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

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

1.2 Induction rules

theorem *acc-induct*:
  $a \in acc\ r ==>$
    $(!!x.\ x \in acc\ r ==> \forall\, y.\ (y,\ x) \in r --> P\ y ==> P\ x) ==> P\ a$
⟨*proof*⟩

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

theorem *acc-downward*: $b \in acc\ r ==> (a,\ b) \in r ==> a \in acc\ r$
  ⟨*proof*⟩

lemma *acc-downwards-aux*: $(b,\ a) \in r^* ==> a \in acc\ r --> b \in acc\ r$
  ⟨*proof*⟩

theorem *acc-downwards*: $a \in acc\ r ==> (b,\ a) \in r^* ==> b \in acc\ r$
  ⟨*proof*⟩

theorem *acc-wfI*: $\forall\, x.\ x \in acc\ r ==> wf\ r$
  ⟨*proof*⟩

theorem *acc-wfD*: $wf\ r ==> x \in acc\ r$
  ⟨*proof*⟩

theorem *wf-acc-iff*: $wf\ r = (\forall\, x.\ x \in acc\ r)$
  ⟨*proof*⟩

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* ⟨*proof*⟩
instance *fun* :: (*type*, *plus*) *plus* ⟨*proof*⟩

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* ⟨*proof*⟩
instance *fun* :: (*type*, *times*) *times* ⟨*proof*⟩

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* ⟨*proof*⟩

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

instance *fun* :: (*type*, *zero*) *zero* ⟨*proof*⟩
instance *set* :: (*zero*) *zero* ⟨*proof*⟩

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

instance *fun* :: (*type*, *one*) *one* ⟨*proof*⟩
instance *set* :: (*one*) *one* ⟨*proof*⟩

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*
⟨*proof*⟩

instance *fun* :: (*type*,*comm-monoid-add*)*comm-monoid-add*
⟨*proof*⟩

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

instance *fun* :: (*type*,*semigroup-mult*)*semigroup-mult*
⟨*proof*⟩

instance *fun* :: (*type*,*comm-monoid-mult*)*comm-monoid-mult*
⟨*proof*⟩

instance *fun* :: (*type*,*comm-ring-1*)*comm-ring-1*
⟨*proof*⟩

instance *set* :: (*semigroup-add*)*semigroup-add*
⟨*proof*⟩

instance *set* :: (*semigroup-mult*)*semigroup-mult*
⟨*proof*⟩

instance *set* :: (*comm-monoid-add*)*comm-monoid-add*
⟨*proof*⟩

instance *set* :: (*comm-monoid-mult*)*comm-monoid-mult*
⟨*proof*⟩

2.2 Basic properties

lemma *set-plus-intro* [*intro*]: $a : C ==> b : D ==> a + b : C + D$
⟨*proof*⟩

lemma *set-plus-intro2* [*intro*]: $b : C ==> a + b : a +o C$
⟨*proof*⟩

lemma *set-plus-rearrange*: $((a::'a::comm\text{-}monoid\text{-}add) +o C) +$
$(b +o D) = (a + b) +o (C + D)$
⟨*proof*⟩

lemma *set-plus-rearrange2*: $(a::'a::semigroup\text{-}add) +o (b +o C) =$
$(a + b) +o C$
⟨*proof*⟩

lemma *set-plus-rearrange3*: $((a::'a::semigroup\text{-}add) +o B) + C =$
$a +o (B + C)$
⟨*proof*⟩

theorem *set-plus-rearrange4*: $C + ((a::'a::comm\text{-}monoid\text{-}add) +o D) =$
$a +o (C + D)$
⟨*proof*⟩

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

lemma *set-plus-mono* [*intro*!]: $C <= D ==> a +o C <= a +o D$
⟨*proof*⟩

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

lemma *set-plus-mono3* [*intro*]: $a : C ==> a +o D <= C + D$
⟨*proof*⟩

lemma *set-plus-mono4* [*intro*]: $(a::'a::comm\text{-}monoid\text{-}add) : C ==>$
$a +o D <= D + C$
⟨*proof*⟩

lemma *set-plus-mono5*: $a{:}C ==> B <= D ==> a +o B <= C + D$
⟨*proof*⟩

lemma *set-plus-mono-b*: $C <= D ==> x : a +o C$
$==> x : a +o D$
⟨*proof*⟩

lemma *set-plus-mono2-b*: $C <= D ==> E <= F ==> x : C + E ==>$
$x : D + F$
⟨*proof*⟩

lemma *set-plus-mono3-b*: $a : C ==> x : a +o D ==> x : C + D$
⟨*proof*⟩

lemma *set-plus-mono4-b*: (*a*::'*a*::*comm-monoid-add*) : *C* ==>
x : *a* +*o* *D* ==> *x* : *D* + *C*
⟨*proof*⟩

lemma *set-zero-plus* [*simp*]: (*0*::'*a*::*comm-monoid-add*) +*o* *C* = *C*
⟨*proof*⟩

lemma *set-zero-plus2*: (*0*::'*a*::*comm-monoid-add*) : *A* ==> *B* <= *A* + *B*
⟨*proof*⟩

lemma *set-plus-imp-minus*: (*a*::'*a*::*ab-group-add*) : *b* +*o* *C* ==> (*a* − *b*) : *C*
⟨*proof*⟩

lemma *set-minus-imp-plus*: (*a*::'*a*::*ab-group-add*) − *b* : *C* ==> *a* : *b* +*o* *C*
⟨*proof*⟩

lemma *set-minus-plus*: ((*a*::'*a*::*ab-group-add*) − *b* : *C*) = (*a* : *b* +*o* *C*)
⟨*proof*⟩

lemma *set-times-intro* [*intro*]: *a* : *C* ==> *b* : *D* ==> *a* ∗ *b* : *C* ∗ *D*
⟨*proof*⟩

lemma *set-times-intro2* [*intro*!]: *b* : *C* ==> *a* ∗ *b* : *a* ∗*o* *C*
⟨*proof*⟩

lemma *set-times-rearrange*: ((*a*::'*a*::*comm-monoid-mult*) ∗*o* *C*) ∗
(*b* ∗*o* *D*) = (*a* ∗ *b*) ∗*o* (*C* ∗ *D*)
⟨*proof*⟩

lemma *set-times-rearrange2*: (*a*::'*a*::*semigroup-mult*) ∗*o* (*b* ∗*o* *C*) =
(*a* ∗ *b*) ∗*o* *C*
⟨*proof*⟩

lemma *set-times-rearrange3*: ((*a*::'*a*::*semigroup-mult*) ∗*o* *B*) ∗ *C* =
a ∗*o* (*B* ∗ *C*)
⟨*proof*⟩

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

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*
⟨*proof*⟩

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

$C * E <= D * F$
⟨proof⟩

lemma *set-times-mono3* [*intro*]: $a : C ==> a *o D <= C * D$
⟨proof⟩

lemma *set-times-mono4* [*intro*]: $(a::'a::comm\text{-}monoid\text{-}mult) : C ==>$
$a *o D <= D * C$
⟨proof⟩

lemma *set-times-mono5*: $a{:}C ==> B <= D ==> a *o B <= C * D$
⟨proof⟩

lemma *set-times-mono-b*: $C <= D ==> x : a *o C$
$==> x : a *o D$
⟨proof⟩

lemma *set-times-mono2-b*: $C <= D ==> E <= F ==> x : C * E ==>$
$x : D * F$
⟨proof⟩

lemma *set-times-mono3-b*: $a : C ==> x : a *o D ==> x : C * D$
⟨proof⟩

lemma *set-times-mono4-b*: $(a::'a::comm\text{-}monoid\text{-}mult) : C ==>$
$x : a *o D ==> x : D * C$
⟨proof⟩

lemma *set-one-times* [*simp*]: $(1::'a::comm\text{-}monoid\text{-}mult) *o C = C$
⟨proof⟩

lemma *set-times-plus-distrib*: $(a::'a::semiring) *o (b +o C)=$
$(a * b) +o (a *o C)$
⟨proof⟩

lemma *set-times-plus-distrib2*: $(a::'a::semiring) *o (B + C) =$
$(a *o B) + (a *o C)$
⟨proof⟩

lemma *set-times-plus-distrib3*: $((a::'a::semiring) +o C) * D <=$
$a *o D + C * D$
⟨proof⟩

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

lemma *set-neg-intro*: $(a::'a::ring\text{-}1) : (- 1) *o C ==>$
$- a : C$
⟨proof⟩

lemma *set-neg-intro2*: $(a::'a::ring\text{-}1) : C ==>$
  $- a : (- 1) *o\ C$
$\langle proof \rangle$

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

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

lemma *bigo-pos-const*: $(EX\ (c::'a::ordered\text{-}idom).$
  $ALL\ x.\ (abs\ (h\ x)) <= (c * (abs\ (f\ x))))$

$= (EX\ c.\ 0 < c\ \&\ (ALL\ x.\ (abs(h\ x)) <= (c * (abs\ (f\ x)))))$
⟨*proof*⟩

lemma *bigo-alt-def*: $O(f) =$
$\{h.\ EX\ c.\ (0 < c\ \&\ (ALL\ x.\ abs\ (h\ x) <= c * abs\ (f\ x)))\}$
⟨*proof*⟩

lemma *bigo-elt-subset* [*intro*]: $f : O(g) ==> O(f) <= O(g)$
⟨*proof*⟩

lemma *bigo-refl* [*intro*]: $f : O(f)$
⟨*proof*⟩

lemma *bigo-zero*: $0 : O(g)$
⟨*proof*⟩

lemma *bigo-zero2*: $O(\%x.0) = \{\%x.0\}$
⟨*proof*⟩

lemma *bigo-plus-self-subset* [*intro*]:
$O(f) + O(f) <= O(f)$
⟨*proof*⟩

lemma *bigo-plus-idemp* [*simp*]: $O(f) + O(f) = O(f)$
⟨*proof*⟩

lemma *bigo-plus-subset* [*intro*]: $O(f + g) <= O(f) + O(g)$
⟨*proof*⟩

lemma *bigo-plus-subset2* [*intro*]: $A <= O(f) ==> B <= O(f) ==> A + B <= O(f)$
⟨*proof*⟩

lemma *bigo-plus-eq*: $ALL\ x.\ 0 <= f\ x ==> ALL\ x.\ 0 <= g\ x ==>$
$O(f + g) = O(f) + O(g)$
⟨*proof*⟩

lemma *bigo-bounded-alt*: $ALL\ x.\ 0 <= f\ x ==> ALL\ x.\ f\ x <= c * g\ x ==>$
$f : O(g)$
⟨*proof*⟩

lemma *bigo-bounded*: $ALL\ x.\ 0 <= f\ x ==> ALL\ x.\ f\ x <= g\ x ==>$
$f : O(g)$
⟨*proof*⟩

lemma *bigo-bounded2*: $ALL\ x.\ lb\ x <= f\ x ==> ALL\ x.\ f\ x <= lb\ x + g\ x ==>$
$f : lb +o\ O(g)$
⟨*proof*⟩

lemma *bigo-abs*: $(\%x.\ abs(f\ x)) =o\ O(f)$
⟨*proof*⟩

lemma *bigo-abs2*: $f =o\ O(\%x.\ abs(f\ x))$
⟨*proof*⟩

lemma *bigo-abs3*: $O(f) = O(\%x.\ abs(f\ x))$
⟨*proof*⟩

lemma *bigo-abs4*: $f =o\ g\ +o\ O(h) ==>$
$(\%x.\ abs\ (f\ x)) =o\ (\%x.\ abs\ (g\ x))\ +o\ O(h)$
⟨*proof*⟩

lemma *bigo-abs5*: $f =o\ O(g) ==> (\%x.\ abs(f\ x)) =o\ O(g)$
⟨*proof*⟩

lemma *bigo-elt-subset2* [*intro*]: $f : g\ +o\ O(h) ==> O(f) <= O(g) + O(h)$
⟨*proof*⟩

lemma *bigo-mult* [*intro*]: $O(f)*O(g) <= O(f * g)$
⟨*proof*⟩

lemma *bigo-mult2* [*intro*]: $f *o\ O(g) <= O(f * g)$
⟨*proof*⟩

lemma *bigo-mult3*: $f : O(h) ==> g : O(j) ==> f * g : O(h * j)$
⟨*proof*⟩

lemma *bigo-mult4* [*intro*]:$f : k\ +o\ O(h) ==> g * f : (g * k)\ +o\ O(g * h)$
⟨*proof*⟩

lemma *bigo-mult5*: $ALL\ x.\ f\ x\ \tilde{}= 0 ==>$
$O(f * g) <= (f::'a => ('b::ordered\text{-}field)) *o\ O(g)$
⟨*proof*⟩

lemma *bigo-mult6*: $ALL\ x.\ f\ x\ \tilde{}= 0 ==>$
$O(f * g) = (f::'a => ('b::ordered\text{-}field)) *o\ O(g)$
⟨*proof*⟩

lemma *bigo-mult7*: $ALL\ x.\ f\ x\ \tilde{}= 0 ==>$
$O(f * g) <= O(f::'a => ('b::ordered\text{-}field)) * O(g)$
⟨*proof*⟩

lemma *bigo-mult8*: $ALL\ x.\ f\ x\ \tilde{}= 0 ==>$
$O(f * g) = O(f::'a => ('b::ordered\text{-}field)) * O(g)$
⟨*proof*⟩

lemma *bigo-minus* [*intro*]: $f : O(g) ==> - f : O(g)$
⟨*proof*⟩

lemma *bigo-minus2*: $f : g +o O(h) ==> -f : -g +o O(h)$
⟨*proof*⟩

lemma *bigo-minus3*: $O(-f) = O(f)$
⟨*proof*⟩

lemma *bigo-plus-absorb-lemma1*: $f : O(g) ==> f +o O(g) <= O(g)$
⟨*proof*⟩

lemma *bigo-plus-absorb-lemma2*: $f : O(g) ==> O(g) <= f +o O(g)$
⟨*proof*⟩

lemma *bigo-plus-absorb* [*simp*]: $f : O(g) ==> f +o O(g) = O(g)$
⟨*proof*⟩

lemma *bigo-plus-absorb2* [*intro*]: $f : O(g) ==> A <= O(g) ==> f +o A <= O(g)$
⟨*proof*⟩

lemma *bigo-add-commute-imp*: $f : g +o O(h) ==> g : f +o O(h)$
⟨*proof*⟩

lemma *bigo-add-commute*: $(f : g +o O(h)) = (g : f +o O(h))$
⟨*proof*⟩

lemma *bigo-const1*: $(\%x.\ c) : O(\%x.\ 1)$
⟨*proof*⟩

lemma *bigo-const2* [*intro*]: $O(\%x.\ c) <= O(\%x.\ 1)$
⟨*proof*⟩

lemma *bigo-const3*: $(c{::}'a{::}ordered\text{-}field)\ \tilde{}= 0 ==> (\%x.\ 1) : O(\%x.\ c)$
⟨*proof*⟩

lemma *bigo-const4*: $(c{::}'a{::}ordered\text{-}field)\ \tilde{}= 0 ==> O(\%x.\ 1) <= O(\%x.\ c)$
⟨*proof*⟩

lemma *bigo-const* [*simp*]: $(c{::}'a{::}ordered\text{-}field)\ \tilde{}= 0 ==>$
$O(\%x.\ c) = O(\%x.\ 1)$
⟨*proof*⟩

lemma *bigo-const-mult1*: $(\%x.\ c * f\ x) : O(f)$
⟨*proof*⟩

lemma *bigo-const-mult2*: $O(\%x.\ c * f\ x) <= O(f)$
⟨*proof*⟩

lemma *bigo-const-mult3*: $(c{::}'a{::}ordered\text{-}field)\ \tilde{}= 0 ==> f : O(\%x.\ c * f\ x)$

⟨proof⟩

lemma *bigo-const-mult4*: $(c::'a::ordered\text{-}field)\ \tilde{}{=}\ 0 ==>$
$O(f) <= O(\%x.\ c * f\ x)$
⟨proof⟩

lemma *bigo-const-mult* [*simp*]: $(c::'a::ordered\text{-}field)\ \tilde{}{=}\ 0 ==>$
$O(\%x.\ c * f\ x) = O(f)$
⟨proof⟩

lemma *bigo-const-mult5* [*simp*]: $(c::'a::ordered\text{-}field)\ \tilde{}{=}\ 0 ==>$
$(\%x.\ c) *o\ O(f) = O(f)$
⟨proof⟩

lemma *bigo-const-mult6* [*intro*]: $(\%x.\ c) *o\ O(f) <= O(f)$
⟨proof⟩

lemma *bigo-const-mult7* [*intro*]: $f =o\ O(g) ==> (\%x.\ c * f\ x) =o\ O(g)$
⟨proof⟩

lemma *bigo-compose1*: $f =o\ O(g) ==> (\%x.\ f(k\ x)) =o\ O(\%x.\ g(k\ x))$
⟨proof⟩

lemma *bigo-compose2*: $f =o\ g +o\ O(h) ==> (\%x.\ f(k\ x)) =o\ (\%x.\ g(k\ x)) +o$
$O(\%x.\ h(k\ x))$
⟨proof⟩

3.2 Setsum

lemma *bigo-setsum-main*: $ALL\ x.\ ALL\ y : A\ x.\ 0 <= h\ x\ y ==>$
$EX\ c.\ ALL\ x.\ ALL\ y : A\ x.\ abs(f\ x\ y) <= c * (h\ x\ y) ==>$
$(\%x.\ SUM\ y : A\ x.\ f\ x\ y) =o\ O(\%x.\ SUM\ y : A\ x.\ h\ x\ y)$
⟨proof⟩

lemma *bigo-setsum1*: $ALL\ x\ y.\ 0 <= h\ x\ y ==>$
$EX\ c.\ ALL\ x\ y.\ abs(f\ x\ y) <= c * (h\ x\ y) ==>$
$(\%x.\ SUM\ y : A\ x.\ f\ x\ y) =o\ O(\%x.\ SUM\ y : A\ x.\ h\ x\ y)$
⟨proof⟩

lemma *bigo-setsum2*: $ALL\ y.\ 0 <= h\ y ==>$
$EX\ c.\ ALL\ y.\ abs(f\ y) <= c * (h\ y) ==>$
$(\%x.\ SUM\ y : A\ x.\ f\ y) =o\ O(\%x.\ SUM\ y : A\ x.\ h\ y)$
⟨proof⟩

lemma *bigo-setsum3*: $f =o\ O(h) ==>$
$(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * f(k\ x\ y)) =o$
$O(\%x.\ SUM\ y : A\ x.\ abs(l\ x\ y * h(k\ x\ y)))$
⟨proof⟩

lemma *bigo-setsum4*: $f =o\ g +o\ O(h) ==>$
  $(\%x.\ SUM\ y : A\ x.\ l\ x\ y * f(k\ x\ y)) =o$
    $(\%x.\ SUM\ y : A\ x.\ l\ x\ y * g(k\ x\ y)) +o$
      $O(\%x.\ SUM\ y : A\ x.\ abs(l\ x\ y * h(k\ x\ y)))$
  ⟨*proof*⟩

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

lemma *bigo-setsum6*: $f =o\ g +o\ O(h) ==> ALL\ x\ y.\ 0 <= l\ x\ y ==>$
  $ALL\ x.\ 0 <= h\ x ==>$
    $(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * f(k\ x\ y)) =o$
      $(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * g(k\ x\ y)) +o$
        $O(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * h(k\ x\ y))$
  ⟨*proof*⟩

3.3 Misc useful stuff

lemma *bigo-useful-intro*: $A <= O(f) ==> B <= O(f) ==>$
  $A + B <= O(f)$
  ⟨*proof*⟩

lemma *bigo-useful-add*: $f =o\ O(h) ==> g =o\ O(h) ==> f + g =o\ O(h)$
  ⟨*proof*⟩

lemma *bigo-useful-const-mult*: $(c::'a::ordered\text{-}field)\ \tilde{}= 0 ==>$
  $(\%x.\ c) * f =o\ O(h) ==> f =o\ O(h)$
  ⟨*proof*⟩

lemma *bigo-fix*: $(\%x.\ f\ ((x::nat) + 1)) =o\ O(\%x.\ h(x + 1)) ==> f\ 0 = 0 ==>$
  $f =o\ O(h)$
  ⟨*proof*⟩

lemma *bigo-fix2*:
  $(\%x.\ f\ ((x::nat) + 1)) =o\ (\%x.\ g(x + 1)) +o\ O(\%x.\ h(x + 1)) ==>$
    $f\ 0 = g\ 0 ==> f =o\ g +o\ O(h)$
  ⟨*proof*⟩

3.4 Less than or equal to

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

lemma *bigo-lesseq1*: $f =o\ O(h) ==> ALL\ x.\ abs\ (g\ x) <= abs\ (f\ x) ==>$
  $g =o\ O(h)$

⟨*proof*⟩

lemma *bigo-lesseq2*: *f =o O(h) ==> ALL x. abs (g x) <= f x ==>*
g =o O(h)
⟨*proof*⟩

lemma *bigo-lesseq3*: *f =o O(h) ==> ALL x. 0 <= g x ==> ALL x. g x <= f x ==>*
g =o O(h)
⟨*proof*⟩

lemma *bigo-lesseq4*: *f =o O(h) ==>*
ALL x. 0 <= g x ==> ALL x. g x <= abs (f x) ==>
g =o O(h)
⟨*proof*⟩

lemma *bigo-lesso1*: *ALL x. f x <= g x ==> f <o g =o O(h)*
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *bigo-lesso5*: *f <o g =o O(h) ==>*
*EX C. ALL x. f x <= g x + C * abs(h x)*
⟨*proof*⟩

lemma *lesso-add*: *f <o g =o O(h) ==>*
k <o l =o O(h) ==> (f + k) <o (g + l) =o O(h)
⟨*proof*⟩

end

4 Continuity: Continuity and iterations (of set transformers)

theory *Continuity*

imports *Main*
begin

4.1 Chains

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

lemma *up-chainI*: $(!!i.\ F\ i \subseteq F\ (Suc\ i)) ==> up\text{-}chain\ F$
  ⟨*proof*⟩

lemma *up-chainD*: $up\text{-}chain\ F ==> F\ i \subseteq F\ (Suc\ i)$
  ⟨*proof*⟩

lemma *up-chain-less-mono* [*rule-format*]:
    $up\text{-}chain\ F ==> x < y --> F\ x \subseteq F\ y$
  ⟨*proof*⟩

lemma *up-chain-mono*: $up\text{-}chain\ F ==> x \le y ==> F\ x \subseteq F\ y$
  ⟨*proof*⟩

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

lemma *down-chainI*: $(!!i.\ F\ (Suc\ i) \subseteq F\ i) ==> down\text{-}chain\ F$
  ⟨*proof*⟩

lemma *down-chainD*: $down\text{-}chain\ F ==> F\ (Suc\ i) \subseteq F\ i$
  ⟨*proof*⟩

lemma *down-chain-less-mono* [*rule-format*]:
    $down\text{-}chain\ F ==> x < y --> F\ y \subseteq F\ x$
  ⟨*proof*⟩

lemma *down-chain-mono*: $down\text{-}chain\ F ==> x \le y ==> F\ y \subseteq F\ x$
  ⟨*proof*⟩

4.2 Continuity

constdefs
  $up\text{-}cont :: ('a\ set => 'a\ set) => bool$
  $up\text{-}cont\ f == \forall F.\ up\text{-}chain\ F --> f\ (\bigcup (range\ F)) = \bigcup (f\ `\ range\ F)$

lemma *up-contI*:
    $(!!F.\ up\text{-}chain\ F ==> f\ (\bigcup (range\ F)) = \bigcup (f\ `\ range\ F)) ==> up\text{-}cont\ f$
  ⟨*proof*⟩

lemma *up-contD*:
  up-cont f ==> *up-chain F* ==> *f* ($\bigcup$(*range F*)) = $\bigcup$(*f* ' *range F*)
⟨*proof*⟩

lemma *up-cont-mono*: *up-cont f* ==> *mono f*
⟨*proof*⟩

constdefs
down-cont :: ('*a set* => '*a set*) => *bool*
down-cont f ==
  ∀ *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
⟨*proof*⟩

lemma *down-contD*: *down-cont f* ==> *down-chain F* ==>
  f (*Inter* (*range F*)) = *Inter* (*f* ' *range F*)
⟨*proof*⟩

lemma *down-cont-mono*: *down-cont f* ==> *mono f*
⟨*proof*⟩

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* = {}
⟨*proof*⟩

lemma *up-iterate-Suc* [*simp*]: *up-iterate f* (*Suc i*) = *f* (*up-iterate f i*)
⟨*proof*⟩

lemma *up-iterate-chain*: *mono F* ==> *up-chain* (*up-iterate F*)
⟨*proof*⟩

lemma *UNION-up-iterate-is-fp*:
up-cont F ==>
  F (*UNION UNIV* (*up-iterate F*)) = *UNION UNIV* (*up-iterate F*)
⟨*proof*⟩

lemma *UNION-up-iterate-lowerbound*:
  mono F ==> *F P* = *P* ==> *UNION UNIV* (*up-iterate F*) ⊆ *P*
⟨*proof*⟩

lemma *UNION-up-iterate-is-lfp*:
  up-cont F ==> lfp F = UNION UNIV (up-iterate F)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *down-iterate-Suc* [*simp*]:
  down-iterate f (*Suc i*) = *f* (*down-iterate f i*)
⟨*proof*⟩

lemma *down-iterate-chain*: *mono F ==> down-chain* (*down-iterate F*)
⟨*proof*⟩

lemma *INTER-down-iterate-is-fp*:
down-cont F ==>
  F (*INTER UNIV* (*down-iterate F*)) = *INTER UNIV* (*down-iterate F*)
⟨*proof*⟩

lemma *INTER-down-iterate-upperbound*:
  mono F ==> F P = P ==> P ⊆ *INTER UNIV* (*down-iterate F*)
⟨*proof*⟩

lemma *INTER-down-iterate-is-gfp*:
  down-cont F ==> gfp F = INTER UNIV (*down-iterate F*)
⟨*proof*⟩

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 $nat \Rightarrow int$ 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*]: $nat\text{-}case \equiv (\lambda f\ g\ n.\ if\ n = 0\ then\ f\ else\ g\ (n - 1))$
  ⟨*proof*⟩

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

lemma [*code*]: $m - n = nat\ (int\ m - int\ n)$ ⟨*proof*⟩
lemma [*code*]: $m + n = nat\ (int\ m + int\ n)$ ⟨*proof*⟩
lemma [*code*]: $m * n = nat\ (int\ m * int\ n)$
  ⟨*proof*⟩
lemma [*code*]: $m\ div\ n = nat\ (int\ m\ div\ int\ n)$
  ⟨*proof*⟩
lemma [*code*]: $m\ mod\ n = nat\ (int\ m\ mod\ int\ n)$
  ⟨*proof*⟩
lemma [*code*]: $(m < n) = (int\ m < int\ n)$
  ⟨*proof*⟩

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*: $(\bigwedge n.\ f\ (Suc\ n) = h\ n) \Longrightarrow f\ 0 = g \Longrightarrow$
$f\ n = (if\ n = 0\ then\ g\ else\ h\ (n - 1))$
⟨*proof*⟩

theorem *Suc-clause*: $(\bigwedge n.\ P\ n\ (Suc\ n)) \Longrightarrow n \neq 0 \Longrightarrow P\ (n - 1)\ n$
⟨*proof*⟩

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.

⟨*ML*⟩
end

6 ExecutableSet: Implementation of finite sets by lists

theory *ExecutableSet*
imports *Main*
begin

lemma [*code target*: *Set*]: $(A = B) = (A \subseteq B \wedge B \subseteq A)$
⟨*proof*⟩

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*)

  $@lam :: [pttrn, 'a\ set, 'a => 'b] => ('a=>'b)\ \ ((3\%\text{-}:\text{-}./\ \text{-})\ [0,0,3]\ 3)$

syntax (*xsymbols*)
  $@Pi :: [pttrn, 'a\ set, 'b\ set] => ('a => 'b)\ set\ \ ((3\Pi\ \text{-}\in\text{-}./\ \text{-})\quad 10)$
  $funcset :: ['a\ set, 'b\ set] => ('a => 'b)\ set$ (**infixr** $\rightarrow$ *60*)
  $@lam :: [pttrn, 'a\ set, 'a => 'b] => ('a=>'b)\ \ ((3\lambda\text{-}\in\text{-}./\ \text{-})\ [0,0,3]\ 3)$

syntax (*HTML* **output**)
  $@Pi :: [pttrn, 'a\ set, 'b\ set] => ('a => 'b)\ set\ \ ((3\Pi\ \text{-}\in\text{-}./\ \text{-})\quad 10)$
  $@lam :: [pttrn, 'a\ set, 'a => 'b] => ('a=>'b)\ \ ((3\lambda\text{-}\in\text{-}./\ \text{-})\ [0,0,3]\ 3)$

translations
  $PI\ x{:}A.\ B => Pi\ A\ (\%x.\ B)$
  $A -> B => Pi\ A\ (\text{-}K\ B)$
  $\%x{:}A.\ f == restrict\ (\%x.\ f)\ A$

constdefs
  $compose :: ['a\ set, 'b => 'c, 'a => 'b] => ('a => 'c)$
  $compose\ A\ g\ f == \lambda x{\in}A.\ g\ (f\ x)$

⟨*ML*⟩

7.1 Basic Properties of *Pi*

lemma *Pi-I*: $(!!x.\ x \in A ==> f\ x \in B\ x) ==> f \in Pi\ A\ B$
  ⟨*proof*⟩

lemma *funcsetI*: $(!!x.\ x \in A ==> f\ x \in B) ==> f \in A -> B$
  ⟨*proof*⟩

lemma *Pi-mem*: $[|f{:}\ Pi\ A\ B;\ x \in A|] ==> f\ x \in B\ x$
  ⟨*proof*⟩

lemma *funcset-mem*: $[|f \in A -> B;\ x \in A|] ==> f\ x \in B$
  ⟨*proof*⟩

lemma *funcset-image*: $f \in A{\rightarrow}B ==> f\ `\ A \subseteq B$
⟨*proof*⟩

lemma *Pi-eq-empty*: $((PI\ x{:}\ A.\ B\ x) = \{\}) = (\exists\, x{\in}A.\ B(x) = \{\})$
⟨*proof*⟩

lemma *Pi-empty* [*simp*]: $Pi\ \{\}\ B = UNIV$
  ⟨*proof*⟩

lemma *Pi-UNIV* [*simp*]: $A -> UNIV = UNIV$
  ⟨*proof*⟩

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: $(!!x.\ x \in A ==> B\ x <= C\ x) ==> Pi\ A\ B <= Pi\ A\ C$
⟨*proof*⟩

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: $A' <= A ==> Pi\ A\ B <= Pi\ A'\ B$
⟨*proof*⟩

7.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*:
$[|\ f \in A -> B;\ g \in B -> C\ |]==> compose\ A\ g\ f \in A -> C$
⟨*proof*⟩

lemma *compose-assoc*:
$[|\ f \in A -> B;\ g \in B -> C;\ h \in C -> D\ |]$
$==> compose\ A\ h\ (compose\ A\ g\ f) = compose\ A\ (compose\ B\ h\ g)\ f$
⟨*proof*⟩

lemma *compose-eq*: $x \in A ==> compose\ A\ g\ f\ x = g(f(x))$
⟨*proof*⟩

lemma *surj-compose*: $[|\ f\ `\ A = B;\ g\ `\ B = C\ |] ==> compose\ A\ g\ f\ `\ A = C$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *restrictI*: $(!!x.\ x \in A ==> f\ x \in B\ x) ==> (\lambda x{\in}A.\ f\ x) \in Pi\ A\ B$
⟨*proof*⟩

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

lemma *restrict-ext*:
$(!!x.\ x \in A ==> f\ x = g\ x) ==> (\lambda x{\in}A.\ f\ x) = (\lambda x{\in}A.\ g\ x)$
⟨*proof*⟩

lemma *inj-on-restrict-eq* [*simp*]: $inj\text{-}on\ (restrict\ f\ A)\ A = inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *Id-compose*:
$[|f \in A -> B;\ f \in extensional\ A|] ==> compose\ A\ (\lambda y{\in}B.\ y)\ f = f$
⟨*proof*⟩

lemma *compose-Id*:

$[|g \in A \; {-}{>} \; B; \;\; g \in extensional\ A|] \; {=}{=}{>} \; compose\ A\ g\ (\lambda x{\in}A.\ x) = g$
⟨*proof*⟩

lemma *image-restrict-eq* [*simp*]: $(restrict\ f\ A)\ `\ A = f\ `\ A$
⟨*proof*⟩

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 $f\ A\ B \; {=}{=} \; inj\text{-}on\ f\ A\ \&\ f\ `\ A = B$

lemma *bij-betw-imp-inj-on*: $bij\text{-}betw\ f\ A\ B \Longrightarrow inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *bij-betw-imp-funcset*: $bij\text{-}betw\ f\ A\ B \Longrightarrow f \in A \to B$
⟨*proof*⟩

lemma *bij-betw-Inv*: $bij\text{-}betw\ f\ A\ B \Longrightarrow bij\text{-}betw\ (Inv\ A\ f)\ B\ A$
⟨*proof*⟩

lemma *inj-on-compose*:
$[|\ bij\text{-}betw\ f\ A\ B;\ inj\text{-}on\ g\ B\ |] \; {=}{=}{>} \; inj\text{-}on\ (compose\ A\ g\ f)\ A$
⟨*proof*⟩

lemma *bij-betw-compose*:
$[|\ bij\text{-}betw\ f\ A\ B;\ bij\text{-}betw\ g\ B\ C\ |] \; {=}{=}{>} \; bij\text{-}betw\ (compose\ A\ g\ f)\ A\ C$
⟨*proof*⟩

lemma *bij-betw-restrict-eq* [*simp*]:
$bij\text{-}betw\ (restrict\ f\ A)\ A\ B = bij\text{-}betw\ f\ A\ B$
⟨*proof*⟩

7.5 Extensionality

lemma *extensional-arb*: $[|f \in extensional\ A;\ x \notin A|] \; {=}{=}{>} \; f\ x = arbitrary$
⟨*proof*⟩

lemma *restrict-extensional* [*simp*]: $restrict\ f\ A \in extensional\ A$
⟨*proof*⟩

lemma *compose-extensional* [*simp*]: $compose\ A\ f\ g \in extensional\ A$
⟨*proof*⟩

lemma *extensionalityI*:
$[|\ f \in extensional\ A;\ g \in extensional\ A;$
$!!x.\ x{\in}A \; {=}{=}{>} \; f\ x = g\ x\ |] \; {=}{=}{>} \; f = g$

⟨*proof*⟩

lemma *Inv-funcset*: f ` $A = B$ ==> $(\lambda x{\in}B.\ Inv\ A\ f\ x) : B$ −> A
⟨*proof*⟩

lemma *compose-Inv-id*:
bij-betw $f\ A\ B$ ==> *compose* A $(\lambda y{\in}B.\ Inv\ A\ f\ y)$ f = $(\lambda x{\in}A.\ x)$
⟨*proof*⟩

lemma *compose-id-Inv*:
f ` $A = B$ ==> *compose* B f $(\lambda y{\in}B.\ Inv\ A\ f\ y)$ = $(\lambda x{\in}B.\ x)$
⟨*proof*⟩

7.6 Cardinality

lemma *card-inj*: [|$f \in A{\to}B$; *inj-on* f A; *finite* B|] ==> *card*$(A) \le$ *card*(B)
⟨*proof*⟩

lemma *card-bij*:
[|$f \in A{\to}B$; *inj-on* f A;
$g \in B{\to}A$; *inj-on* g B; *finite* A; *finite* B|] ==> *card*(A) = *card*(B)
⟨*proof*⟩

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*⟩

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* (λx. *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* (λx. *P*)

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

instance *multiset* :: (*type*) {*plus*, *minus*, *zero*} ⟨*proof*⟩

defs (overloaded)
union-def: *M* + *N* == *Abs-multiset* (λa. *Rep-multiset M a* + *Rep-multiset N a*)
diff-def: *M* − *N* == *Abs-multiset* (λa. *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*]: (λa. *0*) ∈ *multiset*
⟨*proof*⟩

lemma *only1-in-multiset* [*simp*]: (λb. *if b* = *a then 1 else 0*) ∈ *multiset*
⟨*proof*⟩

lemma *union-preserves-multiset* [*simp*]:
M ∈ *multiset* ==> *N* ∈ *multiset* ==> (λa. *M a* + *N a*) ∈ *multiset*
⟨*proof*⟩

lemma *diff-preserves-multiset* [*simp*]:
M ∈ *multiset* ==> (λa. *M a* − *N a*) ∈ *multiset*
⟨*proof*⟩

8.2 Algebraic properties of multisets

8.2.1 Union

lemma *union-empty* [*simp*]: *M* + {#} = *M* ∧ {#} + *M* = *M*
⟨*proof*⟩

lemma *union-commute*: $M + N = N + (M::'a\ multiset)$
⟨*proof*⟩

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ multiset))$
⟨*proof*⟩

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
⟨*proof*⟩

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

instance *multiset* :: (*type*) *comm-monoid-add*
⟨*proof*⟩

8.2.2 Difference

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
⟨*proof*⟩

lemma *diff-union-inverse2* [*simp*]: $M + \{\#a\#\} - \{\#a\#\} = M$
⟨*proof*⟩

8.2.3 Count of elements

lemma *count-empty* [*simp*]: $count\ \{\#\}\ a = 0$
⟨*proof*⟩

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

lemma *count-union* [*simp*]: $count\ (M + N)\ a = count\ M\ a + count\ N\ a$
⟨*proof*⟩

lemma *count-diff* [*simp*]: $count\ (M - N)\ a = count\ M\ a - count\ N\ a$
⟨*proof*⟩

8.2.4 Set of elements

lemma *set-of-empty* [*simp*]: $set\text{-}of\ \{\#\} = \{\}$
⟨*proof*⟩

lemma *set-of-single* [*simp*]: $set\text{-}of\ \{\#b\#\} = \{b\}$
⟨*proof*⟩

lemma *set-of-union* [*simp*]: $set\text{-}of\ (M + N) = set\text{-}of\ M \cup set\text{-}of\ N$
⟨*proof*⟩

lemma *set-of-eq-empty-iff* [*simp*]: $(set\text{-}of\ M = \{\}) = (M = \{\#\})$
⟨*proof*⟩

lemma *mem-set-of-iff* [*simp*]: $(x \in \textit{set-of } M) = (x :\# M)$
⟨*proof*⟩

8.2.5 Size

lemma *size-empty* [*simp*]: $\textit{size } \{\#\} = \textit{0}$
⟨*proof*⟩

lemma *size-single* [*simp*]: $\textit{size } \{\#b\#\} = \textit{1}$
⟨*proof*⟩

lemma *finite-set-of* [*iff*]: $\textit{finite } (\textit{set-of } M)$
⟨*proof*⟩

lemma *setsum-count-Int*:
$\textit{finite } A ==> \textit{setsum } (\textit{count } N)\ (A \cap \textit{set-of } N) = \textit{setsum } (\textit{count } N)\ A$
⟨*proof*⟩

lemma *size-union* [*simp*]: $\textit{size } (M + N{::}'a \textit{ multiset}) = \textit{size } M + \textit{size } N$
⟨*proof*⟩

lemma *size-eq-0-iff-empty* [*iff*]: $(\textit{size } M = \textit{0}) = (M = \{\#\})$
⟨*proof*⟩

lemma *size-eq-Suc-imp-elem*: $\textit{size } M = \textit{Suc } n ==> \exists\, a.\ a :\# M$
⟨*proof*⟩

8.2.6 Equality of multisets

lemma *multiset-eq-conv-count-eq*: $(M = N) = (\forall\, a.\ \textit{count } M\ a = \textit{count } N\ a)$
⟨*proof*⟩

lemma *single-not-empty* [*simp*]: $\{\#a\#\} \neq \{\#\} \wedge \{\#\} \neq \{\#a\#\}$
⟨*proof*⟩

lemma *single-eq-single* [*simp*]: $(\{\#a\#\} = \{\#b\#\}) = (a = b)$
⟨*proof*⟩

lemma *union-eq-empty* [*iff*]: $(M + N = \{\#\}) = (M = \{\#\} \wedge N = \{\#\})$
⟨*proof*⟩

lemma *empty-eq-union* [*iff*]: $(\{\#\} = M + N) = (M = \{\#\} \wedge N = \{\#\})$
⟨*proof*⟩

lemma *union-right-cancel* [*simp*]: $(M + K = N + K) = (M = (N{::}'a \textit{ multiset}))$
⟨*proof*⟩

lemma *union-left-cancel* [*simp*]: $(K + M = K + N) = (M = (N{::}'a \textit{ multiset}))$
⟨*proof*⟩

lemma *union-is-single*:
$(M + N = \{\#a\#\}) = (M = \{\#a\#\} \wedge N{=}\{\#\} \vee M = \{\#\} \wedge N = \{\#a\#\})$
⟨*proof*⟩

lemma *single-is-union*:
$(\{\#a\#\} = M + N) = (\{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N)$
⟨*proof*⟩

lemma *add-eq-conv-diff*:
$(M + \{\#a\#\} = N + \{\#b\#\}) =$
$(M = N \wedge a = b \vee M = N - \{\#a\#\} + \{\#b\#\} \wedge N = M - \{\#b\#\} + \{\#a\#\})$
⟨*proof*⟩

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

8.2.7 Intersection

lemma *multiset-inter-count*:
$\mathit{count}\ (A\ \#\cap\ B)\ x = \mathit{min}\ (\mathit{count}\ A\ x)\ (\mathit{count}\ B\ x)$
⟨*proof*⟩

lemma *multiset-inter-commute*: $A\ \#\cap\ B = B\ \#\cap\ A$
⟨*proof*⟩

lemma *multiset-inter-assoc*: $A\ \#\cap\ (B\ \#\cap\ C) = A\ \#\cap\ B\ \#\cap\ C$
⟨*proof*⟩

lemma *multiset-inter-left-commute*: $A\ \#\cap\ (B\ \#\cap\ C) = B\ \#\cap\ (A\ \#\cap\ C)$
⟨*proof*⟩

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

lemma *multiset-union-diff-commute*: $B\ \#\cap\ C = \{\#\} \Longrightarrow A + B - C = A - C + B$
⟨*proof*⟩

8.3 Induction over multisets

lemma *setsum-decr*:
finite F ==> $(0{::}\mathit{nat}) < f\ a$ ==>
$\mathit{setsum}\ (f\ (a := f\ a - 1))\ F = (\mathit{if}\ a{\in}F\ \mathit{then}\ \mathit{setsum}\ f\ F - 1\ \mathit{else}\ \mathit{setsum}\ f\ F)$
⟨*proof*⟩

lemma *rep-multiset-induct-aux*:

assumes $P\ (\lambda a.\ (0{::}nat))$
and $!!f\ b.\ f \in multiset ==> P\ f ==> P\ (f\ (b := f\ b + 1))$
shows $\forall f.\ f \in multiset --> setsum\ f\ \{x.\ 0 < f\ x\} = n --> P\ f$
$\langle proof \rangle$

theorem *rep-multiset-induct*:
$f \in multiset ==> P\ (\lambda a.\ 0) ==>$
$(!!f\ b.\ f \in multiset ==> P\ f ==> P\ (f\ (b := f\ b + 1))) ==> P\ f$
$\langle proof \rangle$

theorem *multiset-induct* [*induct type*: *multiset*]:
assumes *prem1*: $P\ \{\#\}$
and *prem2*: $!!M\ x.\ P\ M ==> P\ (M + \{\#x\#\})$
shows $P\ M$
$\langle proof \rangle$

lemma *MCollect-preserves-multiset*:
$M \in multiset ==> (\lambda x.\ if\ P\ x\ then\ M\ x\ else\ 0) \in multiset$
$\langle proof \rangle$

lemma *count-MCollect* [*simp*]:
$count\ \{\#\ x{:}M.\ P\ x\ \#\}\ a = (if\ P\ a\ then\ count\ M\ a\ else\ 0)$
$\langle proof \rangle$

lemma *set-of-MCollect* [*simp*]: $set\text{-}of\ \{\#\ x{:}M.\ P\ x\ \#\} = set\text{-}of\ M \cap \{x.\ P\ x\}$
$\langle proof \rangle$

lemma *multiset-partition*: $M = \{\#\ x{:}M.\ P\ x\ \#\} + \{\#\ x{:}M.\ \neg\ P\ x\ \#\}$
$\langle proof \rangle$

lemma *add-eq-conv-ex*:
$(M + \{\#a\#\} = N + \{\#b\#\}) =$
$(M = N \wedge a = b \vee (\exists K.\ M = K + \{\#b\#\} \wedge N = K + \{\#a\#\}))$
$\langle proof \rangle$

declare *multiset-typedef* [*simp del*]

8.4 Multiset orderings

8.4.1 Well-foundedness

constdefs
$mult1 :: ('a \times 'a)\ set => ('a\ multiset \times 'a\ multiset)\ set$
$mult1\ r ==$
$\{(N, M).\ \exists\ a\ M0\ K.\ M = M0 + \{\#a\#\} \wedge N = M0 + K \wedge$
$(\forall b.\ b :\#\ K --> (b, a) \in r)\}$

$mult :: ('a \times 'a)\ set => ('a\ multiset \times 'a\ multiset)\ set$
$mult\ r == (mult1\ r)^+$

lemma *not-less-empty* [*iff*]: $(M, \{\#\}) \notin mult1\ r$
⟨*proof*⟩

lemma *less-add*: $(N, M0 + \{\#a\#\}) \in mult1\ r ==>$
$(\exists M. (M, M0) \in mult1\ r \wedge N = M + \{\#a\#\}) \vee$
$(\exists K. (\forall b.\ b :\#\ K --> (b, a) \in r) \wedge N = M0 + K)$
(**concl is** *?case1* (*mult1 r*) $\vee$ *?case2*)
⟨*proof*⟩

lemma *all-accessible*: $wf\ r ==> \forall M.\ M \in acc\ (mult1\ r)$
⟨*proof*⟩

theorem *wf-mult1*: $wf\ r ==> wf\ (mult1\ r)$
⟨*proof*⟩

theorem *wf-mult*: $wf\ r ==> wf\ (mult\ r)$
⟨*proof*⟩

8.4.2 Closure-free presentation

lemma *diff-union-single-conv*: $a :\#\ J ==> I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
⟨*proof*⟩

One direction.

lemma *mult-implies-one-step*:
$trans\ r ==> (M, N) \in mult\ r ==>$
$\exists I\ J\ K.\ N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge$
$(\forall k \in set\text{-}of\ K.\ \exists j \in set\text{-}of\ J.\ (k, j) \in r)$
⟨*proof*⟩

lemma *elem-imp-eq-diff-union*: $a :\#\ M ==> M = M - \{\#a\#\} + \{\#a\#\}$
⟨*proof*⟩

lemma *size-eq-Suc-imp-eq-union*: $size\ M = Suc\ n ==> \exists a\ N.\ M = N + \{\#a\#\}$
⟨*proof*⟩

lemma *one-step-implies-mult-aux*:
$trans\ r ==>$
$\forall I\ J\ K.\ (size\ J = n \wedge J \neq \{\#\} \wedge (\forall k \in set\text{-}of\ K.\ \exists j \in set\text{-}of\ J.\ (k, j) \in r))$
$--> (I + K, I + J) \in mult\ r$
⟨*proof*⟩

lemma *one-step-implies-mult*:
$trans\ r ==> J \neq \{\#\} ==> \forall k \in set\text{-}of\ K.\ \exists j \in set\text{-}of\ J.\ (k, j) \in r$
$==> (I + K, I + J) \in mult\ r$
⟨*proof*⟩

8.4.3 Partial-order properties

instance *multiset* :: (*type*) *ord* ⟨*proof*⟩

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

lemma *trans-base-order*: $trans\ \{(x', x).\ x' < (x::'a::order)\}$
  ⟨*proof*⟩

Irreflexivity.

lemma *mult-irrefl-aux*:
  $finite\ A ==> (\forall x \in A.\ \exists y \in A.\ x < (y::'a::order)) --> A = \{\}$
  ⟨*proof*⟩

lemma *mult-less-not-refl*: $\neg\ M < (M::'a::order\ multiset)$
  ⟨*proof*⟩

lemma *mult-less-irrefl* [*elim*!]: $M < (M::'a::order\ multiset) ==> R$
⟨*proof*⟩

Transitivity.

theorem *mult-less-trans*: $K < M ==> M < N ==> K < (N::'a::order\ multiset)$
  ⟨*proof*⟩

Asymmetry.

theorem *mult-less-not-sym*: $M < N ==> \neg\ N < (M::'a::order\ multiset)$
  ⟨*proof*⟩

theorem *mult-less-asym*:
  $M < N ==> (\neg\ P ==> N < (M::'a::order\ multiset)) ==> P$
  ⟨*proof*⟩

theorem *mult-le-refl* [*iff*]: $M <= (M::'a::order\ multiset)$
⟨*proof*⟩

Anti-symmetry.

theorem *mult-le-antisym*:
  $M <= N ==> N <= M ==> M = (N::'a::order\ multiset)$
  ⟨*proof*⟩

Transitivity.

theorem *mult-le-trans*:
  $K <= M ==> M <= N ==> K <= (N::'a::order\ multiset)$
  ⟨*proof*⟩

theorem *mult-less-le*: $(M < N) = (M <= N \land M \neq (N::'a::order\ multiset))$
⟨*proof*⟩

Partial order.

instance *multiset* :: (*order*) *order*
⟨*proof*⟩

8.4.4 Monotonicity of multiset union

lemma *mult1-union*:
$(B, D) \in mult1\ r ==> trans\ r ==> (C + B, C + D) \in mult1\ r$
⟨*proof*⟩

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

lemma *union-less-mono1*: $B < D ==> B + C < D + (C::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-less-mono*:
$A < C ==> B < D ==> A + B < C + (D::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-le-mono*:
$A <= C ==> B <= D ==> A + B <= C + (D::'a::order\ multiset)$
⟨*proof*⟩

lemma *empty-leI* [*iff*]: $\{\#\} <= (M::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-upper1*: $A <= A + (B::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-upper2*: $B <= A + (B::'a::order\ multiset)$
⟨*proof*⟩

8.5 Link with lists

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

lemma *multiset-of-zero-iff* [*simp*]: (*multiset-of* $x = \{\#\}) = (x = [])$
⟨*proof*⟩

lemma *multiset-of-zero-iff-right*[*simp*]: $(\{\#\} =$ *multiset-of* $x) = (x = [])$
⟨*proof*⟩

lemma *set-of-multiset-of*[*simp*]: *set-of*(*multiset-of* x) $=$ *set* x
⟨*proof*⟩

lemma *mem-set-multiset-eq*: $x \in set\ xs = (x :\#$ *multiset-of* $xs)$

$\langle proof \rangle$

lemma *multiset-of-append*[*simp*]:
$multiset\text{-}of\ (xs\ @\ ys) = multiset\text{-}of\ xs + multiset\text{-}of\ ys$
$\langle proof \rangle$

lemma *surj-multiset-of*: *surj multiset-of*
$\langle proof \rangle$

lemma *set-count-greater-0*: $set\ x = \{a.\ 0 < count\ (multiset\text{-}of\ x)\ a\}$
$\langle proof \rangle$

lemma *distinct-count-atmost-1*:
$distinct\ x = (!\ a.\ count\ (multiset\text{-}of\ x)\ a = (if\ a \in set\ x\ then\ 1\ else\ 0))$
$\langle proof \rangle$

lemma *multiset-of-eq-setD*:
$multiset\text{-}of\ xs = multiset\text{-}of\ ys \Longrightarrow set\ xs = set\ ys$
$\langle proof \rangle$

lemma *set-eq-iff-multiset-of-eq-distinct*:
$[\![distinct\ x;\ distinct\ y]\!]$
$\Longrightarrow (set\ x = set\ y) = (multiset\text{-}of\ x = multiset\text{-}of\ y)$
$\langle proof \rangle$

lemma *set-eq-iff-multiset-of-remdups-eq*:
$(set\ x = set\ y) = (multiset\text{-}of\ (remdups\ x) = multiset\text{-}of\ (remdups\ y))$
$\langle proof \rangle$

lemma *multiset-of-compl-union*[*simp*]:
$multiset\text{-}of\ [x{\in}xs.\ P\ x] + multiset\text{-}of\ [x{\in}xs.\ \neg P\ x] = multiset\text{-}of\ xs$
$\langle proof \rangle$

lemma *count-filter*:
$count\ (multiset\text{-}of\ xs)\ x = length\ [y \in xs.\ y = x]$
$\langle proof \rangle$

8.6 Pointwise ordering induced by count

consts
$mset\text{-}le :: [{}'a\ multiset,\ {}'a\ multiset] \Rightarrow bool$

syntax
$\text{-}mset\text{-}le :: {}'a\ multiset \Rightarrow {}'a\ multiset \Rightarrow bool\quad (\text{- } \le\#\text{ - } [50,51]\ 50)$
translations
$x \le\#\ y == mset\text{-}le\ x\ y$

defs
$mset\text{-}le\text{-}def:\ xs \le\#\ ys == (\forall\ a.\ count\ xs\ a \le count\ ys\ a)$

lemma *mset-le-refl*[*simp*]: $xs \le\# \; xs$
⟨*proof*⟩

lemma *mset-le-trans*: $[\![\; xs \le\# \; ys; \; ys \le\# \; zs \;]\!] \Longrightarrow xs \le\# \; zs$
⟨*proof*⟩

lemma *mset-le-antisym*: $[\![\; xs{\le}\# \; ys; \; ys \le\# \; xs]\!] \Longrightarrow xs = ys$
⟨*proof*⟩

lemma *mset-le-exists-conv*:
$(xs \le\# \; ys) = (\exists \, zs. \; ys = xs + zs)$
⟨*proof*⟩

lemma *mset-le-mono-add-right-cancel*[*simp*]: $(xs + zs \le\# \; ys + zs) = (xs \le\# \; ys)$
⟨*proof*⟩

lemma *mset-le-mono-add-left-cancel*[*simp*]: $(zs + xs \le\# \; zs + ys) = (xs \le\# \; ys)$
⟨*proof*⟩

lemma *mset-le-mono-add*: $[\![\; xs \le\# \; ys; \; vs \le\# \; ws \;]\!] \Longrightarrow xs + vs \le\# \; ys + ws$
⟨*proof*⟩

lemma *mset-le-add-left*[*simp*]: $xs \le\# \; xs + ys$
⟨*proof*⟩

lemma *mset-le-add-right*[*simp*]: $ys \le\# \; xs + ys$
⟨*proof*⟩

lemma *multiset-of-remdups-le*: *multiset-of* (*remdups x*) $\le\#$ *multiset-of x*
⟨*proof*⟩

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:: $(nat * nat) \Rightarrow nat$
nat2-to-nat pair $\equiv$ *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*⟩

lemma
  assumes *eq*: *nat2-to-nat* (u,v) = *nat2-to-nat* (x,y)
  shows *nat2-to-nat-help*: $u+v \leq x+y$
⟨*proof*⟩

theorem *nat2-to-nat-inj*: *inj nat2-to-nat*
⟨*proof*⟩

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*} ⟨*proof*⟩

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*) $\leq$ *n* == $\neg$ ($n < m$)

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*: $Fin\ 0 = 0$
⟨*proof*⟩

lemma *Infty-ne-i0* [*simp*]: $\infty \neq 0$
⟨*proof*⟩

lemma *i0-ne-Infty* [*simp*]: $0 \neq \infty$
⟨*proof*⟩

lemma *iSuc-Fin* [*simp*]: $iSuc\ (Fin\ n) = Fin\ (Suc\ n)$
⟨*proof*⟩

lemma *iSuc-Infty* [*simp*]: $iSuc\ \infty = \infty$
⟨*proof*⟩

lemma *iSuc-ne-0* [*simp*]: $iSuc\ n \neq 0$
⟨*proof*⟩

lemma *iSuc-inject* [*simp*]: $(iSuc\ x = iSuc\ y) = (x = y)$
⟨*proof*⟩

10.3 Ordering relations

lemma *Infty-ilessE* [*elim*!]: $\infty < Fin\ m \Longrightarrow R$
⟨*proof*⟩

lemma *iless-linear*: $m < n \vee m = n \vee n < (m::inat)$
⟨*proof*⟩

lemma *iless-not-refl* [*simp*]: $\neg\ n < (n::inat)$
⟨*proof*⟩

lemma *iless-trans*: $i < j \Longrightarrow j < k \Longrightarrow i < (k::inat)$
⟨*proof*⟩

lemma *iless-not-sym*: $n < m \Longrightarrow \neg\ m < (n::inat)$
⟨*proof*⟩

lemma *Fin-iless-mono* [*simp*]: $(Fin\ n < Fin\ m) = (n < m)$
⟨*proof*⟩

lemma *Fin-iless-Infty* [*simp*]: $Fin\ n < \infty$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *i0-eq* [*simp*]: $((0::inat) < n) = (n \neq 0)$
⟨*proof*⟩

lemma *i0-iless-iSuc* [*simp*]: $0 < iSuc\ n$
⟨*proof*⟩

lemma *not-ilessi0* [*simp*]: $\neg\ n < (0::inat)$
⟨*proof*⟩

lemma *Fin-iless*: $n < Fin\ m ==> \exists k.\ n = Fin\ k$
⟨*proof*⟩

lemma *iSuc-mono* [*simp*]: $(iSuc\ n < iSuc\ m) = (n < m)$
⟨*proof*⟩

lemma *ile-def2*: $(m \leq n) = (m < n \vee m = (n::inat))$
⟨*proof*⟩

lemma *ile-refl* [*simp*]: $n \leq (n::inat)$
⟨*proof*⟩

lemma *ile-trans*: $i \leq j ==> j \leq k ==> i \leq (k::inat)$
⟨*proof*⟩

lemma *ile-iless-trans*: $i \leq j ==> j < k ==> i < (k::inat)$
⟨*proof*⟩

lemma *iless-ile-trans*: $i < j ==> j \leq k ==> i < (k::inat)$
⟨*proof*⟩

lemma *Infty-ub* [*simp*]: $n \leq \infty$
⟨*proof*⟩

lemma *i0-lb* [*simp*]: $(0::inat) \leq n$
⟨*proof*⟩

lemma *Infty-ileE* [*elim*!]: $\infty \leq Fin\ m ==> R$
⟨*proof*⟩

lemma *Fin-ile-mono* [*simp*]: $(Fin\ n \leq Fin\ m) = (n \leq m)$
⟨*proof*⟩

lemma *ilessI1*: $n \leq m ==> n \neq m ==> n < (m::inat)$
⟨*proof*⟩

lemma *ileI1*: $m < n \Longrightarrow iSuc\ m \leq n$
⟨*proof*⟩

lemma *Suc-ile-eq*: $(Fin\ (Suc\ m) \leq n) = (Fin\ m < n)$
⟨*proof*⟩

lemma *iSuc-ile-mono* [*simp*]: $(iSuc\ n \leq iSuc\ m) = (n \leq m)$
⟨*proof*⟩

lemma *iless-Suc-eq* [*simp*]: $(Fin\ m < iSuc\ n) = (Fin\ m \leq n)$
⟨*proof*⟩

lemma *not-iSuc-ilei0* [*simp*]: $\neg\ iSuc\ n \leq 0$
⟨*proof*⟩

lemma *ile-iSuc* [*simp*]: $n \leq iSuc\ n$
⟨*proof*⟩

lemma *Fin-ile*: $n \leq Fin\ m \Longrightarrow \exists k.\ n = Fin\ k$
⟨*proof*⟩

lemma *chain-incr*: $\forall i.\ \exists j.\ Y\ i < Y\ j \Longrightarrow \exists j.\ Fin\ k < Y\ j$
⟨*proof*⟩

end

11 Nested-Environment: Nested environments

theory *Nested-Environment*
imports *Main*
begin

Consider a partial function $e :: 'a => 'b\ option$; 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*
⟨*proof*⟩

theorem *lookup-val-cons*: *lookup (Val a) (x # xs) = None*
⟨*proof*⟩

theorem *lookup-env-cons*:
lookup (Env b es) (x # xs) =
  (case es x of
    None => None
  | Some e => lookup e xs)
⟨*proof*⟩

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)))
  ⟨proof⟩
```

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⟩

theorem lookup-append-some:
  !!env e. lookup env xs = Some e ==> lookup env (xs @ ys) = lookup e ys
  (is PROP ?P xs)
⟨proof⟩
```

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⟩
```

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⟩
```

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
  ⟨proof⟩

theorem update-nil-some: update [] (Some e) env = e
  ⟨proof⟩

theorem update-cons-val: update (x # xs) opt (Val a) = Val a
  ⟨proof⟩

theorem update-cons-nil-env:
    update [x] opt (Env b es) = Env b (es (x := opt))
  ⟨proof⟩

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))))
  ⟨proof⟩

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)))))))
⟨proof⟩
```

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⟩
```

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⟩
```

```
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⟩
```

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*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *xperm-empty-imp*: [] <~~> *ys* ==> *ys* = []
  ⟨*proof*⟩

This more general theorem is easier to understand!

lemma *perm-length*: *xs* <~~> *ys* ==> *length xs* = *length ys*
  ⟨*proof*⟩

lemma *perm-empty-imp*: [] <~~> *xs* ==> *xs* = []
  ⟨*proof*⟩

lemma *perm-sym*: *xs* <~~> *ys* ==> *ys* <~~> *xs*

⟨proof⟩

lemma *perm-mem* [*rule-format*]: *xs* <~~> *ys* ==> *x mem xs* --> *x mem ys*
⟨proof⟩

12.2 Ways of making new permutations

We can insert the head anywhere in the list.

lemma *perm-append-Cons*: *a* # *xs* @ *ys* <~~> *xs* @ *a* # *ys*
⟨proof⟩

lemma *perm-append-swap*: *xs* @ *ys* <~~> *ys* @ *xs*
⟨proof⟩

lemma *perm-append-single*: *a* # *xs* <~~> *xs* @ [*a*]
⟨proof⟩

lemma *perm-rev*: *rev xs* <~~> *xs*
⟨proof⟩

lemma *perm-append1*: *xs* <~~> *ys* ==> *l* @ *xs* <~~> *l* @ *ys*
⟨proof⟩

lemma *perm-append2*: *xs* <~~> *ys* ==> *xs* @ *l* <~~> *ys* @ *l*
⟨proof⟩

12.3 Further results

lemma *perm-empty* [*iff*]: ([] <~~> *xs*) = (*xs* = [])
⟨proof⟩

lemma *perm-empty2* [*iff*]: (*xs* <~~> []) = (*xs* = [])
⟨proof⟩

lemma *perm-sing-imp* [*rule-format*]: *ys* <~~> *xs* ==> *xs* = [*y*] --> *ys* = [*y*]
⟨proof⟩

lemma *perm-sing-eq* [*iff*]: (*ys* <~~> [*y*]) = (*ys* = [*y*])
⟨proof⟩

lemma *perm-sing-eq2* [*iff*]: ([*y*] <~~> *ys*) = (*ys* = [*y*])
⟨proof⟩

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*
⟨*proof*⟩

lemma *remove-commute*: *remove x (remove y l) = remove y (remove x l)*
⟨*proof*⟩

lemma *multiset-of-remove*[*simp*]:
multiset-of (remove a x) = multiset-of x − {#a#}
⟨*proof*⟩

Congruence rule

lemma *perm-remove-perm*: *xs <~~> ys ==> remove z xs <~~> remove z ys*
⟨*proof*⟩

lemma *remove-hd* [*simp*]: *remove z (z # xs) = xs*
⟨*proof*⟩

lemma *cons-perm-imp-perm*: *z # xs <~~> z # ys ==> xs <~~> ys*
⟨*proof*⟩

lemma *cons-perm-eq* [*iff*]: *(z#xs <~~> z#ys) = (xs <~~> ys)*
⟨*proof*⟩

lemma *append-perm-imp-perm*: *!!xs ys. zs @ xs <~~> zs @ ys ==> xs <~~> ys*
⟨*proof*⟩

lemma *perm-append1-eq* [*iff*]: *(zs @ xs <~~> zs @ ys) = (xs <~~> ys)*
⟨*proof*⟩

lemma *perm-append2-eq* [*iff*]: *(xs @ zs <~~> ys @ zs) = (xs <~~> ys)*
⟨*proof*⟩

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

lemma *multiset-of-le-perm-append*:
(multiset-of xs ≤# multiset-of ys) = (∃ zs. xs @ zs <~~> ys)
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *prime-imp-relprime*: *prime p* ==> ¬ *p dvd n* ==> *gcd* (*p*, *n*) = *1*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *prime-dvd-square*: *prime p* ==> *p dvd m*ˆ*Suc* (*Suc 0*) ==> *p dvd m*
  ⟨*proof*⟩

lemma *prime-dvd-power-two*: *prime p* ==> *p dvd* m^2 ==> *p dvd m*
  ⟨*proof*⟩

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 ∼ :: ′*a* => ′*a* => *bool*.

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

axclass $equiv \subseteq eqv$
  $equiv\text{-}refl$ $[intro]$: $x \sim x$
  $equiv\text{-}trans$ $[trans]$: $x \sim y ==> y \sim z ==> x \sim z$
  $equiv\text{-}sym$ $[sym]$: $x \sim y ==> y \sim x$

lemma $equiv\text{-}not\text{-}sym$ $[sym]$: $\neg (x \sim y) ==> \neg (y \sim (x::'a::equiv))$
$\langle proof \rangle$

lemma $not\text{-}equiv\text{-}trans1$ $[trans]$: $\neg (x \sim y) ==> y \sim z ==> \neg (x \sim (z::'a::equiv))$
$\langle proof \rangle$

lemma $not\text{-}equiv\text{-}trans2$ $[trans]$: $x \sim y ==> \neg (y \sim z) ==> \neg (x \sim (z::'a::equiv))$
$\langle proof \rangle$

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

typedef $'a\ quot = \{\{x.\ a \sim x\} \mid a::'a::eqv.\ True\}$
  $\langle proof \rangle$

lemma $quotI$ $[intro]$: $\{x.\ a \sim x\} \in quot$
  $\langle proof \rangle$

lemma $quotE$ $[elim]$: $R \in quot ==> (!!a.\ R = \{x.\ a \sim x\} ==> C) ==> C$
  $\langle proof \rangle$

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

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

theorem $quot\text{-}exhaust$: $\exists\, a.\ A = \lfloor a \rfloor$
$\langle proof \rangle$

lemma $quot\text{-}cases$ $[cases\ type: quot]$: $(!!a.\ A = \lfloor a \rfloor ==> C) ==> C$
  $\langle proof \rangle$

14.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

theorem $quot\text{-}equality$ $[iff?]$: $(\lfloor a \rfloor = \lfloor b \rfloor) = (a \sim b)$
$\langle proof \rangle$

14.3 Picking representing elements

constdefs

pick :: $'a$::*equiv quot* => $'a$
pick A == *SOME* a. $A = \lfloor a \rfloor$

theorem *pick-equiv* [*intro*]: *pick* $\lfloor a \rfloor \sim a$
⟨*proof*⟩

theorem *pick-inverse* [*intro*]: $\lfloor$*pick* $A\rfloor = A$
⟨*proof*⟩

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*⟩

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*⟩

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
⟨*proof*⟩

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* :: ($'a$ => *bool*) × ($'a$ => $'a$) × $'a$ => $'a$
recdef (**permissive**) *while-aux*

$\mathit{same\text{-}fst}\ (\lambda b.\ \mathit{True})\ (\lambda b.\ \mathit{same\text{-}fst}\ (\lambda c.\ \mathit{True})\ (\lambda c.$
$\{(t,\ s).\ \ b\ s \wedge c\ s = t \wedge$
$\neg\ (\exists f.\ f\ (0{::}nat) = s \wedge (\forall i.\ b\ (f\ i) \wedge c\ (f\ i) = f\ (i + 1)))\}))$
$\mathit{while\text{-}aux}\ (b,\ c,\ s) =$
$(\mathit{if}\ (\exists f.\ f\ (0{::}nat) = s \wedge (\forall i.\ b\ (f\ i) \wedge c\ (f\ i) = f\ (i + 1)))$
$\mathit{then}\ \mathit{arbitrary}$
$\mathit{else}\ \mathit{if}\ b\ s\ \mathit{then}\ \mathit{while\text{-}aux}\ (b,\ c,\ c\ s)$
$\mathit{else}\ s)$

recdef-tc *while-aux-tc*: *while-aux*
⟨*proof*⟩

constdefs
$\mathit{while} :: ('a => \mathit{bool}) => ('a => 'a) => 'a => 'a$
$\mathit{while}\ b\ c\ s == \mathit{while\text{-}aux}\ (b,\ c,\ s)$

lemma *while-aux-unfold*:
$\mathit{while\text{-}aux}\ (b,\ c,\ s) =$
$(\mathit{if}\ \exists f.\ f\ (0{::}nat) = s \wedge (\forall i.\ b\ (f\ i) \wedge c\ (f\ i) = f\ (i + 1))$
$\mathit{then}\ \mathit{arbitrary}$
$\mathit{else}\ \mathit{if}\ b\ s\ \mathit{then}\ \mathit{while\text{-}aux}\ (b,\ c,\ c\ s)$
$\mathit{else}\ s)$
⟨*proof*⟩

The recursion equation for *while*: directly executable!

theorem *while-unfold* [*code*]:
$\mathit{while}\ b\ c\ s = (\mathit{if}\ b\ s\ \mathit{then}\ \mathit{while}\ b\ c\ (c\ s)\ \mathit{else}\ s)$
⟨*proof*⟩

hide *const while-aux*

lemma *def-while-unfold*: **assumes** *fdef*: $f == \mathit{while}\ \mathit{test}\ \mathit{do}$
shows $f\ x = (\mathit{if}\ \mathit{test}\ x\ \mathit{then}\ f(\mathit{do}\ x)\ \mathit{else}\ x)$
⟨*proof*⟩

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

theorem *while-rule-lemma*[*rule-format*]:
$[|\ !!s.\ P\ s ==> b\ s ==> P\ (c\ s);$
$!!s.\ P\ s ==> \neg\ b\ s ==> Q\ s;$
$\mathit{wf}\ \{(t,\ s).\ P\ s \wedge b\ s \wedge t = c\ s\}\ |] ==>$
$P\ s\ {-}{-}{>}\ Q\ (\mathit{while}\ b\ c\ s)$
⟨*proof*⟩

theorem *while-rule*:
$[|\ P\ s;$
$!!s.\ [|\ P\ s;\ b\ s\ \ |] ==> P\ (c\ s);$
$!!s.\ [|\ P\ s;\ \neg\ b\ s\ \ |] ==> Q\ s;$
$\mathit{wf}\ r;$
$!!s.\ [|\ P\ s;\ b\ s\ \ |] ==> (c\ s,\ s) \in r\ |] ==>$

  Q (*while* b c s)
⟨*proof*⟩

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* ($\lambda(A, fA).\ A \neq fA$) ($\lambda(A, fA).\ (fA, f\,fA)$) ({}, f {}))
⟨*proof*⟩

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))
⟨*proof*⟩

theorem P (*lfp* (λN::*int set*. {*0*} ∪ {(n + *2*) *mod 6* | n. $n \in N$})) =
  P {*0*, *4*, *2*}
⟨*proof*⟩

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 $y \leq z$
  ⟨*proof*⟩

lemma *max-mono*:
  fixes x :: *'a*::*linorder*
  assumes *mf*: *mono* f
  shows *max* (f x) (f y) $\leq$ f (*max* x y)
⟨*proof*⟩

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

lemma *int-nat-two-exp*: *2* ^ k = *int* (*2* ^ k)
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *bitand-cancel* [*simp*]: (*b bitand b*) = *b*
⟨*proof*⟩

lemma *bitor-cancel* [*simp*]: (*b bitor b*) = *b*
⟨*proof*⟩

lemma *bitxor-cancel* [*simp*]: (*b bitxor b*) = **0**
⟨*proof*⟩

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* = [] ==> *P w*
and *zero*: !!*bs*. *w* = **0** # *bs* ==> *P w*
and *one*: !!*bs*. *w* = **1** # *bs* ==> *P w*
shows *P w*
⟨*proof*⟩

lemma *bit-list-induct*:
assumes *empty*: *P* []
and *zero*: !!*bs*. *P bs* ==> *P* (**0**#*bs*)
and *one*: !!*bs*. *P bs* ==> *P* (**1**#*bs*)
shows *P w*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bv-not-Nil* [*simp*]: *bv-not* [] = []
⟨*proof*⟩

lemma *bv-not-Cons* [*simp*]: *bv-not* (*b*#*bs*) = (*bitnot b*) # *bv-not bs*
⟨*proof*⟩

lemma *bv-not-bv-not* [*simp*]: *bv-not* (*bv-not w*) = *w*
⟨*proof*⟩

lemma *bv-msb-Nil* [*simp*]: *bv-msb* [] = **0**
⟨*proof*⟩

lemma *bv-msb-Cons* [*simp*]: *bv-msb* (*b*#*bs*) = *b*
⟨*proof*⟩

lemma *bv-msb-bv-not* [*simp*]: *0* < *length w* ==> *bv-msb* (*bv-not w*) = (*bitnot* (*bv-msb w*))
⟨*proof*⟩

lemma *bv-msb-one-length* [*simp*,*intro*]: *bv-msb w* = **1** ==> *0* < *length w*
⟨*proof*⟩

lemma *length-bv-not* [*simp*]: *length* (*bv-not w*) = *length w*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bv-to-nat-helper* [*simp*]: *bv-to-nat* (*b* # *bs*) = *bitval b* ∗ *2* ˆ *length bs* + *bv-to-nat bs*
⟨*proof*⟩

lemma *bv-to-nat0* [*simp*]: *bv-to-nat* (**0**#*bs*) = *bv-to-nat bs*
⟨*proof*⟩

lemma *bv-to-nat1* [*simp*]: *bv-to-nat* (**1**#*bs*) = *2* ˆ *length bs* + *bv-to-nat bs*
⟨*proof*⟩

lemma *bv-to-nat-upper-range*: *bv-to-nat w* < *2* ˆ *length w*
⟨*proof*⟩

lemma *bv-extend-longer* [*simp*]:
assumes *wn*: *n* ≤ *length w*
shows *bv-extend n b w* = *w*
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *rem-initial-equal*:
assumes *p*: *length (rem-initial b w) = length w*
shows *rem-initial b w = w*
⟨*proof*⟩

lemma *bv-extend-rem-initial*: *bv-extend (length w) b (rem-initial b w) = w*
⟨*proof*⟩

lemma *rem-initial-append1*:
assumes *rem-initial b xs ~= []*
shows *rem-initial b (xs @ ys) = rem-initial b xs @ ys*
⟨*proof*⟩

lemma *rem-initial-append2*:
assumes *rem-initial b xs = []*
shows *rem-initial b (xs @ ys) = rem-initial b ys*
⟨*proof*⟩

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

lemma *norm-unsigned-Nil* [*simp*]: *norm-unsigned [] = []*
⟨*proof*⟩

lemma *norm-unsigned-Cons0* [*simp*]: *norm-unsigned* (**0**#*bs*) = *norm-unsigned bs*
⟨*proof*⟩

lemma *norm-unsigned-Cons1* [*simp*]: *norm-unsigned* (**1**#*bs*) = **1**#*bs*
⟨*proof*⟩

lemma *norm-unsigned-idem* [*simp*]: *norm-unsigned (norm-unsigned w) = norm-unsigned w*
⟨*proof*⟩

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* = []
⟨*proof*⟩

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*⟩

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*⟩

lemma *unfold-nat-to-bv-helper*:
nat-to-bv-helper b l = *nat-to-bv-helper b* [] @ *l*
⟨*proof*⟩

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*⟩

lemma *bv-to-nat-dist-append*: *bv-to-nat* (*l1* @ *l2*) = *bv-to-nat l1* ∗ *2* ˆ *length l2* + *bv-to-nat l2*
⟨*proof*⟩

lemma *bv-nat-bv* [*simp*]: *bv-to-nat* (*nat-to-bv n*) = *n*
⟨*proof*⟩

lemma *bv-to-nat-type* [*simp*]: *bv-to-nat* (*norm-unsigned w*) = *bv-to-nat w*
⟨*proof*⟩

lemma *length-norm-unsigned-le* [*simp*]: *length* (*norm-unsigned w*) ≤ *length w*
⟨*proof*⟩

lemma *bv-to-nat-rew-msb*: *bv-msb w* = **1** ==> *bv-to-nat w* = *2* ˆ (*length w* − *1*) + *bv-to-nat* (*tl w*)
⟨*proof*⟩

lemma *norm-unsigned-result*: *norm-unsigned xs* = [] ∨ *bv-msb* (*norm-unsigned xs*) = **1**
⟨*proof*⟩

lemma *norm-empty-bv-to-nat-zero*:
assumes *nw*: *norm-unsigned w* = []
shows *bv-to-nat w* = *0*
⟨*proof*⟩

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*⟩

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

lemma *norm-unsigned-length* [*intro*!]: *length* (*norm-unsigned w*) ≤ *length w*
⟨*proof*⟩

lemma *norm-unsigned-equal*: *length* (*norm-unsigned w*) = *length w* ==> *norm-unsigned w* = *w*
⟨*proof*⟩

lemma *bv-extend-norm-unsigned*: *bv-extend* (*length w*) **0** (*norm-unsigned w*) = *w*
⟨*proof*⟩

lemma *norm-unsigned-append1* [*simp*]: *norm-unsigned xs* ≠ [] ==> *norm-unsigned* (*xs* @ *ys*) = *norm-unsigned xs* @ *ys*
⟨*proof*⟩

lemma *norm-unsigned-append2* [*simp*]: *norm-unsigned xs* = [] ==> *norm-unsigned* (*xs* @ *ys*) = *norm-unsigned ys*
⟨*proof*⟩

lemma *bv-to-nat-zero-imp-empty* [*rule-format*]:
bv-to-nat w = *0* ⟶ *norm-unsigned w* = []
⟨*proof*⟩

lemma *bv-to-nat-nzero-imp-nempty*:
assumes *bv-to-nat w* ≠ *0*
shows *norm-unsigned w* ≠ []
⟨*proof*⟩

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*⟩

lemma *nat-helper2*: *nat-to-bv* (*2* ^ *length xs* + *bv-to-nat xs*) = **1** # *xs*
⟨*proof*⟩

lemma *nat-bv-nat* [*simp*]: *nat-to-bv* (*bv-to-nat w*) = *norm-unsigned w*

⟨*proof*⟩

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*⟩

lemma *norm-unsigned-nat-to-bv* [*simp*]:
norm-unsigned (*nat-to-bv n*) = *nat-to-bv n*
⟨*proof*⟩

lemma *length-nat-to-bv-upper-limit*:
assumes *nk*: $n \leq 2 \hat{}\ k - 1$
shows *length* (*nat-to-bv n*) $\leq k$
⟨*proof*⟩

lemma *length-nat-to-bv-lower-limit*:
assumes *nk*: $2 \hat{}\ k \leq n$
shows $k <$ *length* (*nat-to-bv n*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bv-add-type2* [*simp*]: *bv-add w1* (*norm-unsigned w2*) = *bv-add w1 w2*
⟨*proof*⟩

lemma *bv-add-returntype* [*simp*]: *norm-unsigned* (*bv-add w1 w2*) = *bv-add w1 w2*
⟨*proof*⟩

lemma *bv-add-length*: *length* (*bv-add w1 w2*) ≤ *Suc* (*max* (*length w1*) (*length w2*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bv-mult-type2* [*simp*]: *bv-mult w1* (*norm-unsigned w2*) = *bv-mult w1 w2*
⟨*proof*⟩

lemma *bv-mult-returntype* [*simp*]: *norm-unsigned* (*bv-mult w1 w2*) = *bv-mult w1 w2*
⟨*proof*⟩

lemma *bv-mult-length*: *length* (*bv-mult w1 w2*) ≤ *length w1* + *length w2*
⟨*proof*⟩

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**] = []
⟨*proof*⟩

lemma *norm-signed1* [*simp*]: *norm-signed* [**1**] = [**1**]
⟨*proof*⟩

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

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

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

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

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*))
⟨*proof*⟩

lemma *int-to-bv-lt0* [*simp*]: *n* < *0* ==> *int-to-bv n* = *norm-signed* (*bv-not* (**0**#*nat-to-bv*

(nat (−n− 1))))
⟨*proof*⟩

lemma *norm-signed-idem* [*simp*]: *norm-signed* (*norm-signed w*) = *norm-signed w*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bv-to-int-Cons0* [*simp*]: *bv-to-int* (**0**#*bs*) = *int* (*bv-to-nat bs*)
⟨*proof*⟩

lemma *bv-to-int-Cons1* [*simp*]: *bv-to-int* (**1**#*bs*) = − *int* (*bv-to-nat* (*bv-not bs*) + *1*)
⟨*proof*⟩

lemma *bv-to-int-type* [*simp*]: *bv-to-int* (*norm-signed w*) = *bv-to-int w*
⟨*proof*⟩

lemma *bv-to-int-upper-range*: *bv-to-int w* < *2* ^ (*length w* − *1*)
⟨*proof*⟩

lemma *bv-to-int-lower-range*: − (*2* ^ (*length w* − *1*)) ≤ *bv-to-int w*
⟨*proof*⟩

lemma *int-bv-int* [*simp*]: *int-to-bv* (*bv-to-int w*) = *norm-signed w*
⟨*proof*⟩

lemma *bv-int-bv* [*simp*]: *bv-to-int* (*int-to-bv i*) = *i*
⟨*proof*⟩

lemma *bv-msb-norm* [*simp*]: *bv-msb* (*norm-signed w*) = *bv-msb w*
⟨*proof*⟩

lemma *norm-signed-length*: *length* (*norm-signed w*) ≤ *length w*
⟨*proof*⟩

lemma *norm-signed-equal*: *length* (*norm-signed w*) = *length w* ==> *norm-signed w* = *w*
⟨*proof*⟩

lemma *bv-extend-norm-signed*: *bv-msb w* = *b* ==> *bv-extend* (*length w*) *b* (*norm-signed w*) = *w*
⟨*proof*⟩

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*⟩

lemma *int-to-bv-returntype* [*simp*]: *norm-signed* (*int-to-bv w*) = *int-to-bv w*
⟨*proof*⟩

lemma *bv-to-int-msb0*: $0 \leq$ *bv-to-int w1* ==> *bv-msb w1* = **0**
⟨*proof*⟩

lemma *bv-to-int-msb1*: *bv-to-int w1* < *0* ==> *bv-msb w1* = **1**
⟨*proof*⟩

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*⟩

lemma *norm-signed-result*: *norm-signed w* = [] ∨ *norm-signed w* = [**1**] ∨ *bv-msb* (*norm-signed w*) ≠ *bv-msb* (*tl* (*norm-signed w*))
⟨*proof*⟩

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*⟩

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*⟩

lemma *pos-length-pos*:
assumes *i0*: *0* < *bv-to-int w*
shows *0* < *length w*
⟨*proof*⟩

lemma *neg-length-pos*:
assumes *i0*: *bv-to-int w* < −*1*
shows *0* < *length w*
⟨*proof*⟩

lemma *length-int-to-bv-lower-limit-gt0*:
assumes *wk*: *2* ^ (*k* − *1*) ≤ *i*
shows *k* < *length* (*int-to-bv i*)

⟨*proof*⟩

lemma *length-int-to-bv-upper-limit-lem1*:
assumes $w1$: $i < -1$
and wk: $-(2 \hat{} (k - 1)) \leq i$
shows $length\ (int\text{-}to\text{-}bv\ i) \leq k$
⟨*proof*⟩

lemma *length-int-to-bv-lower-limit-lem1*:
assumes wk: $i < -(2 \hat{} (k - 1))$
shows $k < length\ (int\text{-}to\text{-}bv\ i)$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *utos-returntype* [*simp*]: *norm-signed* (*utos w*) = *utos w*
⟨*proof*⟩

lemma *utos-length*: *length* (*utos w*) ≤ *Suc* (*length w*)
⟨*proof*⟩

lemma *bv-to-int-utos*: *bv-to-int* (*utos w*) = *int* (*bv-to-nat w*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bv-uminus-returntype* [*simp*]: *norm-signed* (*bv-uminus w*) = *bv-uminus w*
⟨*proof*⟩

lemma *bv-uminus-length*: *length* (*bv-uminus w*) ≤ *Suc* (*length w*)
⟨*proof*⟩

lemma *bv-uminus-length-utos*: *length* (*bv-uminus* (*utos w*)) ≤ *Suc* (*length w*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bv-sadd-type2* [*simp*]: *bv-sadd w1* (*norm-signed w2*) = *bv-sadd w1 w2*
⟨*proof*⟩

lemma *bv-sadd-returntype* [*simp*]: *norm-signed* (*bv-sadd w1 w2*) = *bv-sadd w1 w2*
⟨*proof*⟩

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*⟩

lemma *bv-sadd-length*: *length* (*bv-sadd w1 w2*) ≤ *Suc* (*max* (*length w1*) (*length w2*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bv-sub-type2* [*simp*]: *bv-sub w1* (*norm-signed w2*) = *bv-sub w1 w2*
⟨*proof*⟩

lemma *bv-sub-returntype* [*simp*]: *norm-signed* (*bv-sub w1 w2*) = *bv-sub w1 w2*
⟨*proof*⟩

lemma *bv-sub-length*: *length* (*bv-sub w1 w2*) ≤ *Suc* (*max* (*length w1*) (*length w2*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bv-smult-type2* [*simp*]: *bv-smult w1* (*norm-signed w2*) = *bv-smult w1 w2*
⟨*proof*⟩

lemma *bv-smult-returntype* [*simp*]: *norm-signed* (*bv-smult w1 w2*) = *bv-smult w1 w2*
⟨*proof*⟩

lemma *bv-smult-length*: *length* (*bv-smult w1 w2*) ≤ *length w1* + *length w2*
⟨*proof*⟩

lemma *bv-msb-one*: *bv-msb w* = **1** ==> *0* < *bv-to-nat w*
⟨*proof*⟩

lemma *bv-smult-length-utos*: *length* (*bv-smult* (*utos w1*) *w2*) ≤ *length w1* + *length w2*
⟨*proof*⟩

lemma *bv-smult-sym*: *bv-smult w1 w2* = *bv-smult w2 w1*
⟨*proof*⟩

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*⟩

lemma *bv-chop-append*: *bv-chop* (*w1* @ *w2*) (*length w2*) = (*w1*,*w2*)
⟨*proof*⟩

lemma *append-bv-chop-id*: *fst* (*bv-chop w l*) @ *snd* (*bv-chop w l*) = *w*
⟨*proof*⟩

lemma *bv-chop-length-fst* [*simp*]: *length* (*fst* (*bv-chop w i*)) = *length w* − *i*
⟨*proof*⟩

lemma *bv-chop-length-snd* [*simp*]: *length* (*snd* (*bv-chop w i*)) = *min i* (*length w*)
⟨*proof*⟩

lemma *bv-slice-length* [*simp*]: [| *j* ≤ *i*; *i* < *length w* |] ==> *length* (*bv-slice w* (*i*,*j*)) = *i* − *j* + *1*
⟨*proof*⟩

constdefs
length-nat :: *nat* => *nat*
length-nat x == *LEAST n. x* < *2* ˆ *n*

lemma *length-nat*: *length* (*nat-to-bv n*) = *length-nat n*
⟨*proof*⟩

lemma *length-nat-0* [*simp*]: *length-nat 0* = *0*
⟨*proof*⟩

lemma *length-nat-non0*:
assumes *n0*: *0* < *n*
shows *length-nat n* = *Suc* (*length-nat* (*n div 2*))
⟨*proof*⟩

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*⟩

lemma *length-int-0* [*simp*]: *length-int 0* = *0*
⟨*proof*⟩

lemma *length-int-gt0*: *0* < *i* ==> *length-int i* = *Suc* (*length-nat* (*nat i*))
⟨*proof*⟩

lemma *length-int-lt0*: *i* < *0* ==> *length-int i* = *Suc* (*length-nat* (*nat* (− *i*) − *1*))
⟨*proof*⟩

lemma *bv-chopI*: [| *w* = *w1* @ *w2* ; *i* = *length w2* |] ==> *bv-chop w i* = (*w1*,*w2*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *bv-to-nat-extend* [*simp*]: *bv-to-nat* (*bv-extend n* **0** *w*) = *bv-to-nat w*
⟨*proof*⟩

lemma *bv-msb-extend-same* [*simp*]: *bv-msb w* = *b* ==> *bv-msb* (*bv-extend n b w*) = *b*
⟨*proof*⟩

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*⟩

lemma *length-nat-mono* [*simp*]: *x* ≤ *y* ==> *length-nat x* ≤ *length-nat y*
⟨*proof*⟩

lemma *length-nat-mono-int* [*simp*]: *x* ≤ *y* ==> *length-nat x* ≤ *length-nat y*
⟨*proof*⟩

lemma *length-nat-pos* [*simp*,*intro*!]: *0* < *x* ==> *0* < *length-nat x*
⟨*proof*⟩

lemma *length-int-mono-gt0*: [| *0* ≤ *x* ; *x* ≤ *y* |] ==> *length-int x* ≤ *length-int y*
⟨*proof*⟩

lemma *length-int-mono-lt0*: [| *x* ≤ *y* ; *y* ≤ *0* |] ==> *length-int y* ≤ *length-int x*
⟨*proof*⟩

lemmas [*simp*] = *length-nat-non0*

lemma *nat-to-bv* (*number-of Numeral.Pls*) = []
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fast-bv-to-nat-Cons1*: *fast-bv-to-nat-helper* (**1**#*bs*) *bin* = *fast-bv-to-nat-helper bs* (*bin BIT bit.B1*)
⟨*proof*⟩

lemma *fast-bv-to-nat-def*: *bv-to-nat bs == number-of (fast-bv-to-nat-helper bs Numeral.Pls)*
⟨*proof*⟩

declare *fast-bv-to-nat-Cons* [*simp del*]
declare *fast-bv-to-nat-Cons0* [*simp*]
declare *fast-bv-to-nat-Cons1* [*simp*]

⟨*ML*⟩

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)*
⟨*proof*⟩

lemma *bv-mapzip-Nil* [*simp*]: *bv-mapzip f* [] [] = []
⟨*proof*⟩

lemma *bv-mapzip-Cons* [*simp*]: *length w1 = length w2 ==> bv-mapzip f (x#w1) (y#w2) = f x y # bv-mapzip f w1 w2*
⟨*proof*⟩

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 == $\{F.\ F \subseteq S\ \&\ (\forall x \in F.\ \forall y \in F.\ x \subseteq y \mid y \subseteq x)\}$

super :: [*'a set set,'a set set*] => *'a set set set*
super S c == $\{d.\ d \in chain\ S\ \&\ c \subset d\}$

maxchain :: *'a set set => 'a set set set*
maxchain S == $\{c.\ c \in chain\ S\ \&\ super\ S\ c = \{\}\}$

succ :: $['a\ set\ set,'a\ set\ set] => 'a\ set\ set$
succ S c ==
if $c \notin$ *chain S* | $c \in$ *maxchain S*
then c else SOME $c'.\ c' \in$ *super S c*

consts
TFin :: $'a\ set\ set => 'a\ set\ set\ set$

inductive *TFin S*
intros
succI: $x \in$ *TFin S* ==> *succ S x* $\in$ *TFin S*
Pow-UnionI: $Y \in$ *Pow(TFin S)* ==> *Union(Y)* $\in$ *TFin S*
monos *Pow-mono*

17.1 Mathematical Preamble

lemma *Union-lemma0*:
$(\forall x \in C.\ x \subseteq A \mid B \subseteq x)$ ==> *Union(C)<=A* | $B \subseteq$ *Union(C)*
⟨*proof*⟩

This is theorem *increasingD2* of ZF/Zorn.thy

lemma *Abrial-axiom1*: $x \subseteq$ *succ S x*
⟨*proof*⟩

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)* |]
==> $P(n)$
⟨*proof*⟩

lemma *succ-trans*: $x \subseteq y$ ==> $x \subseteq$ *succ S y*
⟨*proof*⟩

Lemma 1 of section 3.1

lemma *TFin-linear-lemma1*:
[| $n \in$ *TFin S*; $m \in$ *TFin S*;
$\forall x \in$ *TFin S*. $x \subseteq m$ --> $x = m$ | *succ S x* $\subseteq m$
|] ==> $n \subseteq m$ | *succ S m* $\subseteq n$
⟨*proof*⟩

Lemma 2 of section 3.2

lemma *TFin-linear-lemma2*:
$m \in$ *TFin S* ==> $\forall n \in$ *TFin S*. $n \subseteq m$ --> $n=m$ | *succ S n* $\subseteq m$
⟨*proof*⟩

Re-ordering the premises of Lemma 2

lemma *TFin-subsetD*:
$[| \ n \subseteq m; \ m \in TFin\ S; \ n \in TFin\ S \ |] ==> n{=}m \mid succ\ S\ n \subseteq m$
⟨*proof*⟩

Consequences from section 3.3 – Property 3.2, the ordering is total

lemma *TFin-subset-linear*: $[| \ m \in TFin\ S; \ n \in TFin\ S|] ==> n \subseteq m \mid m \subseteq n$
⟨*proof*⟩

Lemma 3 of section 3.3

lemma *eq-succ-upper*: $[| \ n \in TFin\ S; \ m \in TFin\ S; \ m = succ\ S\ m \ |] ==> n \subseteq m$
⟨*proof*⟩

Property 3.3 of section 3.3

lemma *equal-succ-Union*: $m \in TFin\ S ==> (m = succ\ S\ m) = (m = Union(TFin\ S))$
⟨*proof*⟩

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*: $(\{\} :: 'a\ set\ set) \in chain\ S$
⟨*proof*⟩

lemma *super-subset-chain*: $super\ S\ c \subseteq chain\ S$
⟨*proof*⟩

lemma *maxchain-subset-chain*: $maxchain\ S \subseteq chain\ S$
⟨*proof*⟩

lemma *mem-super-Ex*: $c \in chain\ S - maxchain\ S ==> ?\ d.\ d \in super\ S\ c$
⟨*proof*⟩

lemma *select-super*: $c \in chain\ S - maxchain\ S ==>$
$(\epsilon\, c'.\ c'\text{: } super\ S\ c)\text{: } super\ S\ c$
⟨*proof*⟩

lemma *select-not-equals*: $c \in chain\ S - maxchain\ S ==>$
$(\epsilon\, c'.\ c'\text{: } super\ S\ c) \neq c$
⟨*proof*⟩

lemma *succI3*: $c \in chain\ S - maxchain\ S ==> succ\ S\ c = (\epsilon\, c'.\ c'\text{: } super\ S\ c)$
⟨*proof*⟩

lemma *succ-not-equals*: $c \in chain\ S - maxchain\ S ==> succ\ S\ c \neq c$
⟨*proof*⟩

lemma *TFin-chain-lemma4*: $c \in$ *TFin S* ==> (c :: *'a set set*): *chain S*
⟨*proof*⟩

theorem *Hausdorff*: $\exists\, c.$ (c :: *'a set set*): *maxchain S*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *chain-Union-upper*: [| $c \in$ *chain S*; $x \in c$ |] ==> $x \subseteq$ *Union*(c)
⟨*proof*⟩

lemma *chain-ball-Union-upper*: $c \in$ *chain S* ==> $\forall\, x \in c.$ $x \subseteq$ *Union*(c)
⟨*proof*⟩

lemma *maxchain-Zorn*:
[| $c \in$ *maxchain S*; $u \in S$; *Union*(c) $\subseteq u$ |] ==> *Union*(c) = u
⟨*proof*⟩

theorem *Zorn-Lemma*:
$\forall\, c \in$ *chain S*. *Union*(c): S ==> $\exists\, y \in S.$ $\forall\, z \in S.$ $y \subseteq z$ --> $y = z$
⟨*proof*⟩

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*) $\subseteq x$ --> $y = x$
⟨*proof*⟩

Various other lemmas

lemma *chainD*: [| $c \in$ *chain S*; $x \in c$; $y \in c$ |] ==> $x \subseteq y$ | $y \subseteq x$
⟨*proof*⟩

lemma *chainD2*: !!(c :: *'a set set*). $c \in$ *chain S* ==> $c \subseteq S$
⟨*proof*⟩

end

18 Product-ord: Order on product types

theory *Product-ord*
imports *Main*

begin

instance $* :: (ord,ord)\ ord$ $\langle proof \rangle$

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 $* :: (order,order)\ order$
$\langle proof \rangle$

instance $*:: (linorder,linorder)linorder$
$\langle proof \rangle$

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:
  $\mathit{int\text{-}to\text{-}nibble}\ x \equiv (\mathit{let}\ y = x\ \mathit{mod}\ 16\ \mathit{in}$
    $\mathit{if}\ y = 0\ \mathit{then}\ \mathit{Nibble0}\ \mathit{else}$
    $\mathit{if}\ y = 1\ \mathit{then}\ \mathit{Nibble1}\ \mathit{else}$
    $\mathit{if}\ y = 2\ \mathit{then}\ \mathit{Nibble2}\ \mathit{else}$
    $\mathit{if}\ y = 3\ \mathit{then}\ \mathit{Nibble3}\ \mathit{else}$
    $\mathit{if}\ y = 4\ \mathit{then}\ \mathit{Nibble4}\ \mathit{else}$
    $\mathit{if}\ y = 5\ \mathit{then}\ \mathit{Nibble5}\ \mathit{else}$
    $\mathit{if}\ y = 6\ \mathit{then}\ \mathit{Nibble6}\ \mathit{else}$
    $\mathit{if}\ y = 7\ \mathit{then}\ \mathit{Nibble7}\ \mathit{else}$
    $\mathit{if}\ y = 8\ \mathit{then}\ \mathit{Nibble8}\ \mathit{else}$
    $\mathit{if}\ y = 9\ \mathit{then}\ \mathit{Nibble9}\ \mathit{else}$
    $\mathit{if}\ y = 10\ \mathit{then}\ \mathit{NibbleA}\ \mathit{else}$
    $\mathit{if}\ y = 11\ \mathit{then}\ \mathit{NibbleB}\ \mathit{else}$
    $\mathit{if}\ y = 12\ \mathit{then}\ \mathit{NibbleC}\ \mathit{else}$
    $\mathit{if}\ y = 13\ \mathit{then}\ \mathit{NibbleD}\ \mathit{else}$
    $\mathit{if}\ y = 14\ \mathit{then}\ \mathit{NibbleE}\ \mathit{else}$
    $\mathit{NibbleF})$

lemma *int-to-nibble-nibble-to-int*: $\mathit{int\text{-}to\text{-}nibble}(\mathit{nibble\text{-}to\text{-}int}\ x) = x$
  ⟨*proof*⟩

lemma *inj-nibble-to-int*: *inj nibble-to-int*
  ⟨*proof*⟩

lemmas *nibble-to-int-eq* = *inj-nibble-to-int* [*THEN inj-eq*]

lemma *nibble-to-int-ge-0*: $0 \leq \mathit{nibble\text{-}to\text{-}int}\ x$
  ⟨*proof*⟩

lemma *nibble-to-int-less-16*: $\mathit{nibble\text{-}to\text{-}int}\ x < 16$
  ⟨*proof*⟩

Conversion between chars and int pairs.

consts
  $\mathit{char\text{-}to\text{-}int\text{-}pair} :: \mathit{char} \Rightarrow \mathit{int} \times \mathit{int}$
primrec
  $\mathit{char\text{-}to\text{-}int\text{-}pair}\ (\mathit{Char}\ a\ b) = (\mathit{nibble\text{-}to\text{-}int}\ a,\ \mathit{nibble\text{-}to\text{-}int}\ b)$

lemma *inj-char-to-int-pair*: *inj char-to-int-pair*
  ⟨*proof*⟩

lemmas *char-to-int-pair-eq* = *inj-char-to-int-pair* [*THEN inj-eq*]

Instantiation of order classes

instance *char* :: *ord* ⟨*proof*⟩

defs (overloaded)
  char-le-def: $c \leq d \equiv (\mathit{char\text{-}to\text{-}int\text{-}pair}\ c \leq \mathit{char\text{-}to\text{-}int\text{-}pair}\ d)$

char-less-def: $c < d \equiv$ (*char-to-int-pair* $c <$ *char-to-int-pair* d)

lemmas *char-ord-defs* = *char-less-def char-le-def*

instance *char* :: *order*
⟨*proof*⟩

instance *char*::*linorder*
⟨*proof*⟩

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* ˆ *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* $(\lambda(x, y).$ *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* $\Longrightarrow$ (*n*::*nat*) *div 2* < *n* ⟨*proof*⟩
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*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

mkPX preserves semantics

lemma *mkPX-ci*: *Ipol l* (*mkPX A b C*) = *Ipol l* (*PX A b C*)
⟨*proof*⟩

Correctness theorems for the implemented operations

Negation

lemma *neg-ci*: $\bigwedge$*l*. *Ipol l* (*neg P*) = −(*Ipol l P*)
⟨*proof*⟩

Addition

lemma *add-ci*: $\bigwedge$*l*. *Ipol l* (*add* (*P*, *Q*)) = *Ipol l P* + *Ipol l Q*
⟨*proof*⟩

Multiplication

lemma *mul-ci*: $\bigwedge$*l*. *Ipol l* (*mul* (*P*, *Q*)) = *Ipol l P* ∗ *Ipol l Q*
⟨*proof*⟩

Substraction

lemma *sub-ci*: $Ipol\ l\ (sub\ p\ q) = Ipol\ l\ p\ -\ Ipol\ l\ q$
⟨*proof*⟩

Square

lemma *sqr-ci*: $\bigwedge ls.\ Ipol\ ls\ (sqr\ p) = Ipol\ ls\ p * Ipol\ ls\ p$
⟨*proof*⟩

Power

lemma *even-pow*: $even\ n \Longrightarrow pow\ (p,\ n) = pow\ (sqr\ p,\ n\ div\ 2)$ ⟨*proof*⟩

lemma *pow-ci*: $\bigwedge p.\ Ipol\ ls\ (pow\ (p,\ n)) = (Ipol\ ls\ p)\ \hat{}\ n$
⟨*proof*⟩

Normalization preserves semantics

lemma *norm-ci*: $Ipolex\ l\ Pe = Ipol\ l\ (norm\ Pe)$
⟨*proof*⟩

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*⟩

Code generation

⟨*ML*⟩

end

21 List-Prefix: List prefixes and postfixes

theory *List-Prefix*
imports *Main*
begin

21.1 Prefix order on lists

instance *list* :: (*type*) *ord* ⟨*proof*⟩

defs (overloaded)
prefix-def: $xs \le ys == \exists zs.\ ys = xs\ @\ zs$
strict-prefix-def: $xs < ys == xs \le ys \wedge xs \ne (ys{::}'a\ list)$

instance *list* :: (*type*) *order*
⟨*proof*⟩

lemma *prefixI* [*intro?*]: $ys = xs\ @\ zs ==> xs \le ys$
⟨*proof*⟩

lemma *prefixE* [*elim?*]: $xs \leq ys$ ==> (!!zs. $ys = xs$ @ zs ==> C) ==> C
⟨*proof*⟩

lemma *strict-prefixI′* [*intro?*]: $ys = xs$ @ z # zs ==> $xs < ys$
⟨*proof*⟩

lemma *strict-prefixE′* [*elim?*]:
assumes *lt*: $xs < ys$
and *r*: !!z zs. $ys = xs$ @ z # zs ==> C
shows C
⟨*proof*⟩

lemma *strict-prefixI* [*intro?*]: $xs \leq ys$ ==> $xs \neq ys$ ==> $xs <$ (ys::′a *list*)
⟨*proof*⟩

lemma *strict-prefixE* [*elim?*]:
$xs < ys$ ==> ($xs \leq ys$ ==> $xs \neq$ (ys::′a *list*) ==> C) ==> C
⟨*proof*⟩

21.2 Basic properties of prefixes

theorem *Nil-prefix* [*iff*]: $[] \leq xs$
⟨*proof*⟩

theorem *prefix-Nil* [*simp*]: $(xs \leq []) = (xs = [])$
⟨*proof*⟩

lemma *prefix-snoc* [*simp*]: ($xs \leq ys$ @ $[y]$) = ($xs = ys$ @ $[y] \vee xs \leq ys$)
⟨*proof*⟩

lemma *Cons-prefix-Cons* [*simp*]: (x # $xs \leq y$ # ys) = ($x = y \wedge xs \leq ys$)
⟨*proof*⟩

lemma *same-prefix-prefix* [*simp*]: (xs @ $ys \leq xs$ @ zs) = ($ys \leq zs$)
⟨*proof*⟩

lemma *same-prefix-nil* [*iff*]: (xs @ $ys \leq xs$) = ($ys = []$)
⟨*proof*⟩

lemma *prefix-prefix* [*simp*]: $xs \leq ys$ ==> $xs \leq ys$ @ zs
⟨*proof*⟩

lemma *append-prefixD*: xs @ $ys \leq zs \Longrightarrow xs \leq zs$
⟨*proof*⟩

theorem *prefix-Cons*: ($xs \leq y$ # ys) = ($xs = [] \vee (\exists zs.\ xs = y$ # $zs \wedge zs \leq ys$))
⟨*proof*⟩

theorem *prefix-append*:
$(xs \le ys\ @\ zs) = (xs \le ys \lor (\exists\, us.\ xs = ys\ @\ us \land us \le zs))$
$\langle proof \rangle$

lemma *append-one-prefix*:
$xs \le ys ==> length\ xs < length\ ys ==> xs\ @\ [ys\ !\ length\ xs] \le ys$
$\langle proof \rangle$

theorem *prefix-length-le*: $xs \le ys ==> length\ xs \le length\ ys$
$\langle proof \rangle$

lemma *prefix-same-cases*:
$(xs_1{::}'a\ list) \le ys \Longrightarrow xs_2 \le ys \Longrightarrow xs_1 \le xs_2 \lor xs_2 \le xs_1$
$\langle proof \rangle$

lemma *set-mono-prefix*:
$xs \le ys \Longrightarrow set\ xs \subseteq set\ ys$
$\langle proof \rangle$

21.3 Parallel lists

constdefs
$parallel :: 'a\ list => 'a\ list => bool$ (**infixl** $\parallel$ *50*)
$xs \parallel ys == \neg\ xs \le ys \land \neg\ ys \le xs$

lemma *parallelI* [*intro*]: $\neg\ xs \le ys ==> \neg\ ys \le xs ==> xs \parallel ys$
$\langle proof \rangle$

lemma *parallelE* [*elim*]:
$xs \parallel ys ==> (\neg\ xs \le ys ==> \neg\ ys \le xs ==> C) ==> C$
$\langle proof \rangle$

theorem *prefix-cases*:
$(xs \le ys ==> C) ==>$
$(ys < xs ==> C) ==>$
$(xs \parallel ys ==> C) ==> C$
$\langle proof \rangle$

theorem *parallel-decomp*:
$xs \parallel ys ==> \exists\, as\ b\ bs\ c\ cs.\ b \neq c \land xs = as\ @\ b\ \#\ bs \land ys = as\ @\ c\ \#\ cs$
$\langle proof \rangle$

21.4 Postfix order on lists

constdefs
$postfix :: 'a\ list => 'a\ list => bool$ ((-/ >>= -) [*51*, *50*] *50*)
$xs >>= ys == \exists\, zs.\ xs = zs\ @\ ys$

lemma *postfix-refl* [*simp*, *intro*!]: $xs >>= xs$
$\langle proof \rangle$

lemma *postfix-trans*: $[\![xs >>= ys; ys >>= zs]\!] \Longrightarrow xs >>= zs$
⟨*proof*⟩
lemma *postfix-antisym*: $[\![xs >>= ys; ys >>= xs]\!] \Longrightarrow xs = ys$
⟨*proof*⟩

lemma *Nil-postfix* [*iff*]: $xs >>= []$
⟨*proof*⟩
lemma *postfix-Nil* [*simp*]: $([] >>= xs) = (xs = [])$
⟨*proof*⟩

lemma *postfix-ConsI*: $xs >>= ys \Longrightarrow x\#xs >>= ys$
⟨*proof*⟩
lemma *postfix-ConsD*: $xs >>= y\#ys \Longrightarrow xs >>= ys$
⟨*proof*⟩

lemma *postfix-appendI*: $xs >>= ys \Longrightarrow zs\ @\ xs >>= ys$
⟨*proof*⟩
lemma *postfix-appendD*: $xs >>= zs\ @\ ys \Longrightarrow xs >>= ys$
⟨*proof*⟩

lemma *postfix-is-subset-lemma*: $xs = zs\ @\ ys \Longrightarrow set\ ys \subseteq set\ xs$
⟨*proof*⟩
lemma *postfix-is-subset*: $xs >>= ys \Longrightarrow set\ ys \subseteq set\ xs$
⟨*proof*⟩

lemma *postfix-ConsD2-lemma* [*rule-format*]: $x\#xs = zs\ @\ y\#ys \longrightarrow xs >>= ys$
⟨*proof*⟩
lemma *postfix-ConsD2*: $x\#xs >>= y\#ys \Longrightarrow xs >>= ys$
⟨*proof*⟩

lemma *postfix2prefix*: $(xs >>= ys) = (rev\ ys <= rev\ xs)$
⟨*proof*⟩

end

22 List-lexord: Lexicographic order on lists

theory *List-lexord*
imports *Main*
begin

instance *list* :: (*ord*) *ord* ⟨*proof*⟩
defs (overloaded)
list-le-def: $(xs::('a::ord)\ list) \le ys \equiv (xs < ys \lor xs = ys)$
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*
⟨*proof*⟩

instance *list*::(*linorder*)*linorder*
⟨*proof*⟩

lemma *not-less-Nil*[*simp*]: $\sim(x < [])$
⟨*proof*⟩

lemma *Nil-less-Cons*[*simp*]: $[] < a \# x$
⟨*proof*⟩

lemma *Cons-less-Cons*[*simp*]: $(a \# x < b \# y) = (a < b \mid a = b \,\&\, x < y)$
⟨*proof*⟩

lemma *le-Nil*[*simp*]: $(x <= []) = (x = [])$
⟨*proof*⟩

lemma *Nil-le-Cons* [*simp*]: $([] <= x)$
⟨*proof*⟩

lemma *Cons-le-Cons*[*simp*]: $(a \# x <= b \# y) = (a < b \mid a = b \,\&\, x <= y)$
⟨*proof*⟩

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.