv-mapzip f w1 w2*
by (*simp add*: *bv-mapzip-def Let-def*)

end

17 Zorn: Zorn's Lemma

theory *Zorn*
imports *Main*
begin

The lemma and section numbers refer to an unpublished article [1].

constdefs
chain :: *'a set set => 'a set set set*
chain S == {*F*. *F* ⊆ *S* & (∀ *x* ∈ *F*. ∀ *y* ∈ *F*. *x* ⊆ *y* | *y* ⊆ *x*)}

super :: [*'a set set*,*'a set set*] => *'a set set set*
super S c == {*d*. *d* ∈ *chain S* & *c* ⊂ *d*}

maxchain :: *'a set set => 'a set set set*
maxchain S == {*c*. *c* ∈ *chain S* & *super S c* = {}}

succ :: [*'a set set*,*'a set set*] => *'a set set*
succ S c ==
if c ∉ *chain S* | *c* ∈ *maxchain S*
then c else SOME c'. *c'* ∈ *super S c*

consts
TFin :: *'a set set => 'a set set set*

inductive *TFin S*
intros
succI: *x* ∈ *TFin S* ==> *succ S x* ∈ *TFin S*
Pow-UnionI: *Y* ∈ *Pow*(*TFin S*) ==> *Union*(*Y*) ∈ *TFin S*
monos *Pow-mono*

17.1 Mathematical Preamble

lemma *Union-lemma0*:
(∀ *x* ∈ *C*. *x* ⊆ *A* | *B* ⊆ *x*) ==> *Union*(*C*)<=*A* | *B* ⊆ *Union*(*C*)
by *blast*

This is theorem *increasingD2* of ZF/Zorn.thy

lemma *Abrial-axiom1*: $x \subseteq succ\ S\ x$
apply (*unfold succ-def*)
apply (*rule split-if* [*THEN iffD2*])
apply (*auto simp add*: *super-def maxchain-def psubset-def*)
apply (*rule swap*, *assumption*)
apply (*rule someI2*, *blast*+)
done

lemmas *TFin-UnionI* = *TFin.Pow-UnionI* [*OF PowI*]

lemma *TFin-induct*:
$[|\ n \in TFin\ S;$
$!!x.\ [|\ x \in TFin\ S;\ P(x)\ |] ==> P(succ\ S\ x);$
$!!Y.\ [|\ Y \subseteq TFin\ S;\ Ball\ Y\ P\ |] ==> P(Union\ Y)\ |]$
$==> P(n)$
apply (*erule TFin.induct*)
apply *blast*+
done

lemma *succ-trans*: $x \subseteq y ==> x \subseteq succ\ S\ y$
apply (*erule subset-trans*)
apply (*rule Abrial-axiom1*)
done

Lemma 1 of section 3.1

lemma *TFin-linear-lemma1*:
$[|\ n \in TFin\ S;\ \ m \in TFin\ S;$
$\forall x \in TFin\ S.\ x \subseteq m\ --> x = m \mid succ\ S\ x \subseteq m$
$|] ==> n \subseteq m \mid succ\ S\ m \subseteq n$
apply (*erule TFin-induct*)
apply (*erule-tac* [*2*] *Union-lemma0*)
apply (*blast del*: *subsetI intro*: *succ-trans*)
done

Lemma 2 of section 3.2

lemma *TFin-linear-lemma2*:
$m \in TFin\ S ==> \forall n \in TFin\ S.\ n \subseteq m\ --> n=m \mid succ\ S\ n \subseteq m$
apply (*erule TFin-induct*)
apply (*rule impI* [*THEN ballI*])

case split using *TFin-linear-lemma1*

apply (*rule-tac n1* = *n* **and** *m1* = *x* **in** *TFin-linear-lemma1* [*THEN disjE*],
assumption+)
apply (*drule-tac x* = *n* **in** *bspec*, *assumption*)
apply (*blast del*: *subsetI intro*: *succ-trans*, *blast*)

second induction step

apply (*rule impI* [*THEN ballI*])
apply (*rule Union-lemma0* [*THEN disjE*])

```
  apply (rule-tac [3] disjI2)
  prefer 2 apply blast
 apply (rule ballI)
 apply (rule-tac n1 = n and m1 = x in TFin-linear-lemma1 [THEN disjE],
   assumption+, auto)
apply (blast intro!: Abrial-axiom1 [THEN subsetD])
done
```

Re-ordering the premises of Lemma 2

```
lemma TFin-subsetD:
    [| n ⊆ m;  m ∈ TFin S;  n ∈ TFin S |] ==> n=m | succ S n ⊆ m
 by (rule TFin-linear-lemma2 [rule-format])
```

Consequences from section 3.3 – Property 3.2, the ordering is total

```
lemma TFin-subset-linear: [| m ∈ TFin S;  n ∈ TFin S|] ==> n ⊆ m | m ⊆ n
 apply (rule disjE)
  apply (rule TFin-linear-lemma1 [OF - -TFin-linear-lemma2])
    apply (assumption+, erule disjI2)
 apply (blast del: subsetI
   intro: subsetI Abrial-axiom1 [THEN subset-trans])
 done
```

Lemma 3 of section 3.3

```
lemma eq-succ-upper: [| n ∈ TFin S;  m ∈ TFin S;  m = succ S m |] ==> n ⊆
m
 apply (erule TFin-induct)
  apply (drule TFin-subsetD)
    apply (assumption+, force, blast)
 done
```

Property 3.3 of section 3.3

```
lemma equal-succ-Union: m ∈ TFin S ==> (m = succ S m) = (m = Union(TFin
S))
 apply (rule iffI)
  apply (rule Union-upper [THEN equalityI])
   apply (rule-tac [2] eq-succ-upper [THEN Union-least])
     apply (assumption+)
 apply (erule ssubst)
 apply (rule Abrial-axiom1 [THEN equalityI])
 apply (blast del: subsetI intro: subsetI TFin-UnionI TFin.succI)
 done
```

17.2 Hausdorff's Theorem: Every Set Contains a Maximal Chain.

NB: We assume the partial ordering is ⊆, the subset relation!

```
lemma empty-set-mem-chain: ({} :: 'a set set) ∈ chain S
 by (unfold chain-def) auto
```

lemma *super-subset-chain*: *super S c* ⊆ *chain S*
by (*unfold super-def*) *blast*

lemma *maxchain-subset-chain*: *maxchain S* ⊆ *chain S*
by (*unfold maxchain-def*) *blast*

lemma *mem-super-Ex*: *c* ∈ *chain S* − *maxchain S* ==> *?d*. *d* ∈ *super S c*
by (*unfold super-def maxchain-def*) *auto*

lemma *select-super*: *c* ∈ *chain S* − *maxchain S* ==>
(ε *c'*. *c'*: *super S c*): *super S c*
apply (*erule mem-super-Ex* [*THEN exE*])
apply (*rule someI2*, *auto*)
done

lemma *select-not-equals*: *c* ∈ *chain S* − *maxchain S* ==>
(ε *c'*. *c'*: *super S c*) ≠ *c*
apply (*rule notI*)
apply (*drule select-super*)
apply (*simp add*: *super-def psubset-def*)
done

lemma *succI3*: *c* ∈ *chain S* − *maxchain S* ==> *succ S c* = (ε *c'*. *c'*: *super S c*)
by (*unfold succ-def*) (*blast intro*!: *if-not-P*)

lemma *succ-not-equals*: *c* ∈ *chain S* − *maxchain S* ==> *succ S c* ≠ *c*
apply (*frule succI3*)
apply (*simp* (*no-asm-simp*))
apply (*rule select-not-equals*, *assumption*)
done

lemma *TFin-chain-lemma4*: *c* ∈ *TFin S* ==> (*c* :: *'a set set*): *chain S*
apply (*erule TFin-induct*)
apply (*simp add*: *succ-def select-super* [*THEN super-subset-chain*[*THEN subsetD*]])
apply (*unfold chain-def*)
apply (*rule CollectI*, *safe*)
apply (*drule bspec*, *assumption*)
apply (*rule-tac* [*2*] *m1* = *Xa* **and** *n1* = *X* **in** *TFin-subset-linear* [*THEN disjE*], *blast*+)
done

theorem *Hausdorff*: ∃ *c*. (*c* :: *'a set set*): *maxchain S*
apply (*rule-tac x* = *Union* (*TFin S*) **in** *exI*)
apply (*rule classical*)
apply (*subgoal-tac succ S* (*Union* (*TFin S*)) = *Union* (*TFin S*))
prefer *2*
apply (*blast intro*!: *TFin-UnionI equal-succ-Union* [*THEN iffD2*, *symmetric*])

apply (*cut-tac subset-refl* [*THEN TFin-UnionI*, *THEN TFin-chain-lemma4*])
apply (*drule DiffI* [*THEN succ-not-equals*], *blast+*)
done

17.3 Zorn's Lemma: If All Chains Have Upper Bounds Then There Is a Maximal Element

lemma *chain-extend*:
[| $c \in$ *chain* S; $z \in S$;
$\forall x \in c.\ x <= (z :: 'a\ set)$ |] ==> $\{z\}$ *Un* $c \in$ *chain* S
by (*unfold chain-def*) *blast*

lemma *chain-Union-upper*: [| $c \in$ *chain* S; $x \in c$ |] ==> $x \subseteq$ *Union*(c)
by (*unfold chain-def*) *auto*

lemma *chain-ball-Union-upper*: $c \in$ *chain* S ==> $\forall x \in c.\ x \subseteq$ *Union*(c)
by (*unfold chain-def*) *auto*

lemma *maxchain-Zorn*:
[| $c \in$ *maxchain* S; $u \in S$; *Union*(c) $\subseteq u$ |] ==> *Union*(c) = u
apply (*rule ccontr*)
apply (*simp add*: *maxchain-def*)
apply (*erule conjE*)
apply (*subgoal-tac* ($\{u\}$ *Un* c) $\in$ *super* S c)
apply *simp*
apply (*unfold super-def psubset-def*)
apply (*blast intro*: *chain-extend dest*: *chain-Union-upper*)
done

theorem *Zorn-Lemma*:
$\forall c \in$ *chain* S. *Union*(c): S ==> $\exists y \in S.\ \forall z \in S.\ y \subseteq z$ --> $y = z$
apply (*cut-tac Hausdorff maxchain-subset-chain*)
apply (*erule exE*)
apply (*drule subsetD*, *assumption*)
apply (*drule bspec*, *assumption*)
apply (*rule-tac* x = *Union* (c) **in** *bexI*)
apply (*rule ballI*, *rule impI*)
apply (*blast dest*!: *maxchain-Zorn*, *assumption*)
done

17.4 Alternative version of Zorn's Lemma

lemma *Zorn-Lemma2*:
$\forall c \in$ *chain* S. $\exists y \in S.\ \forall x \in c.\ x \subseteq y$
==> $\exists y \in S.\ \forall x \in S.\ (y :: 'a\ set) \subseteq x$ --> $y = x$
apply (*cut-tac Hausdorff maxchain-subset-chain*)
apply (*erule exE*)
apply (*drule subsetD*, *assumption*)
apply (*drule bspec*, *assumption*, *erule bexE*)

apply (*rule-tac* $x = y$ **in** *bexI*)
prefer *2* **apply** *assumption*
apply *clarify*
apply (*rule ccontr*)
apply (*frule-tac* $z = x$ **in** *chain-extend*)
apply (*assumption*, *blast*)
apply (*unfold maxchain-def super-def psubset-def*)
apply (*blast elim*!: *equalityCE*)
done

Various other lemmas

lemma *chainD*: $[|\ c \in chain\ S;\ x \in c;\ y \in c\ |] ==> x \subseteq y \mid y \subseteq x$
by (*unfold chain-def*) *blast*

lemma *chainD2*: !!$(c :: 'a\ set\ set).\ c \in chain\ S ==> c \subseteq S$
by (*unfold chain-def*) *blast*

end

18 Product-ord: Order on product types

theory *Product-ord*
imports *Main*
begin

instance $*$ :: (*ord*,*ord*) *ord* **..**

defs (**overloaded**)
prod-le-def: $(x \leq y) \equiv (fst\ x < fst\ y) \mid (fst\ x = fst\ y\ \&\ snd\ x \leq snd\ y)$
prod-less-def: $(x < y) \equiv (fst\ x < fst\ y) \mid (fst\ x = fst\ y\ \&\ snd\ x < snd\ y)$

lemmas *prod-ord-defs* = *prod-less-def prod-le-def*

instance $*$ :: (*order*,*order*) *order*
apply (*intro-classes*, *unfold prod-ord-defs*)
by (*auto intro*: *order-less-trans*)

instance $*$:: (*linorder*,*linorder*)*linorder*
by (*intro-classes*, *unfold prod-le-def*, *auto*)

end

19 Char-ord: Order on characters

theory *Char-ord*
imports *Product-ord*

begin

Conversions between nibbles and integers in [0..15].

consts
nibble-to-int:: *nibble* ⇒ *int*
int-to-nibble:: *int* ⇒ *nibble*

primrec
nibble-to-int Nibble0 = *0*
nibble-to-int Nibble1 = *1*
nibble-to-int Nibble2 = *2*
nibble-to-int Nibble3 = *3*
nibble-to-int Nibble4 = *4*
nibble-to-int Nibble5 = *5*
nibble-to-int Nibble6 = *6*
nibble-to-int Nibble7 = *7*
nibble-to-int Nibble8 = *8*
nibble-to-int Nibble9 = *9*
nibble-to-int NibbleA = *10*
nibble-to-int NibbleB = *11*
nibble-to-int NibbleC = *12*
nibble-to-int NibbleD = *13*
nibble-to-int NibbleE = *14*
nibble-to-int NibbleF = *15*

defs
int-to-nibble-def:
int-to-nibble x ≡ (*let y* = *x mod 16 in*
if y = *0 then Nibble0 else*
if y = *1 then Nibble1 else*
if y = *2 then Nibble2 else*
if y = *3 then Nibble3 else*
if y = *4 then Nibble4 else*
if y = *5 then Nibble5 else*
if y = *6 then Nibble6 else*
if y = *7 then Nibble7 else*
if y = *8 then Nibble8 else*
if y = *9 then Nibble9 else*
if y = *10 then NibbleA else*
if y = *11 then NibbleB else*
if y = *12 then NibbleC else*
if y = *13 then NibbleD else*
if y = *14 then NibbleE else*
NibbleF)

lemma *int-to-nibble-nibble-to-int*: *int-to-nibble*(*nibble-to-int x*) = *x*
by (*cases x*) (*auto simp*: *int-to-nibble-def Let-def*)

lemma *inj-nibble-to-int*: *inj nibble-to-int*

by (*rule inj-on-inverseI*) (*rule int-to-nibble-nibble-to-int*)

lemmas *nibble-to-int-eq* = *inj-nibble-to-int* [*THEN inj-eq*]

lemma *nibble-to-int-ge-0*: *0* ≤ *nibble-to-int x*
by (*cases x*) *auto*

lemma *nibble-to-int-less-16*: *nibble-to-int x* < *16*
by (*cases x*) *auto*

Conversion between chars and int pairs.

consts
char-to-int-pair :: *char* ⇒ *int* × *int*
primrec
char-to-int-pair (*Char a b*) = (*nibble-to-int a*, *nibble-to-int b*)

lemma *inj-char-to-int-pair*: *inj char-to-int-pair*
apply (*rule inj-onI*)
apply (*case-tac x*, *case-tac y*)
apply (*auto simp*: *nibble-to-int-eq*)
done

lemmas *char-to-int-pair-eq* = *inj-char-to-int-pair* [*THEN inj-eq*]

Instantiation of order classes

instance *char* :: *ord* **..**

defs (**overloaded**)
char-le-def: *c* ≤ *d* ≡ (*char-to-int-pair c* ≤ *char-to-int-pair d*)
char-less-def: *c* < *d* ≡ (*char-to-int-pair c* < *char-to-int-pair d*)

lemmas *char-ord-defs* = *char-less-def char-le-def*

instance *char* :: *order*
apply *intro-classes*
apply (*unfold char-ord-defs*)
apply (*auto simp*: *char-to-int-pair-eq order-less-le*)
done

instance *char*::*linorder*
apply *intro-classes*
apply (*unfold char-le-def*)
apply *auto*
done

end

20 Commutative-Ring: Proving equalities in commutative rings

theory *Commutative-Ring*
imports *Main*
uses (*comm-ring.ML*)
begin

Syntax of multivariate polynomials (pol) and polynomial expressions.

datatype *'a pol* =
  Pc 'a
| *Pinj nat 'a pol*
| *PX 'a pol nat 'a pol*

datatype *'a polex* =
  Pol 'a pol
| *Add 'a polex 'a polex*
| *Sub 'a polex 'a polex*
| *Mul 'a polex 'a polex*
| *Pow 'a polex nat*
| *Neg 'a polex*

Interpretation functions for the shadow syntax.

consts
  Ipol :: *'a*::{*comm-ring*,*recpower*} *list* ⇒ *'a pol* ⇒ *'a*
  Ipolex :: *'a*::{*comm-ring*,*recpower*} *list* ⇒ *'a polex* ⇒ *'a*

primrec
  Ipol l (*Pc c*) = *c*
  Ipol l (*Pinj i P*) = *Ipol* (*drop i l*) *P*
  Ipol l (*PX P x Q*) = *Ipol l P* ∗ (*hd l*) ^*x* + *Ipol* (*drop 1 l*) *Q*

primrec
  Ipolex l (*Pol P*) = *Ipol l P*
  Ipolex l (*Add P Q*) = *Ipolex l P* + *Ipolex l Q*
  Ipolex l (*Sub P Q*) = *Ipolex l P* − *Ipolex l Q*
  Ipolex l (*Mul P Q*) = *Ipolex l P* ∗ *Ipolex l Q*
  Ipolex l (*Pow p n*) = *Ipolex l p* ^ *n*
  Ipolex l (*Neg P*) = − *Ipolex l P*

Create polynomial normalized polynomials given normalized inputs.

constdefs
  mkPinj :: *nat* ⇒ *'a pol* ⇒ *'a pol*
  mkPinj x P ≡ (*case P of*
    Pc c ⇒ *Pc c* |
    Pinj y P ⇒ *Pinj* (*x* + *y*) *P* |
    PX p1 y p2 ⇒ *Pinj x P*)

constdefs
mkPX :: *'a*::{*comm-ring*,*recpower*} *pol* ⇒ *nat* ⇒ *'a pol* ⇒ *'a pol*
mkPX P i Q == (*case P of*
Pc c ⇒ (*if* (*c* = *0*) *then* (*mkPinj 1 Q*) *else* (*PX P i Q*)) |
Pinj j R ⇒ *PX P i Q* |
PX P2 i2 Q2 ⇒ (*if* (*Q2* = (*Pc 0*)) *then* (*PX P2* (*i*+*i2*) *Q*) *else* (*PX P i Q*))
)

Defining the basic ring operations on normalized polynomials

consts
add :: *'a*::{*comm-ring*,*recpower*} *pol* × *'a pol* ⇒ *'a pol*
mul :: *'a*::{*comm-ring*,*recpower*} *pol* × *'a pol* ⇒ *'a pol*
neg :: *'a*::{*comm-ring*,*recpower*} *pol* ⇒ *'a pol*
sqr :: *'a*::{*comm-ring*,*recpower*} *pol* ⇒ *'a pol*
pow :: *'a*::{*comm-ring*,*recpower*} *pol* × *nat* ⇒ *'a pol*

Addition

recdef *add measure* (λ(*x*, *y*). *size x* + *size y*)
add (*Pc a*, *Pc b*) = *Pc* (*a* + *b*)
add (*Pc c*, *Pinj i P*) = *Pinj i* (*add* (*P*, *Pc c*))
add (*Pinj i P*, *Pc c*) = *Pinj i* (*add* (*P*, *Pc c*))
add (*Pc c*, *PX P i Q*) = *PX P i* (*add* (*Q*, *Pc c*))
add (*PX P i Q*, *Pc c*) = *PX P i* (*add* (*Q*, *Pc c*))
add (*Pinj x P*, *Pinj y Q*) =
(*if x*=*y then mkPinj x* (*add* (*P*, *Q*))
else (*if x*>*y then mkPinj y* (*add* (*Pinj* (*x*−*y*) *P*, *Q*))
else mkPinj x (*add* (*Pinj* (*y*−*x*) *Q*, *P*))))
add (*Pinj x P*, *PX Q y R*) =
(*if x*=*0 then add*(*P*, *PX Q y R*)
else (*if x*=*1 then PX Q y* (*add* (*R*, *P*))
else PX Q y (*add* (*R*, *Pinj* (*x* − *1*) *P*))))
add (*PX P x R*, *Pinj y Q*) =
(*if y*=*0 then add*(*PX P x R*, *Q*)
else (*if y*=*1 then PX P x* (*add* (*R*, *Q*))
else PX P x (*add* (*R*, *Pinj* (*y* − *1*) *Q*))))
add (*PX P1 x P2*, *PX Q1 y Q2*) =
(*if x*=*y then mkPX* (*add* (*P1*, *Q1*)) *x* (*add* (*P2*, *Q2*))
else (*if x*>*y then mkPX* (*add* (*PX P1* (*x*−*y*) (*Pc 0*), *Q1*)) *y* (*add* (*P2*,*Q2*))
else mkPX (*add* (*PX Q1* (*y*−*x*) (*Pc 0*), *P1*)) *x* (*add* (*P2*,*Q2*))))

Multiplication

recdef *mul measure* (λ(*x*, *y*). *size x* + *size y*)
mul (*Pc a*, *Pc b*) = *Pc* (*a*∗*b*)
mul (*Pc c*, *Pinj i P*) = (*if c*=*0 then Pc 0 else mkPinj i* (*mul* (*P*, *Pc c*)))
mul (*Pinj i P*, *Pc c*) = (*if c*=*0 then Pc 0 else mkPinj i* (*mul* (*P*, *Pc c*)))
mul (*Pc c*, *PX P i Q*) =
(*if c*=*0 then Pc 0 else mkPX* (*mul* (*P*, *Pc c*)) *i* (*mul* (*Q*, *Pc c*)))
mul (*PX P i Q*, *Pc c*) =
(*if c*=*0 then Pc 0 else mkPX* (*mul* (*P*, *Pc c*)) *i* (*mul* (*Q*, *Pc c*)))

mul (Pinj x P, Pinj y Q) =
(if x=y then mkPinj x (mul (P, Q))
else (if x>y then mkPinj y (mul (Pinj (x−y) P, Q))
else mkPinj x (mul (Pinj (y−x) Q, P))))
mul (Pinj x P, PX Q y R) =
(if x=0 then mul(P, PX Q y R)
else (if x=1 then mkPX (mul (Pinj x P, Q)) y (mul (R, P))
else mkPX (mul (Pinj x P, Q)) y (mul (R, Pinj (x − 1) P))))
mul (PX P x R, Pinj y Q) =
(if y=0 then mul(PX P x R, Q)
else (if y=1 then mkPX (mul (Pinj y Q, P)) x (mul (R, Q))
else mkPX (mul (Pinj y Q, P)) x (mul (R, Pinj (y − 1) Q))))
mul (PX P1 x P2, PX Q1 y Q2) =
add (mkPX (mul (P1, Q1)) (x+y) (mul (P2, Q2)),
add (mkPX (mul (P1, mkPinj 1 Q2)) x (Pc 0), mkPX (mul (Q1, mkPinj 1 P2)) y (Pc 0)))
(**hints** *simp add*: *mkPinj-def split*: *pol.split*)

Negation

primrec
neg (Pc c) = Pc (−c)
neg (Pinj i P) = Pinj i (neg P)
neg (PX P x Q) = PX (neg P) x (neg Q)

Substraction

constdefs
sub :: 'a::{comm-ring,recpower} pol ⇒ 'a pol ⇒ 'a pol
sub p q ≡ add (p, neg q)

Square for Fast Exponentation

primrec
sqr (Pc c) = Pc (c ∗ c)
sqr (Pinj i P) = mkPinj i (sqr P)
sqr (PX A x B) = add (mkPX (sqr A) (x + x) (sqr B),
mkPX (mul (mul (Pc (1 + 1), A), mkPinj 1 B)) x (Pc 0))

Fast Exponentation

lemma *pow-wf*:*odd n ⟹ (n::nat) div 2 < n* **by** (*cases n*) *auto*
recdef *pow measure* (λ(*x*, *y*). *y*)
pow (p, 0) = Pc 1
pow (p, n) = (if even n then (pow (sqr p, n div 2)) else mul (p, pow (sqr p, n div 2)))
(**hints** *simp add*: *pow-wf*)

lemma *pow-if*:
pow (p,n) =
(if n = 0 then Pc 1 else if even n then pow (sqr p, n div 2)
else mul (p, pow (sqr p, n div 2)))

by (*cases n*) *simp-all*

Normalization of polynomial expressions

consts *norm* :: *'a*::{*comm-ring*,*recpower*} *polex* ⇒ *'a pol*
primrec
norm (*Pol P*) = *P*
norm (*Add P Q*) = *add* (*norm P*, *norm Q*)
norm (*Sub p q*) = *sub* (*norm p*) (*norm q*)
norm (*Mul P Q*) = *mul* (*norm P*, *norm Q*)
norm (*Pow p n*) = *pow* (*norm p*, *n*)
norm (*Neg P*) = *neg* (*norm P*)

mkPinj preserve semantics

lemma *mkPinj-ci*: *Ipol l* (*mkPinj a B*) = *Ipol l* (*Pinj a B*)
by (*induct B*) (*auto simp add*: *mkPinj-def ring-eq-simps*)

mkPX preserves semantics

lemma *mkPX-ci*: *Ipol l* (*mkPX A b C*) = *Ipol l* (*PX A b C*)
by (*cases A*) (*auto simp add*: *mkPX-def mkPinj-ci power-add ring-eq-simps*)

Correctness theorems for the implemented operations

Negation

lemma *neg-ci*: ⋀*l*. *Ipol l* (*neg P*) = −(*Ipol l P*)
by (*induct P*) *auto*

Addition

lemma *add-ci*: ⋀*l*. *Ipol l* (*add* (*P*, *Q*)) = *Ipol l P* + *Ipol l Q*
proof (*induct P Q rule*: *add.induct*)
case (*6 x P y Q*)
show *?case*
proof (*rule linorder-cases*)
assume *x* < *y*
with *6* **show** *?case* **by** (*simp add*: *mkPinj-ci ring-eq-simps*)
next
assume *x* = *y*
with *6* **show** *?case* **by** (*simp add*: *mkPinj-ci*)
next
assume *x* > *y*
with *6* **show** *?case* **by** (*simp add*: *mkPinj-ci ring-eq-simps*)
qed
next
case (*7 x P Q y R*)
have *x* = *0* ∨ *x* = *1* ∨ *x* > *1* **by** *arith*
moreover
{ assume *x* = *0* **with** *7* **have** *?case* **by** *simp* **}**
moreover
{ assume *x* = *1* **with** *7* **have** *?case* **by** (*simp add*: *ring-eq-simps*) **}**

moreover
{ assume *x > 1* **from** *7* **have** *?case* **by** (*cases x*) *simp-all* **}**
ultimately show *?case* **by** *blast*
next
case (*8 P x R y Q*)
have *y = 0 ∨ y = 1 ∨ y > 1* **by** *arith*
moreover
{ assume *y = 0* **with** *8* **have** *?case* **by** *simp* **}**
moreover
{ assume *y = 1* **with** *8* **have** *?case* **by** *simp* **}**
moreover
{ assume *y > 1* **with** *8* **have** *?case* **by** *simp* **}**
ultimately show *?case* **by** *blast*
next
case (*9 P1 x P2 Q1 y Q2*)
show *?case*
proof (*rule linorder-cases*)
assume *a*: *x < y* **hence** *EX d. d + x = y* **by** *arith*
with *9 a* **show** *?case* **by** (*auto simp add*: *mkPX-ci power-add ring-eq-simps*)
next
assume *a*: *y < x* **hence** *EX d. d + y = x* **by** *arith*
with *9 a* **show** *?case* **by** (*auto simp add*: *power-add mkPX-ci ring-eq-simps*)
next
assume *x = y*
with *9* **show** *?case* **by** (*simp add*: *mkPX-ci ring-eq-simps*)
qed
qed (*auto simp add*: *ring-eq-simps*)

Multiplication

lemma *mul-ci*: *⋀l. Ipol l* (*mul* (*P*, *Q*)) = *Ipol l P ∗ Ipol l Q*
by (*induct P Q rule*: *mul.induct*)
(*simp-all add*: *mkPX-ci mkPinj-ci ring-eq-simps add-ci power-add*)

Substraction

lemma *sub-ci*: *Ipol l* (*sub p q*) = *Ipol l p − Ipol l q*
by (*simp add*: *add-ci neg-ci sub-def*)

Square

lemma *sqr-ci*:*⋀ls. Ipol ls* (*sqr p*) = *Ipol ls p ∗ Ipol ls p*
by (*induct p*) (*simp-all add*: *add-ci mkPinj-ci mkPX-ci mul-ci ring-eq-simps power-add*)

Power

lemma *even-pow*:*even n ⟹ pow* (*p*, *n*) = *pow* (*sqr p*, *n div 2*) **by** (*induct n*) *simp-all*

lemma *pow-ci*: *⋀p. Ipol ls* (*pow* (*p*, *n*)) = (*Ipol ls p*) *^ n*
proof (*induct n rule*: *nat-less-induct*)

```
  case (1 k)
  have two:2 = Suc (Suc 0) by simp
  show ?case
  proof (cases k)
    case (Suc l)
    show ?thesis
    proof cases
      assume EL: even l
      have Suc l div 2 = l div 2
        by (simp add: nat-number even-nat-plus-one-div-two [OF EL])
      moreover
      from Suc have l < k by simp
      with 1 have ∀ p. Ipol ls (pow (p, l)) = Ipol ls p ^ l by simp
      moreover
      note Suc EL even-nat-plus-one-div-two [OF EL]
      ultimately show ?thesis by (auto simp add: mul-ci power-Suc even-pow)
    next
      assume OL: odd l
      with prems have ⟦∀ m<Suc l. ∀ p. Ipol ls (pow (p, m)) = Ipol ls p ^ m; k =
Suc l; odd l⟧ ⟹ ∀ p. Ipol ls (sqr p) ^ (Suc l div 2) = Ipol ls p ^ Suc l
      proof(cases l)
        case (Suc w)
        from prems have EW: even w by simp
        from two have two-times:(2 * (w div 2))= w
          by (simp only: even-nat-div-two-times-two[OF EW])
        have A: ⋀p. (Ipol ls p * Ipol ls p) = (Ipol ls p) ^ (Suc (Suc 0))
          by (simp add: power-Suc)
       from A two [symmetric] have ALL p.(Ipol ls p * Ipol ls p) = (Ipol ls p) ^ 2
          by simp
        with prems show ?thesis
        by (auto simp add: power-mult[symmetric, of - 2 -] two-times mul-ci sqr-ci)
      qed simp
      with prems show ?thesis by simp
    qed
  next
    case 0
    then show ?thesis by simp
  qed
qed
```

Normalization preserves semantics

```
lemma norm-ci:Ipolex l Pe = Ipol l (norm Pe)
  by (induct Pe) (simp-all add: add-ci sub-ci mul-ci neg-ci pow-ci)
```

Reflection lemma: Key to the (incomplete) decision procedure

```
lemma norm-eq:
  assumes eq: norm P1  = norm P2
  shows Ipolex l P1 = Ipolex l P2
proof −
```

from *eq* **have** *Ipol l* (*norm P1*) = *Ipol l* (*norm P2*) **by** *simp*
thus *?thesis* **by** (*simp only*: *norm-ci*)
qed

Code generation

use *comm-ring.ML*
setup *CommRing.setup*

end

21 List-Prefix: List prefixes and postfixes

theory *List-Prefix*
imports *Main*
begin

21.1 Prefix order on lists

instance *list* :: (*type*) *ord* **..**

defs (**overloaded**)
prefix-def: $xs \le ys == \exists zs.\ ys = xs\ @\ zs$
strict-prefix-def: $xs < ys == xs \le ys \wedge xs \neq (ys::'a\ list)$

instance *list* :: (*type*) *order*
by *intro-classes* (*auto simp add*: *prefix-def strict-prefix-def*)

lemma *prefixI* [*intro?*]: $ys = xs\ @\ zs ==> xs \le ys$
by (*unfold prefix-def*) *blast*

lemma *prefixE* [*elim?*]: $xs \le ys ==> (!!zs.\ ys = xs\ @\ zs ==> C) ==> C$
by (*unfold prefix-def*) *blast*

lemma *strict-prefixI'* [*intro?*]: $ys = xs\ @\ z\ \#\ zs ==> xs < ys$
by (*unfold strict-prefix-def prefix-def*) *blast*

lemma *strict-prefixE'* [*elim?*]:
assumes *lt*: $xs < ys$
and *r*: $!!z\ zs.\ ys = xs\ @\ z\ \#\ zs ==> C$
shows *C*
proof −
from *lt* **obtain** *us* **where** $ys = xs\ @\ us$ **and** $xs \neq ys$
by (*unfold strict-prefix-def prefix-def*) *blast*
with *r* **show** *?thesis* **by** (*auto simp add*: *neq-Nil-conv*)
qed

lemma *strict-prefixI* [*intro?*]: $xs \le ys ==> xs \neq ys ==> xs < (ys::'a\ list)$
by (*unfold strict-prefix-def*) *blast*

lemma *strict-prefixE* [*elim?*]:
  $xs < ys \Longrightarrow (xs \leq ys \Longrightarrow xs \neq (ys::'a\ list) \Longrightarrow C) \Longrightarrow C$
 by (*unfold strict-prefix-def*) *blast*

21.2 Basic properties of prefixes

theorem *Nil-prefix* [*iff*]: $[] \leq xs$
 by (*simp add*: *prefix-def*)

theorem *prefix-Nil* [*simp*]: $(xs \leq []) = (xs = [])$
 by (*induct xs*) (*simp-all add*: *prefix-def*)

lemma *prefix-snoc* [*simp*]: $(xs \leq ys @ [y]) = (xs = ys @ [y] \vee xs \leq ys)$
proof
 assume $xs \leq ys @ [y]$
 then obtain *zs* **where** *zs*: $ys @ [y] = xs @ zs$ **..**
 show $xs = ys @ [y] \vee xs \leq ys$
 proof (*cases zs rule*: *rev-cases*)
  assume $zs = []$
  with *zs* **have** $xs = ys @ [y]$ **by** *simp*
  thus *?thesis* **..**
 next
  fix *z zs'* **assume** $zs = zs' @ [z]$
  with *zs* **have** $ys = xs @ zs'$ **by** *simp*
  hence $xs \leq ys$ **..**
  thus *?thesis* **..**
 qed
next
 assume $xs = ys @ [y] \vee xs \leq ys$
 thus $xs \leq ys @ [y]$
 proof
  assume $xs = ys @ [y]$
  thus *?thesis* **by** *simp*
 next
  assume $xs \leq ys$
  then obtain *zs* **where** $ys = xs @ zs$ **..**
  hence $ys @ [y] = xs @ (zs @ [y])$ **by** *simp*
  thus *?thesis* **..**
 qed
qed

lemma *Cons-prefix-Cons* [*simp*]: $(x \# xs \leq y \# ys) = (x = y \wedge xs \leq ys)$
 by (*auto simp add*: *prefix-def*)

lemma *same-prefix-prefix* [*simp*]: $(xs @ ys \leq xs @ zs) = (ys \leq zs)$
 by (*induct xs*) *simp-all*

lemma *same-prefix-nil* [*iff*]: $(xs @ ys \leq xs) = (ys = [])$

proof −
have (*xs* @ *ys* ≤ *xs* @ []) = (*ys* ≤ []) **by** (*rule same-prefix-prefix*)
thus *?thesis* **by** *simp*
qed

lemma *prefix-prefix* [*simp*]: *xs* ≤ *ys* ==> *xs* ≤ *ys* @ *zs*
proof −
assume *xs* ≤ *ys*
then obtain *us* **where** *ys* = *xs* @ *us* **..**
hence *ys* @ *zs* = *xs* @ (*us* @ *zs*) **by** *simp*
thus *?thesis* **..**
qed

lemma *append-prefixD*: *xs* @ *ys* ≤ *zs* ⟹ *xs* ≤ *zs*
by (*auto simp add*: *prefix-def*)

theorem *prefix-Cons*: (*xs* ≤ *y* # *ys*) = (*xs* = [] ∨ (∃ *zs*. *xs* = *y* # *zs* ∧ *zs* ≤ *ys*))
by (*cases xs*) (*auto simp add*: *prefix-def*)

theorem *prefix-append*:
(*xs* ≤ *ys* @ *zs*) = (*xs* ≤ *ys* ∨ (∃ *us*. *xs* = *ys* @ *us* ∧ *us* ≤ *zs*))
apply (*induct zs rule*: *rev-induct*)
apply *force*
apply (*simp del*: *append-assoc add*: *append-assoc* [*symmetric*])
apply *simp*
apply *blast*
done

lemma *append-one-prefix*:
xs ≤ *ys* ==> *length xs* < *length ys* ==> *xs* @ [*ys* ! *length xs*] ≤ *ys*
apply (*unfold prefix-def*)
apply (*auto simp add*: *nth-append*)
apply (*case-tac zs*)
apply *auto*
done

theorem *prefix-length-le*: *xs* ≤ *ys* ==> *length xs* ≤ *length ys*
by (*auto simp add*: *prefix-def*)

lemma *prefix-same-cases*:
(*xs*$_1$::'*a list*) ≤ *ys* ⟹ *xs*$_2$ ≤ *ys* ⟹ *xs*$_1$ ≤ *xs*$_2$ ∨ *xs*$_2$ ≤ *xs*$_1$
apply (*simp add*: *prefix-def*)
apply (*erule exE*)+
apply (*simp add*: *append-eq-append-conv-if split*: *if-splits*)
apply (*rule disjI2*)
apply (*rule-tac x* = *drop* (*size xs*$_2$) *xs*$_1$ **in** *exI*)
apply *clarify*
apply (*drule sym*)
apply (*insert append-take-drop-id* [*of length xs*$_2$ *xs*$_1$])

apply *simp*
apply (*rule disjI1*)
apply (*rule-tac x = drop* (*size* xs_1) xs_2 **in** *exI*)
apply *clarify*
apply (*insert append-take-drop-id* [*of length* xs_1 xs_2])
apply *simp*
done

lemma *set-mono-prefix*:
$xs \le ys \Longrightarrow set\ xs \subseteq set\ ys$
by (*auto simp add*: *prefix-def*)

21.3 Parallel lists

constdefs
parallel :: *'a list => 'a list => bool* (**infixl** $\parallel$ *50*)
$xs \parallel ys == \neg\ xs \le ys \wedge \neg\ ys \le xs$

lemma *parallelI* [*intro*]: $\neg\ xs \le ys ==> \neg\ ys \le xs ==> xs \parallel ys$
by (*unfold parallel-def*) *blast*

lemma *parallelE* [*elim*]:
$xs \parallel ys ==> (\neg\ xs \le ys ==> \neg\ ys \le xs ==> C) ==> C$
by (*unfold parallel-def*) *blast*

theorem *prefix-cases*:
$(xs \le ys ==> C) ==>$
$(ys < xs ==> C) ==>$
$(xs \parallel ys ==> C) ==> C$
by (*unfold parallel-def strict-prefix-def*) *blast*

theorem *parallel-decomp*:
$xs \parallel ys ==> \exists as\ b\ bs\ c\ cs.\ b \neq c \wedge xs = as\ @\ b\ \#\ bs \wedge ys = as\ @\ c\ \#\ cs$
proof (*induct xs rule*: *rev-induct*)
case *Nil*
hence *False* **by** *auto*
thus *?case* **..**
next
case (*snoc x xs*)
show *?case*
proof (*rule prefix-cases*)
assume *le*: $xs \le ys$
then obtain ys' **where** *ys*: $ys = xs\ @\ ys'$ **..**
show *?thesis*
proof (*cases* ys')
assume $ys' = []$ **with** *ys* **have** $xs = ys$ **by** *simp*
with *snoc* **have** $[x] \parallel []$ **by** *auto*
hence *False* **by** *blast*
thus *?thesis* **..**

```
    next
      fix c cs assume ys': ys' = c # cs
      with snoc ys have xs @ [x] ‖ xs @ c # cs by (simp only:)
      hence x ≠ c by auto
      moreover have xs @ [x] = xs @ x # [] by simp
      moreover from ys ys' have ys = xs @ c # cs by (simp only:)
      ultimately show ?thesis by blast
    qed
  next
    assume ys < xs hence ys ≤ xs @ [x] by (simp add: strict-prefix-def)
    with snoc have False by blast
    thus ?thesis ..
  next
    assume xs ‖ ys
    with snoc obtain as b bs c cs where neq: (b::'a) ≠ c
      and xs: xs = as @ b # bs and ys: ys = as @ c # cs
      by blast
    from xs have xs @ [x] = as @ b # (bs @ [x]) by simp
    with neq ys show ?thesis by blast
  qed
qed
```

21.4 Postfix order on lists

```
constdefs
  postfix :: 'a list => 'a list => bool  ((-/ >>= -) [51, 50] 50)
  xs >>= ys == ∃zs. xs = zs @ ys

lemma postfix-refl [simp, intro!]: xs >>= xs
  by (auto simp add: postfix-def)
lemma postfix-trans: ⟦xs >>= ys; ys >>= zs⟧ ⟹ xs >>= zs
  by (auto simp add: postfix-def)
lemma postfix-antisym: ⟦xs >>= ys; ys >>= xs⟧ ⟹ xs = ys
  by (auto simp add: postfix-def)

lemma Nil-postfix [iff]: xs >>= []
  by (simp add: postfix-def)
lemma postfix-Nil [simp]: ([] >>= xs) = (xs = [])
  by (auto simp add:postfix-def)

lemma postfix-ConsI: xs >>= ys ⟹ x#xs >>= ys
  by (auto simp add: postfix-def)
lemma postfix-ConsD: xs >>= y#ys ⟹ xs >>= ys
  by (auto simp add: postfix-def)

lemma postfix-appendI: xs >>= ys ⟹ zs @ xs >>= ys
  by (auto simp add: postfix-def)
lemma postfix-appendD: xs >>= zs @ ys ⟹ xs >>= ys
  by(auto simp add: postfix-def)
```

lemma *postfix-is-subset-lemma*: $xs = zs \ @ \ ys \Longrightarrow set\ ys \subseteq set\ xs$
by (*induct zs*, *auto*)
lemma *postfix-is-subset*: $xs >>= ys \Longrightarrow set\ ys \subseteq set\ xs$
by (*unfold postfix-def*, *erule exE*, *erule postfix-is-subset-lemma*)

lemma *postfix-ConsD2-lemma* [*rule-format*]: $x\#xs = zs \ @ \ y\#ys \longrightarrow xs >>= ys$
by (*induct zs*, *auto intro*!: *postfix-appendI postfix-ConsI*)
lemma *postfix-ConsD2*: $x\#xs >>= y\#ys \Longrightarrow xs >>= ys$
by (*auto simp add*: *postfix-def dest*!: *postfix-ConsD2-lemma*)

lemma *postfix2prefix*: $(xs >>= ys) = (rev\ ys <= rev\ xs)$
apply (*unfold prefix-def postfix-def*, *safe*)
apply (*rule-tac* $x = rev\ zs$ **in** *exI*, *simp*)
apply (*rule-tac* $x = rev\ zs$ **in** *exI*)
apply (*rule rev-is-rev-conv* [*THEN iffD1*], *simp*)
done

end

22 List-lexord: Lexicographic order on lists

theory *List-lexord*
imports *Main*
begin

instance *list* :: (*ord*) *ord* **..**
defs (**overloaded**)
list-le-def: $(xs::('a::ord)\ list) \leq ys \equiv (xs < ys \vee xs = ys)$
list-less-def: $(xs::('a::ord)\ list) < ys \equiv (xs, ys) \in lexord\ \{(u,v).\ u < v\}$

lemmas *list-ord-defs* = *list-less-def list-le-def*

instance *list* :: (*order*) *order*
apply (*intro-classes*, *unfold list-ord-defs*)
apply (*rule disjI2*, *safe*)
apply (*blast intro*: *lexord-trans transI order-less-trans*)
apply (*rule-tac r1* = $\{(a::'a,b).\ a < b\}$ **in** *lexord-irreflexive* [*THEN notE*])
apply *simp*
apply (*blast intro*: *lexord-trans transI order-less-trans*)
apply (*rule-tac r1* = $\{(a::'a,b).\ a < b\}$ **in** *lexord-irreflexive* [*THEN notE*])
apply *simp*
apply *assumption*
done

instance *list*::(*linorder*)*linorder*
apply (*intro-classes*, *unfold list-le-def list-less-def*, *safe*)
apply (*cut-tac* $x = x$ **and** $y = y$ **and** $r = \{(a,b).\ a < b\}$ **in** *lexord-linear*)

```
 apply force
apply simp
done

lemma not-less-Nil[simp]: ~(x < [])
  by (unfold list-less-def) simp

lemma Nil-less-Cons[simp]: [] < a # x
  by (unfold list-less-def) simp

lemma Cons-less-Cons[simp]: (a # x < b # y) = (a < b | a = b & x < y)
  by (unfold list-less-def) simp

lemma le-Nil[simp]: (x <= []) = (x = [])
  by (unfold list-ord-defs, cases x) auto

lemma Nil-le-Cons [simp]: ([] <= x)
  by (unfold list-ord-defs, cases x) auto

lemma Cons-le-Cons[simp]: (a # x <= b # y) = (a < b | a = b & x <= y)
  by (unfold list-ord-defs) auto

end
```

References

[1] Abrial and Laffitte. Towards the mechanization of the proofs of some classical theorems of set theory. Unpublished.

[2] J. Avigad and K. Donnelly. Formalizing O notation in Isabelle/HOL. In D. Basin and M. Rusiowitch, editors, *Automated Reasoning: second international conference, IJCAR 2004*, pages 357–371. Springer, 2004.

[3] A. Oberschelp. *Rekursionstheorie*. BI-Wissenschafts-Verlag, 1993.

[4] C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, LNCS 664, pages 328–345. Springer, 1993.