Isabelle/HOLCF-Prelude

Joachim Breitner[*], Brian Huffman, Neil Mitchell, and Christian Sternagel[†]

December 14, 2021

Abstract

The Isabelle/HOLCF-Prelude is a formalization of a large part of Haskell's standard prelude [2] in Isabelle/HOLCF. We use it to

- prove the correctness of the Eratosthenes' Sieve, in its self-referential implementation commonly used to showcase Haskell's laziness,
- prove correctness of GHC's "fold/build" rule and related rewrite rules, and
- certify a number of hints suggested by `HLint`.

The work was presented at HART 2013 [1].

Contents

[*]Supported by the Deutsche Telekom Stiftung.
[†]Supported by the Austrian Science Fund (FWF): J3202.

1 Initial Setup for HOLCF-Prelude

theory *HOLCF-Main*

imports
HOLCF
HOLCF−Library.Int-Discrete
HOL−Library.Adhoc-Overloading
begin

All theories from the Isabelle distribution which are used anywhere in the HOLCF-Prelude library must be imported via this file. This way, we only have to hide constant names and syntax in one place.

hide-type (**open**) *list*

hide-const (**open**)
List.append List.concat List.Cons List.distinct List.filter List.last
List.foldr List.foldl List.length List.lists List.map List.Nil List.nth
List.partition List.replicate List.set List.take List.upto List.zip
Orderings.less Product-Type.fst Product-Type.snd

no-notation *Map.map-add* (**infixl** ++ *100*)

no-notation *List.upto* ((*1*[-../-]))

no-notation
Rings.divide (**infixl** *div 70*) **and**
Rings.modulo (**infixl** *mod 70*)

no-notation
Set.member ((:)) **and**
Set.member ((-/ : -) [*51*, *51*] *50*)

no-translations
[*x*, *xs*] == *x* # [*xs*]
[*x*] == *x* # []

no-syntax
-*list* :: *args* ⇒ *'a List.list* ([(-)])

no-notation
List.Nil ([])

no-syntax -*bracket* :: *types* ⇒ *type* ⇒ *type* (([-]/ => -) [*0*, *0*] *0*)
no-syntax -*bracket* :: *types* ⇒ *type* ⇒ *type* (([-]/ ⇒ -) [*0*, *0*] *0*)

no-translations
[*x*<−*xs* . *P*] == *CONST List.filter* (%*x*. *P*) *xs*

no-syntax (*ASCII*)
-*filter* :: *pttrn* ⇒ *'a List.list* ⇒ *bool* ⇒ *'a List.list* ((*1*[-<−-./ -]))

no-syntax

-filter :: *pttrn* ⇒ *'a List.list* ⇒ *bool* ⇒ *'a List.list* ((*1*[-←- ./ -]))

Declarations that belong in HOLCF/Tr.thy:

declare *trE* [*cases type*: *tr*]
declare *tr-induct* [*induct type*: *tr*]

end

2 Type Classes

theory *Type-Classes*
imports *HOLCF-Main*
begin

2.1 Eq class

class *Eq* =
fixes *eq* :: *'a* → *'a* → *tr*

The Haskell type class does allow /= to be specified separately. For now, we assume that all modeled type classes use the default implementation, or an equivalent.

fixrec *neq* :: *'a*::*Eq* → *'a* → *tr* **where**
neq·*x*·*y* = *neg*·(*eq*·*x*·*y*)

class *Eq-strict* = *Eq* +
assumes *eq-strict* [*simp*]:
eq·*x*·⊥ = ⊥
eq·⊥·*y* = ⊥

class *Eq-sym* = *Eq-strict* +
assumes *eq-sym*: *eq*·*x*·*y* = *eq*·*y*·*x*

class *Eq-equiv* = *Eq-sym* +
assumes *eq-self-neq-FF* [*simp*]: *eq*·*x*·*x* ≠ *FF*
and *eq-trans*: *eq*·*x*·*y* = *TT* ⟹ *eq*·*y*·*z* = *TT* ⟹ *eq*·*x*·*z* = *TT*
begin

lemma *eq-refl*: *eq*·*x*·*x* ≠ ⊥ ⟹ *eq*·*x*·*x* = *TT*
by (*cases eq*·*x*·*x*) *simp*+

end

class *Eq-eq* = *Eq-sym* +
assumes *eq-self-neq-FF'*: *eq*·*x*·*x* ≠ *FF*
and *eq-TT-dest*: *eq*·*x*·*y* = *TT* ⟹ *x* = *y*
begin

subclass *Eq-equiv*
  by *standard* (*auto simp*: *eq-self-neq-FF′ dest*: *eq-TT-dest*)

lemma *eqD* [*dest*]:
  $eq{\cdot}x{\cdot}y = TT \Longrightarrow x = y$
  $eq{\cdot}x{\cdot}y = FF \Longrightarrow x \neq y$
  by (*auto elim*: *eq-TT-dest*)

end

2.1.1 Class instances

instantiation *lift* :: (*countable*) *Eq-eq*
begin

definition $eq \equiv (\Lambda(Def\ x)\ (Def\ y).\ Def\ (x = y))$

instance
  by *standard* (*auto simp*: *eq-lift-def flift1-def split*: *lift.splits*)

end

lemma *eq-ONE-ONE* [*simp*]: $eq{\cdot}ONE{\cdot}ONE = TT$
  unfolding *ONE-def eq-lift-def* **by** *simp*

2.2 Ord class

domain $Ordering = LT \mid EQ \mid GT$

definition *oppOrdering* :: $Ordering \to Ordering$ **where**
  $oppOrdering = (\Lambda\ x.\ case\ x\ of\ LT \Rightarrow GT \mid EQ \Rightarrow EQ \mid GT \Rightarrow LT)$

lemma *oppOrdering-simps* [*simp*]:
  $oppOrdering{\cdot}LT = GT$
  $oppOrdering{\cdot}EQ = EQ$
  $oppOrdering{\cdot}GT = LT$
  $oppOrdering{\cdot}\bot = \bot$
  unfolding *oppOrdering-def* **by** *simp-all*

class $Ord = Eq +$
  fixes *compare* :: $'a \to 'a \to Ordering$
begin

definition $lt :: 'a \to 'a \to tr$ **where**
  $lt = (\Lambda\ x\ y.\ case\ compare{\cdot}x{\cdot}y\ of\ LT \Rightarrow TT \mid EQ \Rightarrow FF \mid GT \Rightarrow FF)$

definition $le :: 'a \to 'a \to tr$ **where**
  $le = (\Lambda\ x\ y.\ case\ compare{\cdot}x{\cdot}y\ of\ LT \Rightarrow TT \mid EQ \Rightarrow TT \mid GT \Rightarrow FF)$

lemma *lt-eq-TT-iff*: $lt{\cdot}x{\cdot}y = TT \longleftrightarrow compare{\cdot}x{\cdot}y = LT$

by (*cases compare·x·y*) (*simp add*: *lt-def*)+

end

class *Ord-strict* = *Ord* +
assumes *compare-strict* [*simp*]:
compare·⊥·y = ⊥
compare·x·⊥ = ⊥
begin

lemma *lt-strict* [*simp*]:
shows *lt·⊥·x* = ⊥
and *lt·x·⊥* = ⊥
by (*simp add*: *lt-def*)+

lemma *le-strict* [*simp*]:
shows *le·⊥·x* = ⊥
and *le·x·⊥* = ⊥
by (*simp add*: *le-def*)+

end

TODO: It might make sense to have a class for preorders too, analogous to class *eq-equiv*.

class *Ord-linear* = *Ord-strict* +
assumes *eq-conv-compare*: *eq·x·y* = *is-EQ·*(*compare·x·y*)
and *oppOrdering-compare* [*simp*]:
oppOrdering·(*compare·x·y*) = *compare·y·x*
and *compare-EQ-dest*: *compare·x·y* = *EQ* ⟹ *x* = *y*
and *compare-self-below-EQ*: *compare·x·x* ⊑ *EQ*
and *compare-LT-trans*:
compare·x·y = *LT* ⟹ *compare·y·z* = *LT* ⟹ *compare·x·z* = *LT*

begin

lemma *eq-TT-dest*: *eq·x·y* = *TT* ⟹ *x* = *y*
by (*cases compare·x·y*) (*auto dest*: *compare-EQ-dest simp*: *eq-conv-compare*)+

lemma *le-iff-lt-or-eq*:
le·x·y = *TT* ⟷ *lt·x·y* = *TT* ∨ *eq·x·y* = *TT*
by (*cases compare·x·y*) (*simp add*: *le-def lt-def eq-conv-compare*)+

lemma *compare-sym*:
compare·x·y = (*case compare·y·x of LT* ⇒ *GT* | *EQ* ⇒ *EQ* | *GT* ⇒ *LT*)
by (*subst oppOrdering-compare* [*symmetric*]) (*simp add*: *oppOrdering-def*)

lemma *compare-self-neq-LT* [*simp*]: *compare·x·x* ≠ *LT*
using *compare-self-below-EQ* [*of x*] **by** *clarsimp*

lemma *compare-self-neq-GT* [*simp*]: *compare·x·x* ≠ *GT*
using *compare-self-below-EQ* [*of x*] **by** *clarsimp*

declare *compare-self-below-EQ* [*simp*]

lemma *lt-trans*: *lt·x·y* = *TT* ⟹ *lt·y·z* = *TT* ⟹ *lt·x·z* = *TT*
unfolding *lt-eq-TT-iff* **by** (*rule compare-LT-trans*)

lemma *compare-GT-iff-LT*: *compare·x·y* = *GT* ⟷ *compare·y·x* = *LT*
by (*cases compare·x·y*, *simp-all add*: *compare-sym* [*of y x*])

lemma *compare-GT-trans*:
compare·x·y = *GT* ⟹ *compare·y·z* = *GT* ⟹ *compare·x·z* = *GT*
unfolding *compare-GT-iff-LT* **by** (*rule compare-LT-trans*)

lemma *compare-EQ-iff-eq-TT*:
compare·x·y = *EQ* ⟷ *eq·x·y* = *TT*
by (*cases compare·x·y*) (*simp add*: *is-EQ-def eq-conv-compare*)+

lemma *compare-EQ-trans*:
compare·x·y = *EQ* ⟹ *compare·y·z* = *EQ* ⟹ *compare·x·z* = *EQ*
by (*blast dest*: *compare-EQ-dest*)

lemma *le-trans*:
le·x·y = *TT* ⟹ *le·y·z* = *TT* ⟹ *le·x·z* = *TT*
by (*auto dest*: *eq-TT-dest lt-trans simp*: *le-iff-lt-or-eq*)

lemma *neg-lt*: *neg·*(*lt·x·y*) = *le·y·x*
by (*cases compare·x·y*, *simp-all add*: *le-def lt-def compare-sym* [*of y x*])

lemma *neg-le*: *neg·*(*le·x·y*) = *lt·y·x*
by (*cases compare·x·y*, *simp-all add*: *le-def lt-def compare-sym* [*of y x*])

subclass *Eq-eq*
proof
fix *x y*
show *eq·x·y* = *eq·y·x*
unfolding *eq-conv-compare*
by (*cases compare·x·y*, *simp-all add*: *compare-sym* [*of y x*])
show *eq·x·*⊥ = ⊥
unfolding *eq-conv-compare* **by** *simp*
show *eq·*⊥*·y* = ⊥
unfolding *eq-conv-compare* **by** *simp*
show *eq·x·x* ≠ *FF*
unfolding *eq-conv-compare*
by (*cases compare·x·x*, *simp-all*)
{ assume *eq·x·y* = *TT* **then show** *x* = *y*
unfolding *eq-conv-compare*

by (*cases compare·x·y*, *auto dest*: *compare-EQ-dest*) **}**
qed

end

A combinator for defining Ord instances for datatypes.

definition *thenOrdering* :: *Ordering* → *Ordering* → *Ordering* **where**
thenOrdering = (Λ *x y*. *case x of LT* ⇒ *LT* | *EQ* ⇒ *y* | *GT* ⇒ *GT*)

lemma *thenOrdering-simps* [*simp*]:
thenOrdering·LT·y = *LT*
thenOrdering·EQ·y = *y*
thenOrdering·GT·y = *GT*
thenOrdering·⊥·y = ⊥
unfolding *thenOrdering-def* **by** *simp-all*

lemma *thenOrdering-LT-iff* [*simp*]:
thenOrdering·x·y = *LT* ⟷ *x* = *LT* ∨ *x* = *EQ* ∧ *y* = *LT*
by (*cases x*, *simp-all*)

lemma *thenOrdering-EQ-iff* [*simp*]:
thenOrdering·x·y = *EQ* ⟷ *x* = *EQ* ∧ *y* = *EQ*
by (*cases x*, *simp-all*)

lemma *thenOrdering-GT-iff* [*simp*]:
thenOrdering·x·y = *GT* ⟷ *x* = *GT* ∨ *x* = *EQ* ∧ *y* = *GT*
by (*cases x*, *simp-all*)

lemma *thenOrdering-below-EQ-iff* [*simp*]:
thenOrdering·x·y ⊑ *EQ* ⟷ *x* ⊑ *EQ* ∧ (*x* = ⊥ ∨ *y* ⊑ *EQ*)
by (*cases x*) *simp-all*

lemma *is-EQ-thenOrdering* [*simp*]:
is-EQ·(*thenOrdering·x·y*) = (*is-EQ·x andalso is-EQ·y*)
by (*cases x*) *simp-all*

lemma *oppOrdering-thenOrdering*:
oppOrdering·(*thenOrdering·x·y*) =
thenOrdering·(*oppOrdering·x*)·(*oppOrdering·y*)
by (*cases x*) *simp-all*

instantiation *lift* :: ({*linorder*,*countable*}) *Ord-linear*
begin

definition
compare ≡ (Λ (*Def x*) (*Def y*).
if x < *y then LT else if x* > *y then GT else EQ*)

instance proof

fix *x y z* :: *'a lift*
show *compare·⊥·y = ⊥*
unfolding *compare-lift-def* **by** *simp*
show *compare·x·⊥ = ⊥*
unfolding *compare-lift-def* **by** (*cases x*, *simp-all*)
show *oppOrdering·(compare·x·y) = compare·y·x*
unfolding *compare-lift-def*
by (*cases x*, *cases y*, *simp*, *simp*,
cases y, *simp*, *simp add*: *not-less less-imp-le*)
{ assume *compare·x·y = EQ* **then show** *x = y*
unfolding *compare-lift-def*
by (*cases x*, *cases y*, *simp*, *simp*,
cases y, *simp*, *simp split*: *if-splits*) **}**
{ assume *compare·x·y = LT* **and** *compare·y·z = LT* **then show** *compare·x·z = LT*
unfolding *compare-lift-def*
by (*cases x*, *simp*, *cases y*, *simp*, *cases z*, *simp*,
auto split: *if-splits*) **}**
show *eq·x·y = is-EQ·(compare·x·y)*
unfolding *eq-lift-def compare-lift-def*
by (*cases x*, *simp*, *cases y*, *simp*, *auto*)
show *compare·x·x ⊑ EQ*
unfolding *compare-lift-def*
by (*cases x*, *simp-all*)
qed

end

lemma *lt-le*:
lt·(x::'a::Ord-linear)·y = (le·x·y andalso neq·x·y)
by (*cases compare·x·y*)
(*auto simp*: *lt-def le-def eq-conv-compare*)

end

3 Cpo for Numerals

theory *Numeral-Cpo*
imports *HOLCF-Main*
begin

class *plus-cpo = plus + cpo +*
assumes *cont-plus1*: *cont* (*λx::'a::{plus,cpo}. x + y*)
assumes *cont-plus2*: *cont* (*λy::'a::{plus,cpo}. x + y*)
begin

abbreviation *plus-section* :: *'a → 'a → 'a* (*'(+')*) **where**
(+) ≡ Λ x y. x + y

abbreviation *plus-section-left* :: $'a \Rightarrow 'a \to 'a$ ($'(\text{-}+'$)) **where**
$(x+) \equiv \Lambda\ y.\ x + y$

abbreviation *plus-section-right* :: $'a \Rightarrow 'a \to 'a$ ($'(+\text{-}')$) **where**
$(+y) \equiv \Lambda\ x.\ x + y$

end

class *minus-cpo* = *minus* + *cpo* +
assumes *cont-minus1*: *cont* ($\lambda x::'a::\{minus,cpo\}.\ x - y$)
assumes *cont-minus2*: *cont* ($\lambda y::'a::\{minus,cpo\}.\ x - y$)
begin

abbreviation *minus-section* :: $'a \to 'a \to 'a$ ($'(-')$) **where**
$(-) \equiv \Lambda\ x\ y.\ x - y$

abbreviation *minus-section-left* :: $'a \Rightarrow 'a \to 'a$ ($'(--')$) **where**
$(x-) \equiv \Lambda\ y.\ x - y$

abbreviation *minus-section-right* :: $'a \Rightarrow 'a \to 'a$ ($'(-\text{-}')$) **where**
$(-y) \equiv \Lambda\ x.\ x - y$

end

class *times-cpo* = *times* + *cpo* +
assumes *cont-times1*: *cont* ($\lambda x::'a::\{times,cpo\}.\ x * y$)
assumes *cont-times2*: *cont* ($\lambda y::'a::\{times,cpo\}.\ x * y$)
begin

end

lemma *cont2cont-plus* [*simp*, *cont2cont*]:
assumes *cont* ($\lambda x.\ f\ x$) **and** *cont* ($\lambda x.\ g\ x$)
shows *cont* ($\lambda x.\ f\ x + g\ x :: 'a::plus\text{-}cpo$)
apply (*rule cont-apply* [*OF assms*(*1*) *cont-plus1*])
apply (*rule cont-apply* [*OF assms*(*2*) *cont-plus2*])
apply (*rule cont-const*)
done

lemma *cont2cont-minus* [*simp*, *cont2cont*]:
assumes *cont* ($\lambda x.\ f\ x$) **and** *cont* ($\lambda x.\ g\ x$)
shows *cont* ($\lambda x.\ f\ x - g\ x :: 'a::minus\text{-}cpo$)
apply (*rule cont-apply* [*OF assms*(*1*) *cont-minus1*])
apply (*rule cont-apply* [*OF assms*(*2*) *cont-minus2*])
apply (*rule cont-const*)
done

lemma *cont2cont-times* [*simp*, *cont2cont*]:
assumes *cont* (λ*x*. *f x*) **and** *cont* (λ*x*. *g x*)
shows *cont* (λ*x*. *f x* ∗ *g x* :: *'a*::*times-cpo*)
apply (*rule cont-apply* [*OF assms*(*1*) *cont-times1*])
apply (*rule cont-apply* [*OF assms*(*2*) *cont-times2*])
apply (*rule cont-const*)
done

instantiation *u* :: ({*zero*,*cpo*}) *zero*
begin
definition *zero-u* = *up*·(*0*::*'a*)
instance ..
end

instantiation *u* :: ({*one*,*cpo*}) *one*
begin
definition *one-u* = *up*·(*1*::*'a*)
instance ..
end

instantiation *u* :: (*plus-cpo*) *plus*
begin
definition *plus-u x y* = (Λ(*up*·*a*) (*up*·*b*). *up*·(*a* + *b*))·*x*·*y* **for** *x y* :: *'a*$_\perp$
instance ..
end

instantiation *u* :: (*minus-cpo*) *minus*
begin
definition *minus-u x y* = (Λ(*up*·*a*) (*up*·*b*). *up*·(*a* − *b*))·*x*·*y* **for** *x y* :: *'a*$_\perp$
instance ..
end

instantiation *u* :: (*times-cpo*) *times*
begin
definition *times-u x y* = (Λ(*up*·*a*) (*up*·*b*). *up*·(*a* ∗ *b*))·*x*·*y* **for** *x y* :: *'a*$_\perp$
instance ..
end

lemma *plus-u-strict* [*simp*]:
fixes *x* :: - *u* **shows** *x* + ⊥ = ⊥ **and** ⊥ + *x* = ⊥
unfolding *plus-u-def* **by** (*cases x*, *simp*, *simp*)+

lemma *minus-u-strict* [*simp*]:
fixes *x* :: - *u* **shows** *x* − ⊥ = ⊥ **and** ⊥ − *x* = ⊥
unfolding *minus-u-def* **by** (*cases x*, *simp-all*)+

lemma *times-u-strict* [*simp*]:
fixes *x* :: - *u* **shows** *x* ∗ ⊥ = ⊥ **and** ⊥ ∗ *x* = ⊥
unfolding *times-u-def* **by** (*cases x*, *simp-all*)+

lemma *plus-up-up* [*simp*]: *up*·*x* + *up*·*y* = *up*·(*x* + *y*)
unfolding *plus-u-def* **by** *simp*

lemma *minus-up-up* [*simp*]: *up*·*x* − *up*·*y* = *up*·(*x* − *y*)
unfolding *minus-u-def* **by** *simp*

lemma *times-up-up* [*simp*]: *up*·*x* ∗ *up*·*y* = *up*·(*x* ∗ *y*)
unfolding *times-u-def* **by** *simp*

instance *u* :: (*plus-cpo*) *plus-cpo*
by *standard* (*simp-all add*: *plus-u-def*)

instance *u* :: (*minus-cpo*) *minus-cpo*
by *standard* (*simp-all add*: *minus-u-def*)

instance *u* :: (*times-cpo*) *times-cpo*
by *standard* (*simp-all add*: *times-u-def*)

instance *u* :: ({*semigroup-add*,*plus-cpo*}) *semigroup-add*
proof
fix *a b c* :: *'a u*
show (*a* + *b*) + *c* = *a* + (*b* + *c*)
unfolding *plus-u-def*
by (*cases a*; *cases b*; *cases c*) (*simp-all add*: *add.assoc*)
qed

instance *u* :: ({*ab-semigroup-add*,*plus-cpo*}) *ab-semigroup-add*
proof
fix *a b* :: *'a u*
show *a* + *b* = *b* + *a*
by (*cases a*; *cases b*) (*simp-all add*: *add.commute*)
qed

instance *u* :: ({*monoid-add*,*plus-cpo*}) *monoid-add*
proof
fix *a* :: *'a u*
show *0* + *a* = *a*
unfolding *zero-u-def* **by** (*cases a*) *simp-all*
show *a* + *0* = *a*
unfolding *zero-u-def* **by** (*cases a*) *simp-all*
qed

instance *u* :: ({*comm-monoid-add*,*plus-cpo*}) *comm-monoid-add*
proof
fix *a* :: *'a u*
show *0* + *a* = *a*
unfolding *zero-u-def* **by** (*cases a*) *simp-all*
qed

instance $u :: (\{numeral, plus\text{-}cpo\})\ numeral$ **..**

instance *int* :: *plus-cpo*
by *standard simp-all*

instance *int* :: *minus-cpo*
by *standard simp-all*

end

4 Data: Functions

theory *Data-Function*
imports *HOLCF-Main*
begin

fixrec *flip* :: $('a \mathrel{->} 'b \mathrel{->} 'c) \mathrel{->} 'b \mathrel{->} 'a \mathrel{->} 'c$ **where**
$flip{\cdot}f{\cdot}x{\cdot}y = f{\cdot}y{\cdot}x$

fixrec *const* :: $'a \to 'b \to 'a$ **where**
$const{\cdot}x{\cdot}\text{-} = x$

fixrec *dollar* :: $('a \mathrel{->} 'b) \mathrel{->} 'a \mathrel{->} 'b$ **where**
$dollar{\cdot}f{\cdot}x = f{\cdot}x$

fixrec *dollarBang* :: $('a \mathrel{->} 'b) \mathrel{->} 'a \mathrel{->} 'b$ **where**
$dollarBang{\cdot}f{\cdot}x = seq{\cdot}x{\cdot}(f{\cdot}x)$

fixrec *on* :: $('b \mathrel{->} 'b \mathrel{->} 'c) \mathrel{->} ('a \mathrel{->} 'b) \mathrel{->} 'a \mathrel{->} 'a \mathrel{->} 'c$ **where**
$on{\cdot}g{\cdot}f{\cdot}x{\cdot}y = g{\cdot}(f{\cdot}x){\cdot}(f{\cdot}y)$

end

5 Data: Bool

theory *Data-Bool*
imports *Type-Classes*
begin

5.1 Class instances

Eq

lemma *eq-eqI*[*case-names bottomLTR bottomRTL LTR RTL*]:
$(x = \bot \Longrightarrow y = \bot) \Longrightarrow (y = \bot \Longrightarrow x = \bot) \Longrightarrow (x = TT \Longrightarrow y = TT) \Longrightarrow (y = TT \Longrightarrow x = TT) \Longrightarrow x = y$
by (*metis Exh-tr*)

lemma *eq-tr-simps* [*simp*]:
shows *eq·TT·TT = TT* **and** *eq·TT·FF = FF*
and *eq·FF·TT = FF* **and** *eq·FF·FF = TT*
unfolding *TT-def FF-def eq-lift-def* **by** *simp-all*

Ord

lemma *compare-tr-simps* [*simp*]:
compare·FF·FF = EQ
compare·FF·TT = LT
compare·TT·FF = GT
compare·TT·TT = EQ
unfolding *TT-def FF-def compare-lift-def*
by *simp-all*

5.2 Lemmas

lemma *andalso-eq-TT-iff* [*simp*]:
(*x andalso y*) = *TT* ⟷ *x* = *TT* ∧ *y* = *TT*
by (*cases x*, *simp-all*)

lemma *andalso-eq-FF-iff* [*simp*]:
(*x andalso y*) = *FF* ⟷ *x* = *FF* ∨ (*x* = *TT* ∧ *y* = *FF*)
by (*cases x*, *simp-all*)

lemma *andalso-eq-bottom-iff* [*simp*]:
(*x andalso y*) = ⊥ ⟷ *x* = ⊥ ∨ (*x* = *TT* ∧ *y* = ⊥)
by (*cases x*, *simp-all*)

lemma *orelse-eq-FF-iff* [*simp*]:
(*x orelse y*) = *FF* ⟷ *x* = *FF* ∧ *y* = *FF*
by (*cases x*, *simp-all*)

lemma *orelse-assoc* [*simp*]:
((*x orelse y*) *orelse z*) = (*x orelse y orelse z*)
by (*cases x*) *auto*

lemma *andalso-assoc* [*simp*]:
((*x andalso y*) *andalso z*) = (*x andalso y andalso z*)
by (*cases x*) *auto*

lemma *neg-orelse* [*simp*]:
neg·(*x orelse y*) = (*neg·x andalso neg·y*)
by (*cases x*) *auto*

lemma *neg-andalso* [*simp*]:
neg·(*x andalso y*) = (*neg·x orelse neg·y*)
by (*cases x*) *auto*

Not suitable as default simp rules, because they cause the simplifier to loop:

lemma *neg-eq-simps*:
  $neg{\cdot}x = TT \Longrightarrow x = FF$
  $neg{\cdot}x = FF \Longrightarrow x = TT$
  $neg{\cdot}x = \bot \Longrightarrow x = \bot$
  by (*cases x*, *simp-all*)+

lemma *neg-eq-TT-iff* [*simp*]: $neg{\cdot}x = TT \longleftrightarrow x = FF$
  by (*cases x*, *simp-all*)+

lemma *neg-eq-FF-iff* [*simp*]: $neg{\cdot}x = FF \longleftrightarrow x = TT$
  by (*cases x*, *simp-all*)+

lemma *neg-eq-bottom-iff* [*simp*]: $neg{\cdot}x = \bot \longleftrightarrow x = \bot$
  by (*cases x*, *simp-all*)+

lemma *neg-eq* [*simp*]:
  $neg{\cdot}x = neg{\cdot}y \longleftrightarrow x = y$
  by (*cases y*, *simp-all*)

lemma *neg-neg* [*simp*]:
  $neg{\cdot}(neg{\cdot}x) = x$
  by (*cases x*, *auto*)

lemma *neg-comp-neg* [*simp*]:
  neg oo neg = *ID*
  by (*metis cfun-eqI cfcomp2 neg-neg ID1*)

end

6 Data: Tuple

theory *Data-Tuple*
  imports
    Type-Classes
    Data-Bool
begin

6.1 Datatype definitions

domain *Unit* ($\langle\rangle$) = *Unit* ($\langle\rangle$)

domain ('*a*, '*b*) *Tuple2* ($\langle$-, -$\rangle$) =
  Tuple2 (**lazy** *fst* :: '*a*) (**lazy** *snd* :: '*b*) ($\langle$-, -$\rangle$)

notation *Tuple2* ($\langle$,$\rangle$)

fixrec *uncurry* :: ('*a* → '*b* → '*c*) → $\langle$'*a*, '*b*$\rangle$ → '*c*

where *uncurry*·*f*·*p* = *f*·(*fst*·*p*)·(*snd*·*p*)

fixrec *curry* :: (⟨'*a*, '*b*⟩ → '*c*) → '*a* → '*b* → '*c*
where *curry*·*f*·*a*·*b* = *f*·⟨*a*, *b*⟩

domain ('*a*, '*b*, '*c*) *Tuple3* (⟨-, -, -⟩) =
Tuple3 (**lazy** '*a*) (**lazy** '*b*) (**lazy** '*c*) (⟨-, -, -⟩)

notation *Tuple3* (⟨,,⟩)

6.2 Type class instances

instantiation *Unit* :: *Ord-linear*
begin

definition
eq = (Λ ⟨⟩ ⟨⟩. *TT*)

definition
compare = (Λ ⟨⟩ ⟨⟩. *EQ*)

instance
apply *standard*
apply (*unfold eq-Unit-def compare-Unit-def*)
apply *simp*
apply (*rename-tac x*, *case-tac x*, *simp*, *simp*)
apply (*rename-tac x y*, *case-tac x*, *simp*, *case-tac y*, *simp*, *simp*)
apply (*rename-tac x y*, *case-tac x*, *case-tac y*, *simp*, *simp*, *case-tac y*, *simp*, *simp*)
apply (*rename-tac x y*, *case-tac x*, *simp*, *case-tac y*, *simp*, *simp*)
apply (*rename-tac x*, *case-tac x*, *simp*, *simp*)
apply (*rename-tac x y z*, *case-tac x*, *simp*, *case-tac y*, *simp*, *case-tac z*, *simp*, *simp*)
done

end

instantiation *Tuple2* :: (*Eq*, *Eq*) *Eq-strict*
begin

definition
eq = (Λ ⟨*x1*, *y1*⟩ ⟨*x2*, *y2*⟩. *eq*·*x1*·*x2 andalso eq*·*y1*·*y2*)

instance proof
fix *x* :: ⟨'*a*, '*b*⟩
show *eq*·*x*·⊥ = ⊥
unfolding *eq-Tuple2-def* **by** (*cases x*, *simp-all*)
show *eq*·⊥·*x* = ⊥
unfolding *eq-Tuple2-def* **by** *simp*

qed

end

lemma *eq-Tuple2-simps* [*simp*]:
eq·⟨*x1*, *y1*⟩·⟨*x2*, *y2*⟩ = (*eq*·*x1*·*x2 andalso eq*·*y1*·*y2*)
unfolding *eq-Tuple2-def* **by** *simp*

instance *Tuple2* :: (*Eq-sym*, *Eq-sym*) *Eq-sym*
proof
fix *x y* :: ⟨'*a*, '*b*⟩
show *eq*·*x*·*y* = *eq*·*y*·*x*
unfolding *eq-Tuple2-def*
by (*cases x*, *simp*, (*cases y*, *simp*, *simp add*: *eq-sym*)+)
qed

instance *Tuple2* :: (*Eq-equiv*, *Eq-equiv*) *Eq-equiv*
proof
fix *x y z* :: ⟨'*a*, '*b*⟩
show *eq*·*x*·*x* ≠ *FF*
by (*cases x*, *simp-all*)
{ assume *eq*·*x*·*y* = *TT* **and** *eq*·*y*·*z* = *TT* **then show** *eq*·*x*·*z* = *TT*
by (*cases x*, *simp*, *cases y*, *simp*, *cases z*, *simp*, *simp*,
fast intro: *eq-trans*) **}**
qed

instance *Tuple2* :: (*Eq-eq*, *Eq-eq*) *Eq-eq*
proof
fix *x y* :: ⟨'*a*, '*b*⟩
show *eq*·*x*·*x* ≠ *FF*
by (*cases x*, *simp-all*)
{ assume *eq*·*x*·*y* = *TT* **then show** *x* = *y*
by (*cases x*, *simp*, *cases y*, *simp*, *simp*, *fast*) **}**
qed

instantiation *Tuple2* :: (*Ord*, *Ord*) *Ord-strict*
begin

definition
compare = (Λ ⟨*x1*, *y1*⟩ ⟨*x2*, *y2*⟩.
thenOrdering·(*compare*·*x1*·*x2*)·(*compare*·*y1*·*y2*))

instance
by *standard* (*simp add*: *compare-Tuple2-def*,
rename-tac x, *case-tac x*, *simp-all add*: *compare-Tuple2-def*)

end

lemma *compare-Tuple2-simps* [*simp*]:

```
  compare·⟨x1, y1⟩·⟨x2, y2⟩ = thenOrdering·(compare·x1·x2)·(compare·y1·y2)
  unfolding compare-Tuple2-def by simp

instance Tuple2 :: (Ord-linear, Ord-linear) Ord-linear
proof
  fix x y z :: ⟨'a, 'b⟩
  show eq·x·y = is-EQ·(compare·x·y)
    by (cases x, simp, cases y, simp, simp add: eq-conv-compare)
  show oppOrdering·(compare·x·y) = compare·y·x
    by (cases x, simp, cases y, simp, simp add: oppOrdering-thenOrdering)
  { assume compare·x·y = EQ then show x = y
      by (cases x, simp, cases y, simp, auto elim: compare-EQ-dest) }
  { assume compare·x·y = LT and compare·y·z = LT then show compare·x·z =
LT
      apply (cases x, simp, cases y, simp, cases z, simp, simp)
      apply (elim disjE conjE)
        apply (fast elim!: compare-LT-trans)
       apply (fast dest: compare-EQ-dest)
      apply (fast dest: compare-EQ-dest)
      apply (drule compare-EQ-dest)
      apply (fast elim!: compare-LT-trans)
      done }
  show compare·x·x ⊑ EQ
    by (cases x, simp-all)
qed

instantiation Tuple3 :: (Eq, Eq, Eq) Eq-strict
begin

definition
  eq = (Λ ⟨x1, y1, z1⟩ ⟨x2, y2, z2⟩.
    eq·x1·x2 andalso eq·y1·y2 andalso eq·z1·z2)

instance proof
  fix x :: ⟨'a, 'b, 'c⟩
  show eq·x·⊥ = ⊥
    unfolding eq-Tuple3-def by (cases x, simp-all)
  show eq·⊥·x = ⊥
    unfolding eq-Tuple3-def by simp
qed

end

lemma eq-Tuple3-simps [simp]:
  eq·⟨x1, y1, z1⟩·⟨x2, y2, z2⟩ = (eq·x1·x2 andalso eq·y1·y2 andalso eq·z1·z2)
  unfolding eq-Tuple3-def by simp

instance Tuple3 :: (Eq-sym, Eq-sym, Eq-sym) Eq-sym
proof
```

```
  fix x y :: ⟨'a, 'b, 'c⟩
  show eq·x·y = eq·y·x
    unfolding eq-Tuple3-def
    by (cases x, simp, (cases y, simp, simp add: eq-sym)+)
qed

instance Tuple3 :: (Eq-equiv, Eq-equiv, Eq-equiv) Eq-equiv
proof
  fix x y z :: ⟨'a, 'b, 'c⟩
  show eq·x·x ≠ FF
    by (cases x, simp-all)
  { assume eq·x·y = TT and eq·y·z = TT then show eq·x·z = TT
      by (cases x, simp, cases y, simp, cases z, simp, simp,
          fast intro: eq-trans) }
qed

instance Tuple3 :: (Eq-eq, Eq-eq, Eq-eq) Eq-eq
proof
  fix x y :: ⟨'a, 'b, 'c⟩
  show eq·x·x ≠ FF
    by (cases x, simp-all)
  { assume eq·x·y = TT then show x = y
      by (cases x, simp, cases y, simp, simp, fast) }
qed

instantiation Tuple3 :: (Ord, Ord, Ord) Ord-strict
begin

definition
  compare = (Λ ⟨x1, y1, z1⟩ ⟨x2, y2, z2⟩.
   thenOrdering·(compare·x1·x2)·(thenOrdering·(compare·y1·y2)·(compare·z1·z2)))

instance
  by standard (simp add: compare-Tuple3-def,
    rename-tac x, case-tac x, simp-all add: compare-Tuple3-def)

end

lemma compare-Tuple3-simps [simp]:
  compare·⟨x1, y1, z1⟩·⟨x2, y2, z2⟩ =
    thenOrdering·(compare·x1·x2)·
      (thenOrdering·(compare·y1·y2)·(compare·z1·z2))
  unfolding compare-Tuple3-def by simp

instance Tuple3 :: (Ord-linear, Ord-linear, Ord-linear) Ord-linear
proof
  fix x y z :: ⟨'a, 'b, 'c⟩
  show eq·x·y = is-EQ·(compare·x·y)
    by (cases x, simp, cases y, simp, simp add: eq-conv-compare)
```

show *oppOrdering·(compare·x·y) = compare·y·x*
by (*cases x*, *simp*, *cases y*, *simp*, *simp add*: *oppOrdering-thenOrdering*)
{ assume *compare·x·y = EQ* **then show** *x = y*
by (*cases x*, *simp*, *cases y*, *simp*, *auto elim*: *compare-EQ-dest*) **}**
{ assume *compare·x·y = LT* **and** *compare·y·z = LT* **then show** *compare·x·z = LT*
apply (*cases x*, *simp*, *cases y*, *simp*, *cases z*, *simp*, *simp*)
apply (*elim disjE conjE*)
apply (*fast elim*!: *compare-LT-trans*)
apply (*fast dest*: *compare-EQ-dest*)
apply (*fast dest*: *compare-EQ-dest*)
apply (*fast dest*: *compare-EQ-dest*)
apply (*fast dest*: *compare-EQ-dest*)
apply (*drule compare-EQ-dest*)
apply (*fast elim*!: *compare-LT-trans*)
apply (*fast dest*: *compare-EQ-dest*)
apply (*fast dest*: *compare-EQ-dest*)
apply (*fast dest*: *compare-EQ-dest elim*!: *compare-LT-trans*)
done }
show *compare·x·x ⊑ EQ*
by (*cases x*, *simp-all*)
qed

end

7 Data: Integers

theory *Data-Integer*
imports
Numeral-Cpo
Data-Bool
begin

domain *Integer = MkI* (**lazy** *int*)

instance *Integer* :: *flat*
proof
fix *x y* :: *Integer*
assume *x ⊑ y* **then show** *x = ⊥ ∨ x = y*
by (*cases x*; *cases y*) *simp-all*
qed

instantiation *Integer* :: {*plus*,*times*,*minus*,*uminus*,*zero*,*one*}
begin

definition *0 = MkI·0*
definition *1 = MkI·1*
definition *a + b = (Λ (MkI·x) (MkI·y). MkI·(x + y))·a·b*
definition *a − b = (Λ (MkI·x) (MkI·y). MkI·(x − y))·a·b*

definition $a * b = (\Lambda\ (MkI{\cdot}x)\ (MkI{\cdot}y).\ MkI{\cdot}(x * y)){\cdot}a{\cdot}b$
definition $-\ a = (\Lambda\ (MkI{\cdot}x).\ MkI{\cdot}(uminus\ x)){\cdot}a$

instance ..

end

lemma *Integer-arith-strict* [*simp*]:
fixes x :: *Integer*
shows $\bot + x = \bot$ **and** $x + \bot = \bot$
and $\bot * x = \bot$ **and** $x * \bot = \bot$
and $\bot - x = \bot$ **and** $x - \bot = \bot$
and $-\bot = (\bot{::}Integer)$
unfolding *plus-Integer-def times-Integer-def*
unfolding *minus-Integer-def uminus-Integer-def*
by (*cases x*, *simp*, *simp*)+

lemma *Integer-arith-simps* [*simp*]:
$MkI{\cdot}a + MkI{\cdot}b = MkI{\cdot}(a + b)$
$MkI{\cdot}a * MkI{\cdot}b = MkI{\cdot}(a * b)$
$MkI{\cdot}a - MkI{\cdot}b = MkI{\cdot}(a - b)$
$-\ MkI{\cdot}a = MkI{\cdot}(uminus\ a)$
unfolding *plus-Integer-def times-Integer-def*
unfolding *minus-Integer-def uminus-Integer-def*
by *simp-all*

lemma *plus-MkI-MkI*:
$MkI{\cdot}x + MkI{\cdot}y = MkI{\cdot}(x + y)$
unfolding *plus-Integer-def* **by** *simp*

instance *Integer* :: {*plus-cpo*,*minus-cpo*,*times-cpo*}
by *standard* (*simp-all add*: *flatdom-strict2cont*)

instance *Integer* :: *comm-monoid-add*
proof
fix $a\ b\ c$ :: *Integer*
show $(a + b) + c = a + (b + c)$
by (*cases a*; *cases b*; *cases c*) *simp-all*
show $a + b = b + a$
by (*cases a*; *cases b*) *simp-all*
show $0 + a = a$
unfolding *zero-Integer-def*
by (*cases a*) *simp-all*
qed

instance *Integer* :: *comm-monoid-mult*
proof
fix $a\ b\ c$ :: *Integer*
show $(a * b) * c = a * (b * c)$

by (*cases a*; *cases b*; *cases c*) *simp-all*
show $a * b = b * a$
by (*cases a*; *cases b*) *simp-all*
show $1 * a = a$
unfolding *one-Integer-def*
by (*cases a*) *simp-all*
qed

instance *Integer* :: *comm-semiring*
proof
fix *a b c* :: *Integer*
show $(a + b) * c = a * c + b * c$
by (*cases a*; *cases b*; *cases c*) (*simp-all add*: *distrib-right*)
qed

instance *Integer* :: *semiring-numeral* **..**

lemma *Integer-add-diff-cancel* [*simp*]:
$b \neq \bot \Longrightarrow (a::Integer) + b - b = a$
by (*cases a*; *cases b*) *simp-all*

lemma *zero-Integer-neq-bottom* [*simp*]: $(0::Integer) \neq \bot$
by (*simp add*: *zero-Integer-def*)

lemma *one-Integer-neq-bottom* [*simp*]: $(1::Integer) \neq \bot$
by (*simp add*: *one-Integer-def*)

lemma *plus-Integer-eq-bottom-iff* [*simp*]:
fixes *x y* :: *Integer* **shows** $x + y = \bot \longleftrightarrow x = \bot \vee y = \bot$
by (*cases x*, *simp*, *cases y*, *simp*, *simp*)

lemma *diff-Integer-eq-bottom-iff* [*simp*]:
fixes *x y* :: *Integer* **shows** $x - y = \bot \longleftrightarrow x = \bot \vee y = \bot$
by (*cases x*, *simp*, *cases y*, *simp*, *simp*)

lemma *mult-Integer-eq-bottom-iff* [*simp*]:
fixes *x y* :: *Integer* **shows** $x * y = \bot \longleftrightarrow x = \bot \vee y = \bot$
by (*cases x*, *simp*, *cases y*, *simp*, *simp*)

lemma *minus-Integer-eq-bottom-iff* [*simp*]:
fixes *x* :: *Integer* **shows** $- x = \bot \longleftrightarrow x = \bot$
by (*cases x*, *simp*, *simp*)

lemma *numeral-Integer-eq*: *numeral k* = *MkI*·(*numeral k*)
by (*induct k*, *simp-all only*: *numeral.simps one-Integer-def plus-MkI-MkI*)

lemma *numeral-Integer-neq-bottom* [*simp*]: (*numeral k*::*Integer*) $\neq \bot$
unfolding *numeral-Integer-eq* **by** *simp*

Symmetric versions are also needed, because the reorient simproc does not

apply to these comparisons.

lemma *bottom-neq-zero-Integer* [*simp*]: (⊥::*Integer*) ≠ *0*
by (*simp add*: *zero-Integer-def*)

lemma *bottom-neq-one-Integer* [*simp*]: (⊥::*Integer*) ≠ *1*
by (*simp add*: *one-Integer-def*)

lemma *bottom-neq-numeral-Integer* [*simp*]: (⊥::*Integer*) ≠ *numeral k*
unfolding *numeral-Integer-eq* **by** *simp*

instantiation *Integer* :: *Ord-linear*
begin

definition
eq = (Λ (*MkI*·*x*) (*MkI*·*y*). *if x* = *y then TT else FF*)

definition
compare = (Λ (*MkI*·*x*) (*MkI*·*y*).
if x < *y then LT else if x* > *y then GT else EQ*)

instance proof
fix *x y z* :: *Integer*
show *compare*·⊥·*y* = ⊥
unfolding *compare-Integer-def* **by** *simp*
show *compare*·*x*·⊥ = ⊥
unfolding *compare-Integer-def* **by** (*cases x*, *simp-all*)
show *oppOrdering*·(*compare*·*x*·*y*) = *compare*·*y*·*x*
unfolding *compare-Integer-def*
by (*cases x*, *cases y*, *simp*, *simp*,
cases y, *simp*, *simp add*: *not-less less-imp-le*)
{ assume *compare*·*x*·*y* = *EQ* **then show** *x* = *y*
unfolding *compare-Integer-def*
by (*cases x*, *cases y*, *simp*, *simp*,
cases y, *simp*, *simp split*: *if-splits*) **}**
{ assume *compare*·*x*·*y* = *LT* **and** *compare*·*y*·*z* = *LT* **then show** *compare*·*x*·*z* = *LT*
unfolding *compare-Integer-def*
by (*cases x*, *simp*, *cases y*, *simp*, *cases z*, *simp*,
auto split: *if-splits*) **}**
show *eq*·*x*·*y* = *is-EQ*·(*compare*·*x*·*y*)
unfolding *eq-Integer-def compare-Integer-def*
by (*cases x*, *simp*, *cases y*, *simp*, *auto*)
show *compare*·*x*·*x* ⊑ *EQ*
unfolding *compare-Integer-def*
by (*cases x*, *simp-all*)
qed

end

lemma *eq-MkI-MkI* [*simp*]:
eq·(*MkI*·*m*)·(*MkI*·*n*) = (*if m* = *n then TT else FF*)
by (*simp add*: *eq-Integer-def*)

lemma *compare-MkI-MkI* [*simp*]:
compare·(*MkI*·*x*)·(*MkI*·*y*) = (*if x* < *y then LT else if x* > *y then GT else EQ*)
unfolding *compare-Integer-def* **by** *simp*

lemma *lt-MkI-MkI* [*simp*]:
lt·(*MkI*·*x*)·(*MkI*·*y*) = (*if x* < *y then TT else FF*)
unfolding *lt-def* **by** *simp*

lemma *le-MkI-MkI* [*simp*]:
le·(*MkI*·*x*)·(*MkI*·*y*) = (*if x* ≤ *y then TT else FF*)
unfolding *le-def* **by** *simp*

lemma *eq-Integer-bottom-iff* [*simp*]:
fixes *x y* :: *Integer* **shows** *eq*·*x*·*y* = ⊥ ⟷ *x* = ⊥ ∨ *y* = ⊥
by (*cases x*, *simp*, *cases y*, *simp*, *simp*)

lemma *compare-Integer-bottom-iff* [*simp*]:
fixes *x y* :: *Integer* **shows** *compare*·*x*·*y* = ⊥ ⟷ *x* = ⊥ ∨ *y* = ⊥
by (*cases x*, *simp*, *cases y*, *simp*, *simp*)

lemma *lt-Integer-bottom-iff* [*simp*]:
fixes *x y* :: *Integer* **shows** *lt*·*x*·*y* = ⊥ ⟷ *x* = ⊥ ∨ *y* = ⊥
by (*cases x*, *simp*, *cases y*, *simp*, *simp*)

lemma *le-Integer-bottom-iff* [*simp*]:
fixes *x y* :: *Integer* **shows** *le*·*x*·*y* = ⊥ ⟷ *x* = ⊥ ∨ *y* = ⊥
by (*cases x*, *simp*, *cases y*, *simp*, *simp*)

lemma *compare-refl-Integer* [*simp*]:
(*x*::*Integer*) ≠ ⊥ ⟹ *compare*·*x*·*x* = *EQ*
by (*cases x*) *simp-all*

lemma *eq-refl-Integer* [*simp*]:
(*x*::*Integer*) ≠ ⊥ ⟹ *eq*·*x*·*x* = *TT*
by (*cases x*) *simp-all*

lemma *lt-refl-Integer* [*simp*]:
(*x*::*Integer*) ≠ ⊥ ⟹ *lt*·*x*·*x* = *FF*
by (*cases x*) *simp-all*

lemma *le-refl-Integer* [*simp*]:
(*x*::*Integer*) ≠ ⊥ ⟹ *le*·*x*·*x* = *TT*
by (*cases x*) *simp-all*

lemma *eq-Integer-numeral-simps* [*simp*]:

$eq{\cdot}(0{::}Integer){\cdot}0 = TT$
$eq{\cdot}(0{::}Integer){\cdot}1 = FF$
$eq{\cdot}(1{::}Integer){\cdot}0 = FF$
$eq{\cdot}(1{::}Integer){\cdot}1 = TT$
$eq{\cdot}(0{::}Integer){\cdot}(numeral\ k) = FF$
$eq{\cdot}(numeral\ k){\cdot}(0{::}Integer) = FF$
$k \neq Num.One \Longrightarrow eq{\cdot}(1{::}Integer){\cdot}(numeral\ k) = FF$
$k \neq Num.One \Longrightarrow eq{\cdot}(numeral\ k){\cdot}(1{::}Integer) = FF$
$eq{\cdot}(numeral\ k{::}Integer){\cdot}(numeral\ l) = (if\ k = l\ then\ TT\ else\ FF)$
unfolding *zero-Integer-def one-Integer-def numeral-Integer-eq*
by *simp-all*

lemma *compare-Integer-numeral-simps* [*simp*]:
$compare{\cdot}(0{::}Integer){\cdot}0 = EQ$
$compare{\cdot}(0{::}Integer){\cdot}1 = LT$
$compare{\cdot}(1{::}Integer){\cdot}0 = GT$
$compare{\cdot}(1{::}Integer){\cdot}1 = EQ$
$compare{\cdot}(0{::}Integer){\cdot}(numeral\ k) = LT$
$compare{\cdot}(numeral\ k){\cdot}(0{::}Integer) = GT$
$Num.One < k \Longrightarrow compare{\cdot}(1{::}Integer){\cdot}(numeral\ k) = LT$
$Num.One < k \Longrightarrow compare{\cdot}(numeral\ k){\cdot}(1{::}Integer) = GT$
$compare{\cdot}(numeral\ k{::}Integer){\cdot}(numeral\ l) =$
$(if\ k < l\ then\ LT\ else\ if\ k > l\ then\ GT\ else\ EQ)$
unfolding *zero-Integer-def one-Integer-def numeral-Integer-eq*
by *simp-all*

lemma *lt-Integer-numeral-simps* [*simp*]:
$lt{\cdot}(0{::}Integer){\cdot}0 = FF$
$lt{\cdot}(0{::}Integer){\cdot}1 = TT$
$lt{\cdot}(1{::}Integer){\cdot}0 = FF$
$lt{\cdot}(1{::}Integer){\cdot}1 = FF$
$lt{\cdot}(0{::}Integer){\cdot}(numeral\ k) = TT$
$lt{\cdot}(numeral\ k){\cdot}(0{::}Integer) = FF$
$Num.One < k \Longrightarrow lt{\cdot}(1{::}Integer){\cdot}(numeral\ k) = TT$
$lt{\cdot}(numeral\ k){\cdot}(1{::}Integer) = FF$
$lt{\cdot}(numeral\ k{::}Integer){\cdot}(numeral\ l) = (if\ k < l\ then\ TT\ else\ FF)$
unfolding *zero-Integer-def one-Integer-def numeral-Integer-eq*
by *simp-all*

lemma *le-Integer-numeral-simps* [*simp*]:
$le{\cdot}(0{::}Integer){\cdot}0 = TT$
$le{\cdot}(0{::}Integer){\cdot}1 = TT$
$le{\cdot}(1{::}Integer){\cdot}0 = FF$
$le{\cdot}(1{::}Integer){\cdot}1 = TT$
$le{\cdot}(0{::}Integer){\cdot}(numeral\ k) = TT$
$le{\cdot}(numeral\ k){\cdot}(0{::}Integer) = FF$
$le{\cdot}(1{::}Integer){\cdot}(numeral\ k) = TT$
$Num.One < k \Longrightarrow le{\cdot}(numeral\ k){\cdot}(1{::}Integer) = FF$
$le{\cdot}(numeral\ k{::}Integer){\cdot}(numeral\ l) = (if\ k \leq l\ then\ TT\ else\ FF)$

unfolding *zero-Integer-def one-Integer-def numeral-Integer-eq*
by *simp-all*

lemma *MkI-eq-0-iff* [*simp*]: *MkI·n = 0 ⟷ n = 0*
unfolding *zero-Integer-def* **by** *simp*

lemma *MkI-eq-1-iff* [*simp*]: *MkI·n = 1 ⟷ n = 1*
unfolding *one-Integer-def* **by** *simp*

lemma *MkI-eq-numeral-iff* [*simp*]: *MkI·n = numeral k ⟷ n = numeral k*
unfolding *numeral-Integer-eq* **by** *simp*

lemma *MkI-0*: *MkI·0 = 0*
by *simp*

lemma *MkI-1*: *MkI·1 = 1*
by *simp*

lemma *le-plus-1*:
fixes *m* :: *Integer*
assumes *le·m·n = TT*
shows *le·m·(n + 1) = TT*
proof −
from *assms* **have** *n ≠ ⊥* **by** *auto*
then have *le·n·(n + 1) = TT*
by (*cases n*) (*auto*, *metis le-MkI-MkI less-add-one less-le-not-le one-Integer-def plus-MkI-MkI*)
with *assms* **show** *?thesis* **by** (*auto dest*: *le-trans*)
qed

7.1 Induction rules that do not break the abstraction

lemma *nonneg-Integer-induct* [*consumes 1*, *case-names 0 step*]:
fixes *i* :: *Integer*
assumes *i-nonneg*: *le·0·i = TT*
and *zero*: *P 0*
and *step*: *⋀i. le·1·i = TT ⟹ P (i − 1) ⟹ P i*
shows *P i*
proof (*cases i*)
case *bottom*
then have *False* **using** *i-nonneg* **by** *simp*
then show *?thesis* **..**
next
case (*MkI integer*)
show *?thesis*
proof (*cases integer*)
case *neg*
then have *False* **using** *i-nonneg MkI* **by** (*auto simp add*: *zero-Integer-def one-Integer-def*)

```
    then show ?thesis ..
  next
    case (nonneg nat)
    have P (MkI·(int nat))
    proof(induct nat)
      case 0
      show ?case using zero by (simp add: zero-Integer-def)
    next
      case (Suc nat)
      have le·1·(MkI·(int (Suc nat))) = TT
        by (simp add: one-Integer-def)
      moreover
      have P (MkI·(int (Suc nat)) - 1)
        using Suc
        by (simp add: one-Integer-def)
      ultimately
      show ?case
        by (rule step)
    qed
    then show ?thesis using nonneg MkI by simp
  qed
qed

end
```

8 Data: List

```
theory Data-List
  imports
    Type-Classes
    Data-Function
    Data-Bool
    Data-Tuple
    Data-Integer
    Numeral-Cpo
begin

no-notation (ASCII)
  Set.member  ('(:')) and
  Set.member  ((-/ : -) [51, 51] 50)
```

8.1 Datatype definition

```
domain 'a list ([-]) =
  Nil ([]) |
  Cons (lazy head :: 'a) (lazy tail :: ['a]) (infixr : 65)
```

8.1.1 Section syntax for *Cons*

syntax
  -*Cons-section* :: $'a \to ['a] \to ['a]$ ($'(:')$)
  -*Cons-section-left* :: $'a \Rightarrow ['a] \to ['a]$ ($'(-:')$)
translations
  $(x:) == (CONST\ Rep\text{-}cfun)\ (CONST\ Cons)\ x$

abbreviation *Cons-section-right* :: $['a] \Rightarrow 'a \to ['a]$ ($'(:-')$) **where**
  $(:xs) \equiv \Lambda\ x.\ x{:}xs$

syntax
  -*lazy-list* :: *args* $\Rightarrow ['a]$ ([(-)])
translations
  $[x,\ xs] == x : [xs]$
  $[x] == x : []$

abbreviation *null* :: $['a] \to tr$ **where** *null* $\equiv$ *is-Nil*

8.2 Haskell function definitions

instantiation *list* :: (*Eq*) *Eq-strict*
begin

fixrec *eq-list* :: $['a] \to ['a] \to tr$ **where**
  $eq\text{-}list{\cdot}[]{\cdot}[] = TT\ |$
  $eq\text{-}list{\cdot}(x : xs){\cdot}[] = FF\ |$
  $eq\text{-}list{\cdot}[]{\cdot}(y : ys) = FF\ |$
  $eq\text{-}list{\cdot}(x : xs){\cdot}(y : ys) = (eq{\cdot}x{\cdot}y\ andalso\ eq\text{-}list{\cdot}xs{\cdot}ys)$

instance proof
  fix *xs* :: $['a]$
  show $eq{\cdot}xs{\cdot}\bot = \bot$
    by (*cases xs*, *fixrec-simp*+)
  show $eq{\cdot}\bot{\cdot}xs = \bot$
    by *fixrec-simp*
qed

end

instance *list* :: (*Eq-sym*) *Eq-sym*
proof
  fix *xs ys* :: $['a]$
  show $eq{\cdot}xs{\cdot}ys = eq{\cdot}ys{\cdot}xs$
  proof (*induct xs arbitrary*: *ys*)
    case *Nil*
    show *?case* **by** (*cases ys*; *simp*)
  next
    case *Cons*
    then show *?case* **by** (*cases ys*; *simp add*: *eq-sym*)

```
  qed simp-all
qed

instance list :: (Eq-equiv) Eq-equiv
proof
  fix xs ys zs :: ['a]
  show eq·xs·xs ≠ FF
    by (induct xs, simp-all)
  assume eq·xs·ys = TT and eq·ys·zs = TT then show eq·xs·zs = TT
  proof (induct xs arbitrary: ys zs)
    case (Nil ys zs) then show ?case by (cases ys, simp-all)
  next
    case (Cons x xs ys zs)
    from Cons.prems show ?case
      by (cases ys; cases zs) (auto intro: eq-trans Cons.hyps)
  qed simp-all
qed

instance list :: (Eq-eq) Eq-eq
proof
  fix xs ys :: ['a]
  show eq·xs·xs ≠ FF
    by (induct xs) simp-all
  assume eq·xs·ys = TT then show xs = ys
  proof (induct xs arbitrary: ys)
    case Nil
    then show ?case by (cases ys) auto
  next
    case Cons
    then show ?case by (cases ys) auto
  qed auto
qed

instantiation list :: (Ord) Ord-strict
begin

fixrec compare-list :: ['a] → ['a] → Ordering where
  compare-list·[]·[] = EQ |
  compare-list·(x : xs)·[] = GT |
  compare-list·[]·(y : ys) = LT |
  compare-list·(x : xs)·(y : ys) =
    thenOrdering·(compare·x·y)·(compare-list·xs·ys)

instance
  by standard (fixrec-simp, rename-tac x, case-tac x, fixrec-simp+)

end

instance list :: (Ord-linear) Ord-linear
```

```
proof
  fix xs ys zs :: ['a]
  show oppOrdering·(compare·xs·ys) = compare·ys·xs
  proof (induct xs arbitrary: ys)
    case Nil
    show ?case by (cases ys; simp)
  next
    case Cons
    then show ?case by (cases ys; simp add: oppOrdering-thenOrdering)
  qed simp-all
  show xs = ys if compare·xs·ys = EQ
    using that
  proof (induct xs arbitrary: ys)
    case Nil
    then show ?case by (cases ys; simp)
  next
    case Cons
    then show ?case by (cases ys; auto elim: compare-EQ-dest)
  qed simp-all
  show compare·xs·zs = LT if compare·xs·ys = LT and compare·ys·zs = LT
    using that
  proof (induct xs arbitrary: ys zs)
    case Nil
    then show ?case by (cases ys; cases zs; simp)
  next
    case (Cons a xs)
    then show ?case
      by (cases ys; cases zs) (auto dest: compare-EQ-dest compare-LT-trans)
  qed simp-all
  show eq·xs·ys = is-EQ·(compare·xs·ys)
  proof (induct xs arbitrary: ys)
    case Nil
    show ?case by (cases ys; simp)
  next
    case Cons
    then show ?case by (cases ys; simp add: eq-conv-compare)
  qed simp-all
  show compare·xs·xs ⊑ EQ
    by (induct xs) simp-all
qed

fixrec zipWith :: ('a → 'b → 'c) → ['a] → ['b] → ['c] where
  zipWith·f·(x : xs)·(y : ys) = f·x·y : zipWith·f·xs·ys |
  zipWith·f·(x : xs)·[] = [] |
  zipWith·f·[]·ys = []

definition zip :: ['a] → ['b] → [⟨'a, 'b⟩] where
  zip = zipWith·⟨,⟩
```

fixrec *zipWith3* :: $('a \to 'b \to 'c \to 'd) \to ['a] \to ['b] \to ['c] \to ['d]$ **where**
zipWith3·*f*·(*x* : *xs*)·(*y* : *ys*)·(*z* : *zs*) = *f*·*x*·*y*·*z* : *zipWith3*·*f*·*xs*·*ys*·*zs* |
(**unchecked**) *zipWith3*·*f*·*xs*·*ys*·*zs* = []

definition *zip3* :: $['a] \to ['b] \to ['c] \to [\langle 'a, 'b, 'c\rangle]$ **where**
zip3 = *zipWith3*·⟨,,⟩

fixrec *map* :: $('a \to 'b) \to ['a] \to ['b]$ **where**
map·*f*·[] = [] |
map·*f*·(*x* : *xs*) = *f*·*x* : *map*·*f*·*xs*

fixrec *filter* :: $('a \to tr) \to ['a] \to ['a]$ **where**
filter·*P*·[] = [] |
filter·*P*·(*x* : *xs*) =
If (*P*·*x*) *then* *x* : *filter*·*P*·*xs* *else* *filter*·*P*·*xs*

fixrec *repeat* :: $'a \to ['a]$ **where**
[*simp del*]: *repeat*·*x* = *x* : *repeat*·*x*

fixrec *takeWhile* :: $('a \to tr) \to ['a] \to ['a]$ **where**
takeWhile·*p*·[] = [] |
takeWhile·*p*·(*x*:*xs*) = *If* *p*·*x* *then* *x* : *takeWhile*·*p*·*xs* *else* []

fixrec *dropWhile* :: $('a \to tr) \to ['a] \to ['a]$ **where**
dropWhile·*p*·[] = [] |
dropWhile·*p*·(*x*:*xs*) = *If* *p*·*x* *then* *dropWhile*·*p*·*xs* *else* (*x*:*xs*)

fixrec *span* :: $('a \to tr) \to ['a] \to \langle['a],['a]\rangle$ **where**
span·*p*·[] = ⟨[],[]⟩ |
span·*p*·(*x*:*xs*) = *If* *p*·*x* *then* (*case* *span*·*p*·*xs* *of* ⟨*ys*, *zs*⟩ ⇒ ⟨*x*:*ys*,*zs*⟩) *else* ⟨[], *x*:*xs*⟩

fixrec *break* :: $('a \to tr) \to ['a] \to \langle['a],['a]\rangle$ **where**
break·*p* = *span*·(*neg* *oo* *p*)

fixrec *nth* :: $['a] \to Integer \to 'a$ **where**
nth·[]·*n* = ⊥ |
nth-Cons [*simp del*]:
nth·(*x* : *xs*)·*n* = *If* *eq*·*n*·*0* *then* *x* *else* *nth*·*xs*·(*n* − *1*)

abbreviation *nth-syn* :: $['a] \Rightarrow Integer \Rightarrow 'a$ (**infixl** !! *100*) **where**
xs !! *n* ≡ *nth*·*xs*·*n*

definition *partition* :: $('a \to tr) \to ['a] \to \langle['a], ['a]\rangle$ **where**
partition = (Λ *P* *xs*. ⟨*filter*·*P*·*xs*, *filter*·(*neg* *oo* *P*)·*xs*⟩)

fixrec *iterate* :: $('a \to 'a) \to 'a \to ['a]$ **where**
iterate·*f*·*x* = *x* : *iterate*·*f*·(*f*·*x*)

fixrec *foldl* :: $('a \mathrel{->} 'b \mathrel{->} 'a) \mathrel{->} 'a \mathrel{->} ['b] \mathrel{->} 'a$ **where**
$foldl{\cdot}f{\cdot}z{\cdot}[] \quad = z \mid$
$foldl{\cdot}f{\cdot}z{\cdot}(x{:}xs) = foldl{\cdot}f{\cdot}(f{\cdot}z{\cdot}x){\cdot}xs$

fixrec *foldl1* :: $('a \mathrel{->} 'a \mathrel{->} 'a) \mathrel{->} ['a] \mathrel{->} 'a$ **where**
$foldl1{\cdot}f{\cdot}[] \quad = \bot \mid$
$foldl1{\cdot}f{\cdot}(x{:}xs) = foldl{\cdot}f{\cdot}x{\cdot}xs$

fixrec *foldr* :: $('a \to 'b \to 'b) \to 'b \to ['a] \to 'b$ **where**
$foldr{\cdot}f{\cdot}d{\cdot}[] = d \mid$
$foldr{\cdot}f{\cdot}d{\cdot}(x : xs) = f{\cdot}x{\cdot}(foldr{\cdot}f{\cdot}d{\cdot}xs)$

fixrec *foldr1* :: $('a \to 'a \to 'a) \to ['a] \to 'a$ **where**
$foldr1{\cdot}f{\cdot}[] = \bot \mid$
$foldr1{\cdot}f{\cdot}[x] = x \mid$
$foldr1{\cdot}f{\cdot}(x : (x'{:}xs)) = f{\cdot}x{\cdot}(foldr1{\cdot}f{\cdot}(x'{:}xs))$

fixrec *elem* :: $'a{::}Eq \to ['a] \to tr$ **where**
$elem{\cdot}x{\cdot}[] = FF \mid$
$elem{\cdot}x{\cdot}(y : ys) = (eq{\cdot}y{\cdot}x \ orelse \ elem{\cdot}x{\cdot}ys)$

fixrec *notElem* :: $'a{::}Eq \to ['a] \to tr$ **where**
$notElem{\cdot}x{\cdot}[] = TT \mid$
$notElem{\cdot}x{\cdot}(y : ys) = (neq{\cdot}y{\cdot}x \ andalso \ notElem{\cdot}x{\cdot}ys)$

fixrec *append* :: $['a] \to ['a] \to ['a]$ **where**
$append{\cdot}[]{\cdot}ys = ys \mid$
$append{\cdot}(x : xs){\cdot}ys = x : append{\cdot}xs{\cdot}ys$

abbreviation *append-syn* :: $['a] \Rightarrow ['a] \Rightarrow ['a]$ (**infixr** $++$ *65*) **where**
$xs \mathbin{++} ys \equiv append{\cdot}xs{\cdot}ys$

definition *concat* :: $[['a]] \to ['a]$ **where**
$concat = foldr{\cdot}append{\cdot}[]$

definition *concatMap* :: $('a \to ['b]) \to ['a] \to ['b]$ **where**
$concatMap = (\Lambda\ f.\ concat \ oo \ map{\cdot}f)$

fixrec *last* :: $['a] \mathrel{->} 'a$ **where**
$last{\cdot}[x] = x \mid$
$last{\cdot}(\text{-:}(x{:}xs)) = last{\cdot}(x{:}xs)$

fixrec *init* :: $['a] \mathrel{->} ['a]$ **where**
$init{\cdot}[x] = [] \mid$
$init{\cdot}(x{:}(y{:}xs)) = x{:}(init{\cdot}(y{:}xs))$

fixrec *reverse* :: $['a] \mathrel{->} ['a]$ **where**
$[simp \ del]{:}reverse = foldl{\cdot}(flip{\cdot}(:)){\cdot}[]$

```
fixrec the-and :: [tr] → tr where
  the-and = foldr·trand·TT

fixrec the-or :: [tr] → tr where
  the-or = foldr·tror·FF

fixrec all :: ('a → tr) → ['a] → tr where
  all·P = the-and oo (map·P)

fixrec any :: ('a → tr) → ['a] → tr where
  any·P = the-or oo (map·P)

fixrec tails :: ['a] → [['a]] where
  tails·[] = [[]] |
  tails·(x : xs) = (x : xs) : tails·xs

fixrec inits :: ['a] → [['a]] where
  inits·[] = [[]] |
  inits·(x : xs) = [[]] ++ map·(x:)·(inits·xs)

fixrec scanr :: ('a → 'b → 'b) → 'b → ['a] → ['b]
where
  scanr·f·q0·[] = [q0] |
  scanr·f·q0·(x : xs) = (
    let qs = scanr·f·q0·xs in
    (case qs of
      [] ⇒ ⊥
    | q : qs' ⇒ f·x·q : qs))

fixrec scanr1 :: ('a → 'a → 'a) → ['a] → ['a]
where
  scanr1·f·[] = [] |
  scanr1·f·(x : xs) =
    (case xs of
      [] ⇒ [x]
    | x' : xs' ⇒ (
      let qs = scanr1·f·xs in
      (case qs of
        [] ⇒ ⊥
      | q : qs' ⇒ f·x·q : qs)))

fixrec scanl :: ('a → 'b → 'a) → 'a → ['b] → ['a] where
  scanl·f·q·ls = q : (case ls of
    [] ⇒ []
  | x : xs ⇒ scanl·f·(f·q·x)·xs)

definition scanl1 :: ('a → 'a → 'a) → ['a] → ['a] where
  scanl1 = (Λ f ls. (case ls of
    [] ⇒ []
```

```
  | x : xs ⇒ scanl·f·x·xs))
```

8.2.1 Arithmetic Sequences

```
fixrec upto :: Integer → Integer → [Integer] where
  [simp del]: upto·x·y = If le·x·y then x : upto·(x+1)·y else []

fixrec intsFrom :: Integer → [Integer] where
  [simp del]: intsFrom·x = seq·x·(x : intsFrom·(x+1))

class Enum =
  fixes toEnum :: Integer → 'a
    and fromEnum :: 'a → Integer
begin

definition succ :: 'a → 'a where
  succ = toEnum oo (+1) oo fromEnum

definition pred :: 'a → 'a where
  pred = toEnum oo (−1) oo fromEnum

definition enumFrom :: 'a → ['a] where
  enumFrom = (Λ x. map·toEnum·(intsFrom·(fromEnum·x)))

definition enumFromTo :: 'a → 'a → ['a] where
  enumFromTo = (Λ x y. map·toEnum·(upto·(fromEnum·x)·(fromEnum·y)))

end

abbreviation enumFrom-To-syn :: 'a::Enum ⇒ 'a ⇒ ['a] ((1[-../-])) where
  [m..n] ≡ enumFromTo·m·n

abbreviation enumFrom-syn :: 'a::Enum ⇒ ['a] ((1[-..])) where
  [n..] ≡ enumFrom·n

instantiation Integer :: Enum
begin
definition [simp]: toEnum = ID
definition [simp]: fromEnum = ID
instance ..
end

fixrec take :: Integer → ['a] → ['a] where
  [simp del]: take·n·xs = If le·n·0 then [] else
    (case xs of [] ⇒ [] | y : ys ⇒ y : take·(n − 1)·ys)

fixrec drop :: Integer → ['a] → ['a] where
  [simp del]: drop·n·xs = If le·n·0 then xs else
    (case xs of [] ⇒ [] | y : ys ⇒ drop·(n − 1)·ys)
```

fixrec *isPrefixOf* :: [′*a*::*Eq*] → [′*a*] → *tr* **where**
isPrefixOf·[]·- = *TT* |
isPrefixOf·(*x*:*xs*)·[] = *FF* |
isPrefixOf·(*x*:*xs*)·(*y*:*ys*) = (*eq*·*x*·*y andalso isPrefixOf*·*xs*·*ys*)

fixrec *isSuffixOf* :: [′*a*::*Eq*] → [′*a*] → *tr* **where**
isSuffixOf·*x*·*y* = *isPrefixOf*·(*reverse*·*x*)·(*reverse*·*y*)

fixrec *intersperse* :: ′*a* → [′*a*] → [′*a*] **where**
intersperse·*sep*·[] = [] |
intersperse·*sep*·[*x*] = [*x*] |
intersperse·*sep*·(*x*:*y*:*xs*) = *x*:*sep*:*intersperse*·*sep*·(*y*:*xs*)

fixrec *intercalate* :: [′*a*] → [[′*a*]] → [′*a*] **where**
intercalate·*xs*·*xss* = *concat*·(*intersperse*·*xs*·*xss*)

definition *replicate* :: *Integer* → ′*a* → [′*a*] **where**
replicate = (Λ *n x*. *take*·*n*·(*repeat*·*x*))

definition *findIndices* :: (′*a* → *tr*) → [′*a*] → [*Integer*] **where**
findIndices = (Λ *P xs*.
map·*snd*·(*filter*·(Λ ⟨*x*, *i*⟩. *P*·*x*)·(*zip*·*xs*·[*0*..])))

fixrec *length* :: [′*a*] → *Integer* **where**
length·[] = *0* |
length·(*x* : *xs*) = *length*·*xs* + *1*

fixrec *delete* :: ′*a*::*Eq* → [′*a*] → [′*a*] **where**
delete·*x*·[] = [] |
delete·*x*·(*y* : *ys*) = *If eq*·*x*·*y then ys else y* : *delete*·*x*·*ys*

fixrec *diff* :: [′*a*::*Eq*] → [′*a*] → [′*a*] **where**
diff·*xs*·[] = *xs* |
diff·*xs*·(*y* : *ys*) = *diff*·(*delete*·*y*·*xs*)·*ys*

abbreviation *diff-syn* :: [′*a*::*Eq*] ⇒ [′*a*] ⇒ [′*a*] (**infixl** \\ *70*) **where**
xs \\ *ys* ≡ *diff*·*xs*·*ys*

8.3 Logical predicates on lists

inductive *finite-list* :: [′*a*] ⇒ *bool* **where**
Nil [*intro*!, *simp*]: *finite-list* [] |
Cons [*intro*!, *simp*]: ⋀*x xs*. *finite-list xs* ⟹ *finite-list* (*x* : *xs*)

inductive-cases *finite-listE* [*elim*!]: *finite-list* (*x* : *xs*)

lemma *finite-list-upwards*:
assumes *finite-list xs* **and** *xs* ⊑ *ys*

shows *finite-list ys*
using *assms*
proof (*induct xs arbitrary*: *ys*)
case *Nil*
then have *ys* = [] **by** (*cases ys*) *simp+*
then show *?case* **by** *auto*
next
case (*Cons x xs*)
from ‹*x* : *xs* ⊑ *ys*› **obtain** *y ys′* **where** *ys* = *y* : *ys′* **by** (*cases ys*) *auto*
with ‹*x* : *xs* ⊑ *ys*› **have** *xs* ⊑ *ys′* **by** *auto*
then have *finite-list ys′* **by** (*rule Cons.hyps*)
with ‹*ys* = -› **show** *?case* **by** *auto*
qed

lemma *adm-finite-list* [*simp*]: *adm finite-list*
by (*metis finite-list-upwards adm-upward*)

lemma *bot-not-finite-list* [*simp*]:
finite-list ⊥ = *False*
by (*rule*, *cases rule*: *finite-list.cases*) *auto*

inductive *listmem* :: *′a* ⇒ [*′a*] ⇒ *bool* **where**
listmem x (*x* : *xs*) |
listmem x xs ⟹ *listmem x* (*y* : *xs*)

lemma *listmem-simps* [*simp*]:
shows ¬ *listmem x* ⊥ **and** ¬ *listmem x* []
and *listmem x* (*y* : *ys*) ⟷ *x* = *y* ∨ *listmem x ys*
by (*auto elim*: *listmem.cases intro*: *listmem.intros*)

definition *set* :: [*′a*] ⇒ *′a set* **where**
set xs = {*x*. *listmem x xs*}

lemma *set-simps* [*simp*]:
shows *set* ⊥ = {} **and** *set* [] = {}
and *set* (*x* : *xs*) = *insert x* (*set xs*)
unfolding *set-def* **by** *auto*

inductive *distinct* :: [*′a*] ⇒ *bool* **where**
Nil [*intro*!, *simp*]: *distinct* [] |
Cons [*intro*!, *simp*]: ⋀*x xs*. *distinct xs* ⟹ *x* ∉ *set xs* ⟹ *distinct* (*x* : *xs*)

8.4 Properties

lemma *map-strict* [*simp*]:
map·*P*·⊥ = ⊥
by (*fixrec-simp*)

lemma *map-ID* [*simp*]:

$map{\cdot}ID{\cdot}xs = xs$
by (*induct xs*) *simp-all*

lemma *enumFrom-intsFrom-conv* [*simp*]:
$enumFrom = intsFrom$
by (*intro cfun-eqI*) (*simp add*: *enumFrom-def*)

lemma *enumFromTo-upto-conv* [*simp*]:
$enumFromTo = upto$
by (*intro cfun-eqI*) (*simp add*: *enumFromTo-def*)

lemma *zipWith-strict* [*simp*]:
$zipWith{\cdot}f{\cdot}\bot{\cdot}ys = \bot$
$zipWith{\cdot}f{\cdot}(x : xs){\cdot}\bot = \bot$
by *fixrec-simp+*

lemma *zip-simps* [*simp*]:
$zip{\cdot}(x : xs){\cdot}(y : ys) = \langle x,\ y\rangle : zip{\cdot}xs{\cdot}ys$
$zip{\cdot}(x : xs){\cdot}[] = []$
$zip{\cdot}(x : xs){\cdot}\bot = \bot$
$zip{\cdot}[]{\cdot}ys = []$
$zip{\cdot}\bot{\cdot}ys = \bot$
unfolding *zip-def* **by** *simp-all*

lemma *zip-Nil2* [*simp*]:
$xs \neq \bot \Longrightarrow zip{\cdot}xs{\cdot}[] = []$
by (*cases xs*) *simp-all*

lemma *nth-strict* [*simp*]:
$nth{\cdot}\bot{\cdot}n = \bot$
$nth{\cdot}xs{\cdot}\bot = \bot$
by (*fixrec-simp*) (*cases xs*, *fixrec-simp+*)

lemma *upto-strict* [*simp*]:
$upto{\cdot}\bot{\cdot}y = \bot$
$upto{\cdot}x{\cdot}\bot = \bot$
by *fixrec-simp+*

lemma *upto-simps* [*simp*]:
$n < m \Longrightarrow upto{\cdot}(MkI{\cdot}m){\cdot}(MkI{\cdot}n) = []$
$m \leq n \Longrightarrow upto{\cdot}(MkI{\cdot}m){\cdot}(MkI{\cdot}n) = MkI{\cdot}m : [MkI{\cdot}m{+}1..MkI{\cdot}n]$
by (*subst upto.simps*, *simp*)+

lemma *filter-strict* [*simp*]:
$filter{\cdot}P{\cdot}\bot = \bot$
by (*fixrec-simp*)

lemma *nth-Cons-simp* [*simp*]:
$eq{\cdot}n{\cdot}0 = TT \Longrightarrow nth{\cdot}(x : xs){\cdot}n = x$

eq·n·0 = FF ⟹ nth·(x : xs)·n = nth·xs·(n − 1)
by (*subst nth.simps*, *simp*)+

lemma *nth-Cons-split*:
P (nth·(x : xs)·n) = ((eq·n·0 = FF ⟶ P (nth·(x : xs)·n)) ∧
(eq·n·0 = TT ⟶ P (nth·(x : xs)·n)) ∧
(n = ⊥ ⟶ P (nth·(x : xs)·n)))

apply (*cases n*, *simp*)
apply (*cases n = 0*, *simp-all add*: *zero-Integer-def*)
done

lemma *nth-Cons-numeral* [*simp*]:
(x : xs) !! 0 = x
(x : xs) !! 1 = xs !! 0
(x : xs) !! numeral (Num.Bit0 k) = xs !! numeral (Num.BitM k)
(x : xs) !! numeral (Num.Bit1 k) = xs !! numeral (Num.Bit0 k)
by (*simp-all add*: *nth-Cons numeral-Integer-eq*
zero-Integer-def one-Integer-def)

lemma *take-strict* [*simp*]:
take·⊥·xs = ⊥
by (*fixrec-simp*)

lemma *take-strict-2* [*simp*]:
le·1·i = TT ⟹ take·i·⊥ = ⊥
by (*subst take.simps*, *cases le·i·0*) (*auto dest*: *le-trans*)

lemma *drop-strict* [*simp*]:
drop·⊥·xs = ⊥
by (*fixrec-simp*)

lemma *isPrefixOf-strict* [*simp*]:
isPrefixOf·⊥·xs = ⊥
isPrefixOf·(x:xs)·⊥ = ⊥
by (*fixrec-simp*)+

lemma *last-strict*[*simp*]:
last·⊥= ⊥
last·(x:⊥) = ⊥
by (*fixrec-simp+*)

lemma *last-nil* [*simp*]:
last·[] = ⊥
by (*fixrec-simp*)

lemma *last-spine-strict*: ¬ *finite-list xs ⟹ last·xs = ⊥*

proof (*induct xs*)
  case (*Cons a xs*)
  then show *?case* **by** (*cases xs*) *auto*
qed *auto*

lemma *init-strict* [*simp*]:
  init·⊥= ⊥
  init·(*x*:⊥) = ⊥
  by (*fixrec-simp*+)

lemma *init-nil* [*simp*]:
  init·[] = ⊥
  by (*fixrec-simp*)

lemma *strict-foldr-strict2* [*simp*]:
  (⋀*x*. *f*·*x*·⊥ = ⊥) ⟹ *foldr*·*f*·⊥·*xs* = ⊥
  by (*induct xs*, *auto*) *fixrec-simp*

lemma *foldr-strict* [*simp*]:
  foldr·*f*·*d*·⊥ = ⊥
  foldr·*f*·⊥·[] = ⊥
  foldr·⊥·*d*·(*x* : *xs*) = ⊥
  by *fixrec-simp*+

lemma *foldr-Cons-Nil* [*simp*]:
  foldr·(:)·[]·*xs* = *xs*
  by (*induct xs*) *simp*+

lemma *append-strict1* [*simp*]:
  ⊥ ++ *ys* = ⊥
  by *fixrec-simp*

lemma *foldr-append* [*simp*]:
  foldr·*f*·*a*·(*xs* ++ *ys*) = *foldr*·*f*·(*foldr*·*f*·*a*·*ys*)·*xs*
  by (*induct xs*) *simp*+

lemma *foldl-strict* [*simp*]:
  foldl·*f*·*d*·⊥ = ⊥
  foldl·*f*·⊥·[] = ⊥
  by *fixrec-simp*+

lemma *foldr1-strict* [*simp*]:
  foldr1·*f*·⊥= ⊥
  foldr1·*f*·(*x*:⊥)= ⊥
  by *fixrec-simp*+

lemma *foldl1-strict* [*simp*]:
  foldl1·*f*·⊥= ⊥
  by *fixrec-simp*

lemma *foldl-spine-strict*:
  $\neg$ *finite-list xs* $\Longrightarrow$ *foldl*·*f*·*x*·*xs* = $\bot$
  by (*induct xs arbitrary*: *x*) *auto*

lemma *foldl-assoc-foldr*:
  assumes *finite-list xs*
    and *assoc*: $\bigwedge x\ y\ z.\ f{\cdot}(f{\cdot}x{\cdot}y){\cdot}z = f{\cdot}x{\cdot}(f{\cdot}y{\cdot}z)$
    and *neutr1*: $\bigwedge x.\ f{\cdot}z{\cdot}x = x$
    and *neutr2*: $\bigwedge x.\ f{\cdot}x{\cdot}z = x$
  shows *foldl*·*f*·*z*·*xs* = *foldr*·*f*·*z*·*xs*
  using ‹*finite-list xs*›
proof (*induct xs*)
  case (*Cons x xs*)
  from ‹*finite-list xs*› **have** *step*: $\bigwedge y.\ f{\cdot}y{\cdot}(foldl{\cdot}f{\cdot}z{\cdot}xs) = foldl{\cdot}f{\cdot}(f{\cdot}z{\cdot}y){\cdot}xs$
  proof (*induct xs*)
    case (*Cons x xs y*)
    have $f{\cdot}y{\cdot}(foldl{\cdot}f{\cdot}z{\cdot}(x : xs)) = f{\cdot}y{\cdot}(foldl{\cdot}f{\cdot}(f{\cdot}z{\cdot}x){\cdot}xs)$ **by** *auto*
    also have ... = $f{\cdot}y{\cdot}(f{\cdot}x{\cdot}(foldl{\cdot}f{\cdot}z{\cdot}xs))$ **by** (*simp only*: *Cons.hyps*)
    also have ... = $f{\cdot}(f{\cdot}y{\cdot}x){\cdot}(foldl{\cdot}f{\cdot}z{\cdot}xs)$ **by** (*simp only*: *assoc*)
    also have ... = $foldl{\cdot}f{\cdot}(f{\cdot}z{\cdot}(f{\cdot}y{\cdot}x)){\cdot}xs$ **by** (*simp only*: *Cons.hyps*)
    also have ... = $foldl{\cdot}f{\cdot}(f{\cdot}(f{\cdot}z{\cdot}y){\cdot}x){\cdot}xs$ **by** (*simp only*: *assoc*)
    also have ... = $foldl{\cdot}f{\cdot}(f{\cdot}z{\cdot}y){\cdot}(x : xs)$ **by** *auto*
    finally show *?case*.
  qed (*simp add*: *neutr1 neutr2*)

  have $foldl{\cdot}f{\cdot}z{\cdot}(x : xs) = foldl{\cdot}f{\cdot}(f{\cdot}z{\cdot}x){\cdot}xs$ **by** *auto*
  also have ... = $f{\cdot}x{\cdot}(foldl{\cdot}f{\cdot}z{\cdot}xs)$ **by** (*simp only*: *step*)
  also have ... = $f{\cdot}x{\cdot}(foldr{\cdot}f{\cdot}z{\cdot}xs)$ **by** (*simp only*: *Cons.hyps*)
  also have ... = $(foldr{\cdot}f{\cdot}z{\cdot}(x{:}xs))$ **by** *auto*
  finally show *?case* .
qed *auto*

lemma *elem-strict* [*simp*]:
  elem·*x*·$\bot$ = $\bot$
  by *fixrec-simp*

lemma *notElem-strict* [*simp*]:
  notElem·*x*·$\bot$ = $\bot$
  by *fixrec-simp*

lemma *list-eq-nil*[*simp*]:
  eq·*l*·[] = *TT* $\longleftrightarrow$ *l* = []
  eq·[]·*l* = *TT* $\longleftrightarrow$ *l* = []
  by (*cases l*, *auto*)+

lemma *take-Nil* [*simp*]:
  n $\neq \bot \Longrightarrow$ *take*·*n*·[] = []
  by (*subst take.simps*) (*cases le*·*n*·*0*; *simp*)

lemma *take-0* [*simp*]:
take·0·xs = []
take·(MkI·0)·xs = []
by (*subst take.simps*, *simp add*: *zero-Integer-def*)+

lemma *take-Cons* [*simp*]:
le·1·i = TT ⟹ take·i·(x:xs) = x : take·(i − 1)·xs
by (*subst take.simps*, *cases le·i·0*) (*auto dest*: *le-trans*)

lemma *take-MkI-Cons* [*simp*]:
0 < n ⟹ take·(MkI·n)·(x : xs) = x : take·(MkI·(n − 1))·xs
by (*subst take.simps*) (*simp add*: *zero-Integer-def one-Integer-def*)

lemma *take-numeral-Cons* [*simp*]:
take·1·(x : xs) = [x]
take·(numeral (Num.Bit0 k))·(x : xs) = x : take·(numeral (Num.BitM k))·xs
take·(numeral (Num.Bit1 k))·(x : xs) = x : take·(numeral (Num.Bit0 k))·xs
by (*subst take.simps*,
simp add: *zero-Integer-def one-Integer-def numeral-Integer-eq*)+

lemma *drop-0* [*simp*]:
drop·0·xs = xs
drop·(MkI·0)·xs = xs
by (*subst drop.simps*, *simp add*: *zero-Integer-def*)+

lemma *drop-pos* [*simp*]:
le·n·0 = FF ⟹ drop·n·xs = (case xs of [] ⇒ [] | y : ys ⇒ drop·(n − 1)·ys)
by (*subst drop.simps*, *simp*)

lemma *drop-numeral-Cons* [*simp*]:
drop·1·(x : xs) = xs
drop·(numeral (Num.Bit0 k))·(x : xs) = drop·(numeral (Num.BitM k))·xs
drop·(numeral (Num.Bit1 k))·(x : xs) = drop·(numeral (Num.Bit0 k))·xs
by (*subst drop.simps*,
simp add: *zero-Integer-def one-Integer-def numeral-Integer-eq*)+

lemma *take-drop-append*:
take·(MkI·i)·xs ++ drop·(MkI·i)·xs = xs
proof (*cases i*)
case (*nonneg n*)
then show *?thesis*
proof (*induct n arbitrary* : *i xs*)
case (*Suc n*)
thus *?case*
apply (*subst drop.simps*)
apply (*subst take.simps*)
apply (*cases xs*)
apply (*auto simp add*: *zero-Integer-def one-Integer-def*)

```
    done
  qed simp
next
  case (neg nat)
  then show ?thesis
    apply (subst drop.simps)
    apply (subst take.simps)
    apply (auto simp add: zero-Integer-def one-Integer-def )
    done
qed

lemma take-intsFrom-enumFrom [simp]:
  take·(MkI·n)·[MkI·i..] = [MkI·i..MkI·(n+i) − 1]
proof (cases n)
  fix m
  assume n = int m
  then show ?thesis
  proof (induct m arbitrary: n i)
    case 0 then show ?case by (simp add: one-Integer-def)
  next
    case (Suc m)
    then have n − 1 = int m by simp
    from Suc(1) [OF this]
    have take·(MkI·(n − 1))·[MkI·(i+1)..] = [MkI·(i+1)..MkI·(n − 1 + (i+1))
− 1] .
    moreover have (n − 1) + (i+1) − 1 = n + i − 1 by arith
   ultimately have IH: take·(MkI·(n − 1))·[MkI·(i+1)..] = [MkI·(i+1)..MkI·(n+i)
− 1] by simp
    from Suc(2) have gt: n > 0 by simp
      moreover have [MkI·i..] = MkI·i : [MkI·i + 1..] by (simp, subst ints-
From.simps) simp
    ultimately
    have ∗: take·(MkI·n)·[MkI·i..] = MkI·i : take·(MkI·(n − 1))·[MkI·(i+1)..]
      by (simp add: one-Integer-def)
    show ?case unfolding IH ∗ using gt by (simp add: one-Integer-def)
  qed
next
  fix m
  assume n = − int m
  then have n ≤ 0 by simp
  then show ?thesis
    by (subst take.simps) (simp add: zero-Integer-def one-Integer-def)
qed

lemma drop-intsFrom-enumFrom [simp]:
  assumes n ≥ 0
  shows drop·(MkI·n)·[MkI·i..] = [MkI·(n+i)..]
proof−
  from assms obtain n′ where n = int n′ by (cases n, auto)
```

```
  then show ?thesis
    apply(induct n' arbitrary: n i )
     apply simp
    apply (subst intsFrom.simps[unfolded enumFrom-intsFrom-conv[symmetric]])
    apply (subst drop.simps)
    apply (auto simp add: zero-Integer-def one-Integer-def)
    apply (rule cfun-arg-cong)
    apply (rule cfun-arg-cong)
    apply arith
    done
qed

lemma last-append-singleton:
  finite-list xs ⟹ last·(xs ++ [x]) = x
proof (induct xs rule:finite-list.induct)
  case (Cons x xs)
  then show ?case by (cases xs) auto
qed auto

lemma init-append-singleton:
  finite-list xs ⟹ init·(xs ++ [x]) = xs
proof (induct xs rule:finite-list.induct)
  case (Cons x xs)
  then show ?case by (cases xs) auto
qed auto

lemma append-Nil2 [simp]:
  xs ++ [] = xs
  by (induct xs) simp-all

lemma append-assoc [simp]:
  (xs ++ ys) ++ zs = xs ++ ys ++ zs
  by (induct xs) simp-all

lemma concat-simps [simp]:
  concat·[] = []
  concat·(xs : xss) = xs ++ concat·xss
  concat·⊥ = ⊥
  unfolding concat-def by simp-all

lemma concatMap-simps [simp]:
  concatMap·f·[] = []
  concatMap·f·(x : xs) = f·x ++ concatMap·f·xs
  concatMap·f·⊥ = ⊥
  unfolding concatMap-def by simp-all

lemma filter-append [simp]:
  filter·P·(xs ++ ys) = filter·P·xs ++ filter·P·ys
proof (induct xs)
```

case (*Cons x xs*) **then show** *?case* **by** (*cases P·x*) (*auto simp*: *If-and-if*)
qed *simp-all*

lemma *elem-append* [*simp*]:
elem·x·(*xs* ++ *ys*) = (*elem·x·xs orelse elem·x·ys*)
by (*induct xs*) *auto*

lemma *filter-filter* [*simp*]:
filter·P·(*filter·Q·xs*) = *filter·*(Λ *x*. *Q·x andalso P·x*)*·xs*
by (*induct xs*) (*auto simp*: *If2-def* [*symmetric*] *split*: *split-If2*)

lemma *filter-const-TT* [*simp*]:
filter·(Λ -. *TT*)*·xs* = *xs*
by (*induct xs*) *simp-all*

lemma *tails-strict* [*simp*]:
tails·⊥ = ⊥
by *fixrec-simp*

lemma *inits-strict* [*simp*]:
inits·⊥ = ⊥
by *fixrec-simp*

lemma *the-and-strict* [*simp*]:
the-and·⊥ = ⊥
by *fixrec-simp*

lemma *the-or-strict* [*simp*]:
the-or·⊥ = ⊥
by *fixrec-simp*

lemma *all-strict* [*simp*]:
all·P·⊥ = ⊥
by *fixrec-simp*

lemma *any-strict* [*simp*]:
any·P·⊥ = ⊥
by *fixrec-simp*

lemma *tails-neq-Nil* [*simp*]:
tails·xs ≠ []
by (*cases xs*) *simp-all*

lemma *inits-neq-Nil* [*simp*]:
inits·xs ≠ []
by (*cases xs*) *simp-all*

lemma *Nil-neq-tails* [*simp*]:
[] ≠ *tails·xs*

by (*cases xs*) *simp-all*

lemma *Nil-neq-inits* [*simp*]:
[] ≠ *inits*·*xs*
by (*cases xs*) *simp-all*

lemma *finite-list-not-bottom* [*simp*]:
assumes *finite-list xs* **shows** *xs* ≠ ⊥
using *assms* **by** (*cases*) *simp-all*

lemma *head-append* [*simp*]:
head·(*xs* ++ *ys*) = *If null*·*xs then head*·*ys else head*·*xs*
by (*cases xs*) *simp-all*

lemma *filter-cong*:
∀ *x*∈*set xs*. *p*·*x* = *q*·*x* ⟹ *filter*·*p*·*xs* = *filter*·*q*·*xs*
proof (*induct arbitrary*: *xs rule*: *filter.induct*)
case (*3 x*)
then show *?case* **by** (*cases xs*) *auto*
qed *simp-all*

lemma *filter-TT* [*simp*]:
assumes ∀ *x*∈*set xs*. *P*·*x* = *TT*
shows *filter*·*P*·*xs* = *xs*
by (*rule filter-cong* [*of xs P* Λ -. *TT*, *simplified*, *OF assms*])

lemma *filter-FF* [*simp*]:
assumes *finite-list xs*
and ∀ *x*∈*set xs*. *P*·*x* = *FF*
shows *filter*·*P*·*xs* = []
using *assms* **by** (*induct xs*) *simp-all*

lemma *map-cong*:
∀ *x*∈*set xs*. *p*·*x* = *q*·*x* ⟹ *map*·*p*·*xs* = *map*·*q*·*xs*
proof (*induct arbitrary*: *xs rule*: *map.induct*)
case (*3 x*)
then show *?case* **by** (*cases xs*) *auto*
qed *simp-all*

lemma *finite-list-upto*:
finite-list (*upto*·(*MkI*·*m*)·(*MkI*·*n*)) (**is** *?P m n*)
proof (*cases n* − *m*)
fix *d*
assume *n* − *m* = *int d*
then show *?P m n*
proof (*induct d arbitrary*: *m n*)
case (*Suc d*)
then have *n* − (*m* + *1*) = *int d* **and** *le*: *m* ≤ *n* **by** *simp-all*
from *Suc*(*1*) [*OF this*(*1*)] **have** *IH*: *?P* (*m*+*1*) *n* **.**

```
    then show ?case using le by (simp add: one-Integer-def)
  qed (simp add: one-Integer-def)
next
  fix d
  assume n − m = − int d
  then have n ≤ m by auto
  moreover
  { assume n = m then have ?P m n by (simp add: one-Integer-def) }
  moreover
  { assume n < m then have ?P m n by (simp add: one-Integer-def) }
  ultimately show ?thesis by arith
qed

lemma filter-commute:
  assumes ∀ x∈set xs. (Q·x andalso P·x) = (P·x andalso Q·x)
  shows filter·P·(filter·Q·xs) = filter·Q·(filter·P·xs)
  using filter-cong [of xs Λ x. Q·x andalso P·x Λ x. P·x andalso Q·x]
    and assms by simp

lemma upto-append-intsFrom [simp]:
  assumes m ≤ n
  shows upto·(MkI·m)·(MkI·n) ++ intsFrom·(MkI·n+1) = intsFrom·(MkI·m)
    (is ?u m n ++ - = ?i m)
proof (cases n − m)
  case (nonneg i)
  with assms show ?thesis
  proof (induct i arbitrary: m n)
    case (Suc i)
    then have m + 1 ≤ n and n − (m + 1) = int i by simp-all
    from Suc(1) [OF this]
   have IH: ?u (m+1) n ++ ?i (n+1) = ?i (m+1) by (simp add: one-Integer-def)
    from ‹m + 1 ≤ n› have m ≤ n by simp
    then have ?u m n ++ ?i (n+1) = (MkI·m : ?u (m+1) n) ++ ?i (n+1)
      by (simp add: one-Integer-def)
    also have ... = MkI·m : ?i (m+1) by (simp add: IH)
    finally show ?case by (subst (2) intsFrom.simps) (simp add: one-Integer-def)
  qed (subst (2) intsFrom.simps, simp add: one-Integer-def)
next
  case (neg i)
  then have n < m by simp
  with assms show ?thesis by simp
qed

lemma set-upto [simp]:
  set (upto·(MkI·m)·(MkI·n)) = {MkI·i | i. m ≤ i ∧ i ≤ n}
  (is set (?u m n) = ?R m n)
proof (cases n − m)
  case (nonneg i)
  then show ?thesis
```

proof (*induct i arbitrary*: *m n*)
case (*Suc i*)
then have ∗: *n* − (*m* + *1*) = *int i* **by** *simp*
from *Suc*(*1*) [*OF* ∗] **have** *IH*: *set* (*?u* (*m*+*1*) *n*) = *?R* (*m*+*1*) *n* **.**
from ∗ **have** *m* ≤ *n* **by** *simp*
then have *set* (*?u m n*) = *set* (*MkI*·*m* : *?u* (*m*+*1*) *n*) **by** (*simp add*: *one-Integer-def*)
also have ... = *insert* (*MkI*·*m*) (*?R* (*m*+*1*) *n*) **by** (*simp add*: *IH*)
also have ... = *?R m n* **using** ‹*m* ≤ *n*› **by** *auto*
finally show *?case* **.**
qed (*force simp*: *one-Integer-def*)
qed *simp*

lemma *Nil-append-iff* [*iff*]:
xs ++ *ys* = [] ⟷ *xs* = [] ∧ *ys* = []
by (*induct xs*) *simp-all*

This version of definedness rule for Nil is made necessary by the reorient simproc.

lemma *bottom-neq-Nil* [*simp*]: ⊥ ≠ []
by *simp*

Simproc to rewrite [] = *x* to *x* = [].

setup ‹
Reorient-Proc.add
(*fn Const*(@{*const-name Nil*}, -) => *true* | - => *false*)
›

simproc-setup *reorient-Nil* (*Nil* = *x*) = *Reorient-Proc.proc*

lemma *set-append* [*simp*]:
assumes *finite-list xs*
shows *set* (*xs* ++ *ys*) = *set xs* ∪ *set ys*
using *assms* **by** (*induct*) *simp-all*

lemma *distinct-Cons* [*simp*]:
distinct (*x* : *xs*) ⟷ *distinct xs* ∧ *x* ∉ *set xs*
(**is** *?l* = *?r*)
proof
assume *?l* **then show** *?r* **by** (*cases*) *simp-all*
next
assume *?r* **then show** *?l* **by** *auto*
qed

lemma *finite-list-append* [*iff*]:
finite-list (*xs* ++ *ys*) ⟷ *finite-list xs* ∧ *finite-list ys*
(**is** *?l* = *?r*)
proof
presume *finite-list xs* **and** *finite-list ys*

```
  then show ?l by (induct) simp-all
next
  assume ?l then show ?r
  proof (induct xs ++ ys arbitrary: xs ys)
    case (Cons x xs)
    then show ?case by (cases xs) auto
  qed simp
qed simp-all

lemma distinct-append [simp]:
  assumes finite-list (xs ++ ys)
  shows distinct (xs ++ ys) ⟷ distinct xs ∧ distinct ys ∧ set xs ∩ set ys = {}
    (is ?P xs ys)
  using assms
proof (induct xs ++ ys arbitrary: xs ys)
  case (Cons z zs)
  show ?case
  proof (cases xs)
    note Cons' = Cons
    case (Cons u us)
    with Cons' have finite-list us
      and [simp]: zs = us ++ ys ?P us ys by simp-all
    then show ?thesis by (auto simp: Cons)
  qed (insert Cons, simp-all)
qed simp

lemma finite-set [simp]:
  assumes distinct xs
  shows finite (set xs)
  using assms by induct auto

lemma distinct-card:
  assumes distinct xs
  shows MkI·(int (card (set xs))) = length·xs
  using assms
  by (induct)
    (simp-all add: zero-Integer-def plus-MkI-MkI [symmetric] one-Integer-def ac-simps)

lemma set-delete [simp]:
  fixes xs :: ['a::Eq-eq]
  assumes distinct xs
    and ∀ x∈set xs. eq·a·x ≠ ⊥
  shows set (delete·a·xs) = set xs − {a}
  using assms
proof induct
  case (Cons x xs)
  then show ?case by (cases eq·a·x, force+)
qed simp
```

lemma *distinct-delete* [*simp*]:
fixes *xs* :: ['*a*::*Eq-eq*]
assumes *distinct xs*
and ∀ *x*∈*set xs*. *eq*·*a*·*x* ≠ ⊥
shows *distinct* (*delete*·*a*·*xs*)
using *assms*
proof *induct*
case (*Cons x xs*)
then show *?case* **by** (*cases eq*·*a*·*x*, *force*+)
qed *simp*

lemma *set-diff* [*simp*]:
fixes *xs ys* :: ['*a*::*Eq-eq*]
assumes *distinct ys* **and** *distinct xs*
and ∀ *a*∈*set ys*. ∀ *x*∈*set xs*. *eq*·*a*·*x* ≠ ⊥
shows *set* (*xs* \\ *ys*) = *set xs* − *set ys*
using *assms*
proof (*induct arbitrary*: *xs*)
case *Nil* **then show** *?case* **by** (*induct xs rule*: *distinct.induct*) *simp-all*
next
case (*Cons y ys*)
let *?xs* = *delete*·*y*·*xs*
from *Cons* **have** ∗: ∀ *x*∈*set xs*. *eq*·*y*·*x* ≠ ⊥ **by** *simp*
from *set-delete* [*OF* ‹*distinct xs*› *this*]
have ∗∗: *set ?xs* = *set xs* − {*y*} **.**
with *Cons* **have** ∀ *a*∈*set ys*. ∀ *x*∈*set ?xs*. *eq*·*a*·*x* ≠ ⊥ **by** *simp*
moreover from ∗ **and** ‹*distinct xs*› **have** *distinct ?xs* **by** *simp*
ultimately have *set* (*?xs* \\ *ys*) = *set ?xs* − *set ys*
using *Cons* **by** *blast*
then show *?case* **by** (*force simp*: ∗∗)
qed

lemma *distinct-delete-filter*:
fixes *xs* :: ['*a*::*Eq-eq*]
assumes *distinct xs*
and ∀ *x*∈*set xs*. *eq*·*a*·*x* ≠ ⊥
shows *delete*·*a*·*xs* = *filter*·(Λ *x*. *neq*·*a*·*x*)·*xs*
using *assms*
proof (*induct*)
case (*Cons x xs*)
then have *IH*: *delete*·*a*·*xs* = *filter*·(Λ *x*. *neq*·*a*·*x*)·*xs* **by** *simp*
show *?case*
proof (*cases eq*·*a*·*x*)
case *TT*
have ∀ *x*∈*set xs*. (Λ *x*. *neq*·*a*·*x*)·*x* = (Λ -. *TT*)·*x*
proof
fix *y*
assume *y* ∈ *set xs*
with *Cons*(*3*, *4*) **have** *x* ≠ *y* **and** *eq*·*a*·*y* ≠ ⊥ **by** *auto*

```
    with TT have eq·a·y = FF by (metis (no-types) eqD(1) trE)
    then show (Λ x. neq·a·x)·y = (Λ -. TT)·y by simp
  qed
  from filter-cong [OF this] and TT
  show ?thesis by simp
 qed (simp-all add: IH)
qed simp

lemma distinct-diff-filter:
 fixes xs ys :: ['a::Eq-eq]
 assumes finite-list ys
  and distinct xs
  and ∀ a∈set ys. ∀ x∈set xs. eq·a·x ≠ ⊥
 shows xs \\ ys = filter·(Λ x. neg·(elem·x·ys))·xs
 using assms
proof (induct arbitrary: xs)
 case Nil then show ?case by simp
next
 case (Cons y ys)
 let ?xs = delete·y·xs
 from Cons have ∗: ∀ x∈set xs. eq·y·x ≠ ⊥ by simp
 from set-delete [OF ‹distinct xs› this]
 have set ?xs = set xs − {y} .
 with Cons have ∀ a∈set ys. ∀ x∈set ?xs. eq·a·x ≠ ⊥ by simp
 moreover from ∗ and ‹distinct xs› have distinct ?xs by simp
 ultimately have ?xs \\ ys = filter·(Λ x. neg·(elem·x·ys))·?xs
  using Cons by blast
 then show ?case
  using ‹distinct xs› and ∗
  by (simp add: distinct-delete-filter)
qed

lemma distinct-upto [intro, simp]:
 distinct [MkI·m..MkI·n]
proof (cases n − m)
 case (nonneg i)
 then show ?thesis
 proof (induct i arbitrary: m)
  case (Suc i)
  then have n − (m + 1) = int i and m ≤ n by simp-all
  with Suc have distinct [MkI·(m+1)..MkI·n] by force
  with ‹m ≤ n› show ?case by (simp add: one-Integer-def)
 qed (simp add: one-Integer-def)
qed simp

lemma set-intsFrom [simp]:
 set (intsFrom·(MkI·m)) = {MkI·n | n. m ≤ n}
 (is set (?i m) = ?I)
proof
```

```
  show set (?i m) ⊆ ?I
  proof
    fix n
    assume n ∈ set (?i m)
    then have listmem n (?i m) by (simp add: set-def)
    then show n ∈ ?I
    proof (induct n (?i m) arbitrary: m)
      fix x xs m
      assume x : xs = ?i m
      then have x : xs = MkI·m : ?i (m+1)
        by (subst (asm) intsFrom.simps) (simp add: one-Integer-def)
      then have [simp]: x = MkI·m xs = ?i (m+1) by simp-all
      show x ∈ {MkI·n | n. m ≤ n} by simp
    next
      fix x xs y m
      assume IH: listmem x xs
        ⋀m. xs = ?i m ⟹ x ∈ {MkI·n | n. m ≤ n}
        y : xs = ?i m
      then have y : xs = MkI·m : ?i (m+1)
        by (subst (asm) (2) intsFrom.simps) (simp add: one-Integer-def)
      then have [simp]: y = MkI·m xs = ?i (m+1) by simp-all
       from IH (2) [of m+1] have x ∈ {MkI·n | n. m+1 ≤ n} by (simp add:
one-Integer-def)
      then show x ∈ {MkI·n | n. m ≤ n} by force
    qed
  qed
next
  show ?I ⊆ set (?i m)
  proof
    fix x
    assume x ∈ ?I
    then obtain n where [simp]: x = MkI·n and m ≤ n by blast
    from upto-append-intsFrom [OF this(2), symmetric]
    have ∗: set (?i m) = set (upto·(MkI·m)·(MkI·n)) ∪ set (?i (n+1))
      using finite-list-upto [of m n] by (simp add: one-Integer-def)
    show x ∈ set (?i m) using ‹m ≤ n› by (auto simp: ∗ one-Integer-def)
  qed
qed

lemma If-eq-bottom-iff [simp]:
  (If b then x else y = ⊥) ⟷ b = ⊥ ∨ b = TT ∧ x = ⊥ ∨ b = FF ∧ y = ⊥
  by (induct b) simp-all

lemma upto-eq-bottom-iff [simp]:
  upto·m·n = ⊥ ⟷ m = ⊥ ∨ n = ⊥
  by (subst upto.simps, simp)

lemma seq-eq-bottom-iff [simp]:
  seq·x·y = ⊥ ⟷ x = ⊥ ∨ y = ⊥
```

by (*simp add*: *seq-conv-if*)

lemma *intsFrom-eq-bottom-iff* [*simp*]:
intsFrom·*m* = ⊥ ⟷ *m* = ⊥
by (*subst intsFrom.simps*, *simp*)

lemma *intsFrom-split*:
assumes *m* ≥ *n*
shows [*MkI*·*n*..] = [*MkI*·*n* .. *MkI*·(*m* − *1*)] ++ [*MkI*·*m*..]
proof−
from *assms* **have** *ge*: *m* − *n* ≥ *0* **by** *arith*
have [*MkI*·*n*..] = (*take*·(*MkI*·(*m* − *n*)) · [*MkI*·*n*..]) ++ (*drop*·(*MkI*·(*m* − *n*)) · [*MkI*·*n*..])
by (*subst take-drop-append*, *rule*)
also have ... = [*MkI*·*n*.. *MkI*·(*m* − *1*)] ++ [*MkI*·*m*..]
by (*subst drop-intsFrom-enumFrom*[*OF ge*], *auto simp add*:*take-intsFrom-enumFrom*[*simplified*] *one-Integer-def*)
finally show *?thesis* **.**
qed

lemma *filter-fast-forward*:
assumes *n+1* ≤ *n′*
and ∀ *k* . *n* < *k* ⟶ *k* < *n′* ⟶ ¬ *P k*
shows *filter*·(Λ (*MkI*·*i*) . *Def* (*P i*))·[*MkI*·(*n+1*)..] = *filter*·(Λ (*MkI*·*i*) . *Def* (*P i*))·[*MkI*·*n′*..]
proof−
from *assms*(*1*)
have[*MkI*·(*n+1*)..] = [*MkI*·(*n+1*).. *MkI*·(*n′*− *1*)] ++ [*MkI*·*n′*..] (**is** - = *?l1* ++ *?l2*)
by (*subst intsFrom-split*[*of n+1 n′*], *auto*)
moreover
have *filter*·(Λ (*MkI*·*i*) . *Def* (*P i*))·[*MkI*·(*n+1*).. *MkI*·(*n′*− *1*)] = []
apply (*rule filter-FF*)
apply (*simp*, *rule finite-list-upto*)
using *assms*(*2*)
apply *auto*
done
ultimately
show *?thesis* **by** *simp*
qed

lemma *null-eq-TT-iff* [*simp*]:
null·*xs* = *TT* ⟷ *xs* = []
by (*cases xs*) *simp-all*

lemma *null-set-empty-conv*:
xs ≠ ⊥ ⟹ *null*·*xs* = *TT* ⟷ *set xs* = {}
by (*cases xs*) *simp-all*

lemma *elem-TT* [*simp*]:
fixes $x :: 'a$::*Eq-eq* **shows** *elem*·*x*·*xs* = *TT* ⟹ $x \in$ *set xs*
apply (*induct arbitrary*: *xs rule*: *elem.induct*, *simp-all*)
apply (*rename-tac xs*, *case-tac xs*, *simp-all*)
apply (*rename-tac a list*, *case-tac eq*·*a*·*x*, *force*+)
done

lemma *elem-FF* [*simp*]:
fixes $x :: 'a$::*Eq-equiv* **shows** *elem*·*x*·*xs* = *FF* ⟹ $x \notin$ *set xs*
by (*induct arbitrary*: *xs rule*: *elem.induct*, *simp-all*)
(*rename-tac xs*, *case-tac xs*, *simp-all*, *force*)

lemma *length-strict* [*simp*]:
length·⊥ = ⊥
by (*fixrec-simp*)

lemma *repeat-neq-bottom* [*simp*]:
repeat·*x* ≠ ⊥
by (*subst repeat.simps*) *simp*

lemma *list-case-repeat* [*simp*]:
list-case·*a*·*f*·(*repeat*·*x*) = *f*·*x*·(*repeat*·*x*)
by (*subst repeat.simps*) *simp*

lemma *length-append* [*simp*]:
length·(*xs* ++ *ys*) = *length*·*xs* + *length*·*ys*
by (*induct xs*) (*simp-all add*: *ac-simps*)

lemma *replicate-strict* [*simp*]:
replicate·⊥·*x* = ⊥
by (*simp add*: *replicate-def*)

lemma *replicate-0* [*simp*]:
replicate·*0*·*x* = []
replicate·(*MkI*·*0*)·*xs* = []
by (*simp add*: *replicate-def*)+

lemma *Integer-add-0* [*simp*]: *MkI*·*0* + *n* = *n*
by (*cases n*) *simp-all*

lemma *replicate-MkI-plus-1* [*simp*]:
0 ≤ *n* ⟹ *replicate*·(*MkI*·(*n*+*1*))·*x* = *x* : *replicate*·(*MkI*·*n*)·*x*
0 ≤ *n* ⟹ *replicate*·(*MkI*·(*1*+*n*))·*x* = *x* : *replicate*·(*MkI*·*n*)·*x*
by (*simp add*: *replicate-def*, *subst take.simps*, *simp add*: *one-Integer-def zero-Integer-def*)+

lemma *replicate-append-plus-conv*:
assumes *0* ≤ *m* **and** *0* ≤ *n*
shows *replicate*·(*MkI*·*m*)·*x* ++ *replicate*·(*MkI*·*n*)·*x*
= *replicate*·(*MkI*·*m* + *MkI*·*n*)·*x*

proof (*cases m*)
  case (*nonneg i*)
  with *assms* **show** *?thesis*
  proof (*induct i arbitrary*: *m*)
    case (*Suc i*)
    then have *ge*: *int i + n ≥ 0* **by** *force*
     have *replicate·(MkI·m)·x ++ replicate·(MkI·n)·x = x : (replicate·(MkI·(int i))·x ++ replicate·(MkI·n)·x)* **by** (*simp add*: *Suc*)
    also have *... = x : replicate·(MkI·(int i) + MkI·n)·x* **using** *Suc* **by** *simp*
    finally show *?case* **using** *ge* **by** (*simp add*: *Suc ac-simps*)
  qed *simp*
next
  case (*neg i*)
  with *assms* **show** *?thesis* **by** *simp*
qed

lemma *replicate-MkI-1* [*simp*]:
  replicate·(MkI·1)·x = x : []
 by (*simp add*: *replicate-def*, *subst take.simps*, *simp add*: *zero-Integer-def one-Integer-def*)

lemma *length-replicate* [*simp*]:
  assumes *0 ≤ n*
  shows *length·(replicate·(MkI·n)·x) = MkI·n*
proof (*cases n*)
  case (*nonneg i*)
  with *assms* **show** *?thesis*
    by (*induct i arbitrary*: *n*)
      (*simp-all add*: *replicate-append-plus-conv zero-Integer-def one-Integer-def*)
next
  case (*neg i*)
  with *assms* **show** *?thesis* **by** (*simp add*: *replicate-def*)
qed

lemma *map-oo* [*simp*]:
  map·f·(map·g·xs) = map·(f oo g)·xs
  by (*induct xs*) *simp-all*

lemma *nth-Cons-MkI* [*simp*]:
  0 < i ⟹ (a : xs) !! (MkI·i) = xs !! (MkI·(i − 1))
  unfolding *nth-Cons*
  by (*cases i*, *simp add*: *zero-Integer-def one-Integer-def*) (*case-tac n*, *simp-all*)

lemma *map-plus-intsFrom*:
  map·(+ MkI·n)·(intsFrom·(MkI·m)) = intsFrom·(MkI·(m+n)) (**is** *?l = ?r*)
proof (*rule list.take-lemma*)
  fix *i* **show** *list-take i·?l = list-take i·?r*
  proof (*induct i arbitrary*: *m*)
    case (*Suc i*) **then show** *?case*
      by (*subst* (*1 2*) *intsFrom.simps*) (*simp add*: *ac-simps one-Integer-def*)

qed *simp*
qed

lemma *plus-eq-MkI-conv*:
l + *n* = *MkI*·*m* ⟷ (∃ *l′ n′*. *l* = *MkI*·*l′* ∧ *n* = *MkI*·*n′* ∧ *m* = *l′* + *n′*)
by (*cases l*, *simp*) (*cases n*, *auto*)

lemma *length-ge-0*:
length·*xs* = *MkI*·*n* ⟹ *n* ≥ *0*
by (*induct xs arbitrary*: *n*) (*auto simp*: *one-Integer-def plus-eq-MkI-conv*)

lemma *length-0-conv* [*simp*]:
length·*xs* = *MkI*·*0* ⟷ *xs* = []
apply (*cases xs*)
apply (*simp-all add*: *one-Integer-def*)
apply (*case-tac length*·*list*)
apply (*auto dest*: *length-ge-0*)
done

lemma *length-ge-1* [*simp*]:
length·*xs* = *MkI*·(*1* + *int n*)
⟷ (∃ *u us*. *xs* = *u* : *us* ∧ *length*·*us* = *MkI*·(*int n*))
(**is** *?l* = *?r*)
proof
assume *?r* **then show** *?l* **by** (*auto simp*: *one-Integer-def*)
next
assume *1*: *?l*
then obtain *u us* **where** [*simp*]: *xs* = *u* : *us* **by** (*cases xs*) *auto*
from *1* **have** *2*: *1* + *length*·*us* = *MkI*·(*1* + *int n*) **by** (*simp add*: *ac-simps*)
then have *length*·*us* ≠ ⊥ **by** (*cases length*·*us*) *simp-all*
moreover from *2* **have** *length*·*us* + *1* = *MkI*·(*int n*) + *1* **by** (*simp add*: *one-Integer-def ac-simps*)
ultimately have *length*·*us* = *MkI*·(*int n*)
by (*cases length*·*us*) (*simp-all add*: *one-Integer-def*)
then show *?r* **by** *simp*
qed

lemma *finite-list-length-conv*:
finite-list xs ⟷ (∃ *n*. *length*·*xs* = *MkI*·(*int n*)) (**is** *?l* = *?r*)
proof
assume *?l* **then show** *?r*
by (*induct*, *auto simp*: *one-Integer-def*) *presburger*
next
assume *?r*
then obtain *n* **where** *length*·*xs* = *MkI*·(*int n*) **by** *blast*
then show *?l* **by** (*induct n arbitrary*: *xs*) *auto*
qed

lemma *nth-append*:

assumes *length·xs = MkI·n* **and** $n \leq m$
shows *(xs ++ ys) !! MkI·m = ys !! MkI·(m − n)*
using *assms*
proof *(induct xs arbitrary: n m)*
case *(Cons x xs)*
then have *ge*: $n \geq 0$ **by** *(blast intro: length-ge-0)*
from *Cons(2)*
have *len*: *length·xs = MkI·(n − 1)*
by *(auto simp: plus-eq-MkI-conv one-Integer-def)*
from *Cons(3)* **have** *le*: $n - 1 \leq m - 1$ **by** *simp*
{ assume $m < 0$
with *ge* **have** *?case* **using** *Cons(3)* **by** *simp* **}**
moreover
{ assume $m = 0$
with *Cons(3)* **and** *ge* **have** $n = 0$ **by** *simp*
with *Cons(2)* **have** *?case*
by *(auto dest: length-ge-0 simp: one-Integer-def plus-eq-MkI-conv)* **}**
moreover
{ assume $m > 0$
then have *?case*
by *(auto simp: Cons(1) [OF len le] zero-Integer-def one-Integer-def)* **}**
ultimately show *?case* **by** *arith*
qed *(simp-all add: zero-Integer-def)*

lemma *replicate-nth* [*simp*]:
assumes $0 \leq n$
shows *(replicate·(MkI·n)·x ++ xs) !! MkI·n = xs !! MkI·0*
using *nth-append* [*OF length-replicate* [*OF assms*], *of n*]
by *simp*

lemma *map2-zip*:
map·(Λ⟨x, y⟩. ⟨x, f·y⟩)·(zip·xs·ys) = zip·xs·(map·f·ys)
by *(induct xs arbitrary: ys) (simp-all, case-tac ys, simp-all)*

lemma *map2-filter*:
map·(Λ⟨x, y⟩. ⟨x, f·y⟩)·(filter·(Λ⟨x, y⟩. P·x)·xs)
= filter·(Λ⟨x, y⟩. P·x)·(map·(Λ⟨x, y⟩. ⟨x, f·y⟩)·xs)
apply *(induct xs, simp-all)*
apply *(rename-tac x xs, case-tac x, simp, simp)*
apply *(rename-tac a b, case-tac P·a, auto)*
done

lemma *map-map-snd*:
$f \cdot \bot = \bot \Longrightarrow$ *map·f·(map·snd·xs)*
= map·snd·(map·(Λ⟨x, y⟩. ⟨x, f·y⟩)·xs)
by *(induct xs, simp-all, rename-tac a b, case-tac a, simp-all)*

lemma *findIndices-Cons* [*simp*]:
findIndices·P·(a : xs) =

```
    If P·a then 0 : map·(+1)·(findIndices·P·xs)
    else map·(+1)·(findIndices·P·xs)
  by (auto simp: findIndices-def, subst intsFrom.simps, cases P·a)
     (simp-all
       del: map-oo
       add: map-oo [symmetric] map-map-snd one-Integer-def zero-Integer-def
       map-plus-intsFrom [of 1 0, simplified, symmetric]
       map2-zip [of (+ MkI·1), simplified]
       map2-filter [of (+ MkI·1), simplified])

lemma filter-alt-def:
  fixes xs :: ['a]
  shows filter·P·xs = map·(nth·xs)·(findIndices·P·xs)
proof −
  {
    fix f g :: Integer → 'a
      and P :: 'a → tr
      and i xs
    assume ∀ j≥i. f·(MkI·j) = g·(MkI·j)
    then have map·f·(map·snd·(filter·(Λ⟨x, i⟩. P·x)·(zip·xs·[MkI·i..])))
      = map·g·(map·snd·(filter·(Λ⟨x, i⟩. P·x)·(zip·xs·[MkI·i..])))
      by (induct xs arbitrary: i, simp-all, subst (1 2) intsFrom.simps)
        (rename-tac a b c, case-tac P·a, simp-all add: one-Integer-def)
  } note 1 = this
  {
    fix a and ys :: ['a]
    have ∀ i≥0. nth·ys·(MkI·i) = (nth·(a : ys) oo (+1))·(MkI·i)
      by (auto simp: one-Integer-def zero-Integer-def)
  } note 2 = this
  {
    fix a P and ys xs :: ['a]
    have map·(nth·(a : ys) oo (+1))·(findIndices·P·xs)
      = map·(nth·ys)·(findIndices·P·xs)
     by (simp add: findIndices-def 1 [OF 2, simplified, of ys P xs a] zero-Integer-def)
  } note 3 = this
  show ?thesis
    by (induct xs, simp-all, simp add: findIndices-def, simp add: findIndices-def)
       (rename-tac a b, case-tac P·a, simp add: findIndices-def, simp-all add: 3)
qed

abbreviation cfun-image :: ('a → 'b) ⇒ 'a set ⇒ 'b set (infixr ‘· 90) where
  f ‘· A ≡ Rep-cfun f ‘ A

lemma set-map:
  set (map·f·xs) = f ‘· set xs (is ?l = ?r)
proof
  { fix a assume listmem a xs then have listmem (f·a) (map·f·xs)
      by (induct) simp-all }
  then show ?r ⊆ ?l by (auto simp: set-def)
```

next
{ fix *a* **assume** *listmem a* (*map*·*f*·*xs*)
then have ∃ *b*. *a* = *f*·*b* ∧ *listmem b xs*
by (*induct a map*·*f*·*xs arbitrary*: *xs*)
(*rename-tac xsa*, *case-tac xsa*, *auto*)+ **}**
then show *?l* ⊆ *?r* **unfolding** *set-def* **by** *auto*
qed

8.5 *reverse* and *reverse* induction

Alternative simplification rules for *reverse* (easier to use for equational reasoning):

lemma *reverse-Nil* [*simp*]:
reverse·[] = []
by (*simp add*: *reverse.simps*)

lemma *reverse-singleton* [*simp*]:
reverse·[*x*] = [*x*]
by (*simp add*: *reverse.simps*)

lemma *reverse-strict* [*simp*]:
reverse·⊥ = ⊥
by (*simp add*: *reverse.simps*)

lemma *foldl-flip-Cons-append*:
foldl·(*flip*·(:))·*ys*·*xs* = *foldl*·(*flip*·(:))·[]·*xs* ++ *ys*
proof (*induct xs arbitrary*: *ys*)
case (*Cons x xs*)
show *?case* **by** *simp* (*metis* (*no-types*) *Cons append.simps append-assoc*)
qed *simp*+

lemma *reverse-Cons* [*simp*]:
reverse·(*x*:*xs*) = *reverse*·*xs* ++ [*x*]
by (*simp add*: *reverse.simps*)
(*subst foldl-flip-Cons-append*, *rule refl*)

lemma *reverse-append-below*:
reverse·(*xs* ++ *ys*) ⊑ *reverse*·*ys* ++ *reverse*·*xs*
apply (*induction xs*)
apply (*simp del*: *append-assoc add*: *append-assoc* [*symmetric*])+
apply (*blast intro*: *monofun-cfun append-assoc*)
done

lemma *reverse-reverse-below*:
reverse·(*reverse*·*xs*) ⊑ *xs*
proof (*induction xs*)
case (*Cons x xs*)
have *reverse*·(*reverse*·(*x*:*xs*)) = *reverse*·(*reverse*·*xs* ++ [*x*]) **by** *simp*
also have ... ⊑ *reverse*·[*x*] ++ *reverse*·(*reverse*·*xs*) **by** (*rule reverse-append-below*)

also have ... = x : $reverse{\cdot}(reverse{\cdot}xs)$ **by** *simp*
also have ... $\sqsubseteq$ x : xs **by** (*simp add*: *Cons*)
finally show *?case* **.**
qed *simp+*

lemma *reverse-append* [*simp*]:
assumes *finite-list xs*
shows $reverse{\cdot}(xs +\!\!+ ys) = reverse{\cdot}ys +\!\!+ reverse{\cdot}xs$
using *assms* **by** (*induct xs*) *simp+*

lemma *reverse-spine-strict*:
$\neg$ *finite-list xs* $\Longrightarrow$ $reverse{\cdot}xs = \bot$
by (*auto simp add*: *reverse.simps foldl-spine-strict*)

lemma *reverse-finite* [*simp*]:
assumes *finite-list xs* **shows** *finite-list* ($reverse{\cdot}xs$)
using *assms* **by** (*induct xs*) *simp+*

lemma *reverse-reverse* [*simp*]:
assumes *finite-list xs* **shows** $reverse{\cdot}(reverse{\cdot}xs) = xs$
using *assms* **by** (*induct xs*) *simp+*

lemma *reverse-induct* [*consumes 1*, *case-names Nil snoc*]:
⟦*finite-list xs*; *P* []; $\bigwedge x\ xs$. *finite-list xs* $\Longrightarrow$ *P xs* $\Longrightarrow$ *P* ($xs +\!\!+ [x]$)⟧ $\Longrightarrow$ *P xs*
apply (*subst reverse-reverse* [*symmetric*])
apply *assumption*
apply (*rule finite-list.induct*[**where** $x = reverse{\cdot}xs$])
apply *simp+*
done

lemma *length-plus-not-0*:
$le{\cdot}1{\cdot}n = TT \Longrightarrow le{\cdot}(length{\cdot}xs + n){\cdot}0 = TT \Longrightarrow False$
proof (*induct xs arbitrary*: *n*)
case *Nil* **then show** *?case*
by *auto* (*metis Ord-linear-class.le-trans dist-eq-tr*(*3*) *le-Integer-numeral-simps*(*3*))
next
case (*Cons x xs*)
from *Cons*(*1*) [*of n + 1*] **show** *?case*
using *Cons*(*2*−) **by** (*auto simp*: *ac-simps dest*: *le-plus-1*)
qed *simp+*

lemma *take-length-plus-1*:
$length{\cdot}xs \neq \bot \Longrightarrow take{\cdot}(length{\cdot}xs + 1){\cdot}(y{:}ys) = y : take{\cdot}(length{\cdot}xs){\cdot}ys$
by (*subst take.simps*, *cases* $le{\cdot}(length{\cdot}xs + 1){\cdot}0$)
(*auto*, *metis* (*no-types*) *length-plus-not-0 le-Integer-numeral-simps*(*4*))

lemma *le-length-plus*:
$length{\cdot}xs \neq \bot \Longrightarrow n \neq \bot \Longrightarrow le{\cdot}n{\cdot}(length{\cdot}xs + n) = TT$
proof (*induct xs arbitrary*: *n*)

```
  case (Cons x xs)
  then have le·(n + 1)·(length·xs + (n + 1)) = TT by simp
 moreover have le·n·(n + 1) = TT using ‹n ≠ ⊥› by (metis le-plus-1 le-refl-Integer)
  ultimately have le·n·(length·xs + (n + 1)) = TT by (blast dest: le-trans)
  then show ?case by (simp add: ac-simps)
qed simp+

lemma eq-take-length-isPrefixOf:
  eq·xs·(take·(length·xs)·ys) ⊑ isPrefixOf·xs·ys
proof (induct xs arbitrary: ys)
  case (Cons x xs)
  note IH = this
  show ?case
  proof (cases length·xs = ⊥)
    case True then show ?thesis by simp
  next
    case False
    show ?thesis
    proof (cases ys)
      case bottom
      then show ?thesis using False
        using le-length-plus [of xs 1] by simp
    next
      case Nil then show ?thesis using False by simp
    next
      case (Cons z zs)
      then show ?thesis
        using False and IH [of zs]
        by (simp add: take-length-plus-1 monofun-cfun-arg)
    qed
  qed
qed simp+

end
```

9 Data: Maybe

```
theory Data-Maybe
  imports
    Type-Classes
    Data-Function
    Data-List
    Data-Bool
begin

domain 'a Maybe = Nothing | Just (lazy 'a)

abbreviation maybe :: 'b → ('a → 'b) → 'a Maybe → 'b where
  maybe ≡ Maybe-case
```

fixrec *isJust* :: *'a Maybe* → *tr* **where**
  isJust·(*Just*·*a*) = *TT* |
  isJust·*Nothing* = *FF*

fixrec *isNothing* :: *'a Maybe* → *tr* **where**
  isNothing = *neg oo isJust*

fixrec *fromJust* :: *'a Maybe* → *'a* **where**
  fromJust·(*Just*·*a*) = *a* |
  fromJust·*Nothing* = ⊥

fixrec *fromMaybe* :: *'a* → *'a Maybe* → *'a* **where**
  fromMaybe·*d*·*Nothing* = *d* |
  fromMaybe·*d*·(*Just*·*a*) = *a*

fixrec *maybeToList* :: *'a Maybe* → [*'a*] **where**
  maybeToList·*Nothing* = [] |
  maybeToList·(*Just*·*a*) = [*a*]

fixrec *listToMaybe* :: [*'a*] → *'a Maybe* **where**
  listToMaybe·[] = *Nothing* |
  listToMaybe·(*a*:-) = *Just*·*a*

fixrec *catMaybes* :: [*'a Maybe*] → [*'a*] **where**
  catMaybes = *concatMap*·*maybeToList*

fixrec *mapMaybe* :: (*'a* → *'b Maybe*) → [*'a*] → [*'b*] **where**
  mapMaybe·*f* = *catMaybes oo map*·*f*

instantiation *Maybe* :: (*Eq*) *Eq-strict*
begin

definition
  eq = *maybe*·(*maybe*·*TT*·(Λ *y*. *FF*))·(Λ *x*. *maybe*·*FF*·(Λ *y*. *eq*·*x*·*y*))

instance proof
  fix *x* :: *'a Maybe*
  show *eq*·*x*·⊥ = ⊥
    unfolding *eq-Maybe-def* **by** (*cases x*) *simp-all*
  show *eq*·⊥·*x* = ⊥
    unfolding *eq-Maybe-def* **by** *simp*
qed

end

lemma *eq-Maybe-simps* [*simp*]:
  eq·*Nothing*·*Nothing* = *TT*

```
  eq·Nothing·(Just·y) = FF
  eq·(Just·x)·Nothing = FF
  eq·(Just·x)·(Just·y) = eq·x·y
  unfolding eq-Maybe-def by simp-all

instance Maybe :: (Eq-sym) Eq-sym
proof
  fix x y :: 'a Maybe
  show eq·x·y = eq·y·x
    by (cases x, simp, cases y, simp, simp, simp,
        cases y, simp, simp, simp add: eq-sym)
qed

instance Maybe :: (Eq-equiv) Eq-equiv
proof
  fix x y z :: 'a Maybe
  show eq·x·x ≠ FF
    by (cases x, simp-all)
  assume eq·x·y = TT and eq·y·z = TT then show eq·x·z = TT
    by (cases x, simp, cases y, simp, cases z, simp, simp, simp, simp,
        cases y, simp, simp, cases z, simp, simp, simp, fast elim: eq-trans)
qed

instance Maybe :: (Eq-eq) Eq-eq
proof
  fix x y :: 'a Maybe
  show eq·x·x ≠ FF
    by (cases x, simp-all)
  assume eq·x·y = TT then show x = y
    by (cases x, simp, cases y, simp, simp, simp,
        cases y, simp, simp, simp, fast)
qed

instantiation Maybe :: (Ord) Ord-strict
begin

definition
  compare = maybe·(maybe·EQ·(Λ y. LT))·(Λ x. maybe·GT·(Λ y. compare·x·y))

instance proof
  fix x :: 'a Maybe
  show compare·x·⊥ = ⊥
    unfolding compare-Maybe-def by (cases x) simp-all
  show compare·⊥·x = ⊥
    unfolding compare-Maybe-def by simp
qed

end
```

lemma *compare-Maybe-simps* [*simp*]:
  compare·Nothing·Nothing = *EQ*
  compare·Nothing·(*Just·y*) = *LT*
  compare·(*Just·x*)*·Nothing* = *GT*
  compare·(*Just·x*)*·*(*Just·y*) = *compare·x·y*
  unfolding *compare-Maybe-def* **by** *simp-all*

instance *Maybe* :: (*Ord-linear*) *Ord-linear*
proof
  fix *x y z* :: *'a Maybe*
  show *eq·x·y* = *is-EQ·*(*compare·x·y*)
    by (*cases x*, *simp*, *cases y*, *simp*, *simp*, *simp*,
        cases y, *simp*, *simp*, *simp add*: *eq-conv-compare*)
  show *oppOrdering·*(*compare·x·y*) = *compare·y·x*
    by (*cases x*, *simp*, (*cases y*, *simp*, *simp*, *simp*)+)
  show *compare·x·x* ⊑ *EQ*
    by (*cases x*) *simp-all*
  { assume *compare·x·y* = *EQ* **then show** *x* = *y*
      by (*cases x*, *simp*, *cases y*, *simp*, *simp*, *simp*,
          cases y, *simp*, *simp*, *simp*) (*erule compare-EQ-dest*) **}**
  { assume *compare·x·y* = *LT* **and** *compare·y·z* = *LT* **then show** *compare·x·z* = *LT*
      apply (*cases x*, *simp*)
       apply (*cases y*, *simp*, *simp*)
       apply (*cases z*, *simp*, *simp*, *simp*)
      apply (*cases y*, *simp*, *simp*)
      apply (*cases z*, *simp*, *simp*)
      apply (*auto elim*: *compare-LT-trans*)
      done **}**
qed

lemma *isJust-strict* [*simp*]: *isJust·*⊥ = ⊥ **by** (*fixrec-simp*)
lemma *fromMaybe-strict* [*simp*]: *fromMaybe·x·*⊥ = ⊥ **by** (*fixrec-simp*)
lemma *maybeToList-strict* [*simp*]: *maybeToList·*⊥ = ⊥ **by** (*fixrec-simp*)

end

10 Definedness

theory *Definedness*
  imports
    Data-List
begin

This is an attempt for a setup for better handling bottom, by a better simp setup, and less breaking the abstractions.

definition *defined* :: *'a* :: *pcpo* ⇒ *bool* **where**
  defined x = (*x* ≠ ⊥)

lemma *defined-bottom* [*simp*]: ¬ *defined* ⊥
by (*simp add*: *defined-def*)

lemma *defined-seq* [*simp*]: *defined x* ⟹ *seq*·*x*·*y* = *y*
by (*simp add*: *defined-def*)

consts *val* :: *'a*::*type* ⇒ *'b*::*type* (⟦-⟧)

val for booleans

definition *val-Bool* :: *tr* ⇒ *bool* **where**
val-Bool i = (*THE j*. *i* = *Def j*)

adhoc-overloading
val val-Bool

lemma *defined-Bool-simps* [*simp*]:
defined (*Def i*)
defined TT
defined FF
by (*simp-all add*: *defined-def*)

lemma *val-Bool-simp1* [*simp*]:
⟦*Def i*⟧ = *i*
by (*simp-all add*: *val-Bool-def TT-def FF-def*)

lemma *val-Bool-simp2* [*simp*]:
⟦*TT*⟧ = *True*
⟦*FF*⟧ = *False*
by (*simp-all add*: *TT-def FF-def*)

lemma *IF-simps* [*simp*]:
defined b ⟹ ⟦ *b* ⟧ ⟹ (*If b then x else y*) = *x*
defined b ⟹ ⟦ *b* ⟧ = *False* ⟹ (*If b then x else y*) = *y*
by (*cases b*, *simp-all*)+

lemma *defined-neg* [*simp*]: *defined* (*neg*·*b*) ⟷ *defined b*
by (*cases b*, *auto*)

lemma *val-Bool-neg* [*simp*]: *defined b* ⟹ ⟦ *neg* · *b* ⟧ = (¬ ⟦ *b* ⟧)
by (*cases b*, *auto*)

val for integers

definition *val-Integer* :: *Integer* ⇒ *int* **where**
val-Integer i = (*THE j*. *i* = *MkI*·*j*)

adhoc-overloading
val val-Integer

lemma *defined-Integer-simps* [*simp*]:

defined (*MkI*·*i*)
defined (*0*::*Integer*)
defined (*1*::*Integer*)
by (*simp-all add*: *defined-def*)

lemma *defined-numeral* [*simp*]: *defined* (*numeral x* :: *Integer*)
by (*simp add*: *defined-def*)

lemma *val-Integer-simps* [*simp*]:
⟦*MkI*·*i*⟧ = *i*
⟦*0*⟧ = *0*
⟦*1*⟧ = *1*
by (*simp-all add*: *val-Integer-def*)

lemma *val-Integer-numeral* [*simp*]: ⟦ *numeral x* :: *Integer* ⟧ = *numeral x*
by (*simp-all add*: *val-Integer-def*)

lemma *val-Integer-to-MkI*:
defined i ⟹ *i* = (*MkI* · ⟦ *i* ⟧)
apply (*cases i*)
apply (*auto simp add*: *val-Integer-def defined-def*)
done

lemma *defined-Integer-minus* [*simp*]: *defined i* ⟹ *defined j* ⟹ *defined* (*i* − (*j*::*Integer*))
apply (*cases i*, *auto*)
apply (*cases j*, *auto*)
done

lemma *val-Integer-minus* [*simp*]: *defined i* ⟹ *defined j* ⟹ ⟦ *i* − *j* ⟧ = ⟦ *i* ⟧ − ⟦ *j* ⟧
apply (*cases i*, *auto*)
apply (*cases j*, *auto*)
done

lemma *defined-Integer-plus* [*simp*]: *defined i* ⟹ *defined j* ⟹ *defined* (*i* + (*j*::*Integer*))
apply (*cases i*, *auto*)
apply (*cases j*, *auto*)
done

lemma *val-Integer-plus* [*simp*]: *defined i* ⟹ *defined j* ⟹ ⟦ *i* + *j* ⟧ = ⟦ *i* ⟧ + ⟦ *j* ⟧
apply (*cases i*, *auto*)
apply (*cases j*, *auto*)
done

lemma *defined-Integer-eq* [*simp*]: *defined* (*eq*·*a*·*b*) ⟷ *defined a* ∧ *defined* (*b*::*Integer*)
apply (*cases a*, *simp*)
apply (*cases b*, *simp*)

```
  apply simp
  done

lemma val-Integer-eq [simp]: defined a ⟹ defined b ⟹ ⟦ eq·a·b ⟧ = (⟦ a ⟧ = (⟦
b ⟧ :: int))
  apply (cases a, simp)
  apply (cases b, simp)
  apply simp
  done
```

Full induction for non-negative integers

```
lemma nonneg-full-Int-induct [consumes 1, case-names neg Suc]:
  assumes defined: defined i
  assumes neg: ⋀ i. defined i ⟹ ⟦i⟧ < 0 ⟹ P i
  assumes step: ⋀ i. defined i ⟹ 0 ≤ ⟦i⟧ ⟹ (⋀ j. defined j ⟹ ⟦ j ⟧ < ⟦ i ⟧
⟹ P j) ⟹ P i
  shows P (i::Integer)
proof (cases i)
  case bottom
  then have False using defined by simp
  then show ?thesis ..
next
  case (MkI integer)
  show ?thesis
  proof (cases integer)
    case neg
    then show ?thesis using assms(2) MkI by simp
  next
    case (nonneg nat)
    have P (MkI·(int nat))
    proof(induction nat rule:full-nat-induct)
      case (1 nat)
      have defined (MkI·(int nat)) by simp
      moreover
      have 0 ≤ ⟦ MkI·(int nat) ⟧  by simp
      moreover
      { fix j::Integer
        assume defined j and le: ⟦ j ⟧ < ⟦ MkI·(int nat) ⟧
        have P j
        proof(cases j)
          case bottom with ‹defined j› show ?thesis by simp
        next
          case (MkI integer)
          show ?thesis
          proof(cases integer)
            case (neg nat)
            have ⟦j⟧ < 0 using neg MkI by simp
            with ‹defined j›
            show ?thesis by (rule assms(2))
```

```
        next
          case (nonneg m)
          have Suc m ≤ nat using le nonneg MkI by simp
          then have P (MkI·(int m)) by (metis 1.IH)
          then show ?thesis using nonneg MkI by simp
        qed
      qed
    }
    ultimately
    show ?case
      by (rule step)
  qed
  then show ?thesis using nonneg MkI by simp
 qed
qed
```

Some list lemmas re-done with the new setup.

```
lemma nth-tail:
  defined n ⟹ ⟦ n ⟧ ≥ 0 ⟹ tail·xs !! n = xs !! (1 + n)
  apply (cases xs, simp-all)
  apply (cases n, simp)
  apply (simp add: one-Integer-def zero-Integer-def)
  done

lemma nth-zipWith:
  assumes f1 [simp]: ⋀y. f·⊥·y = ⊥
  assumes f2 [simp]: ⋀x. f·x·⊥ = ⊥
  shows zipWith·f·xs·ys !! n = f·(xs !! n)·(ys !! n)
proof (induct xs arbitrary: ys n)
  case (Cons x xs ys n) then show ?case
    by (cases ys, simp-all split:nth-Cons-split)
qed simp-all


lemma nth-neg [simp]: defined n ⟹ ⟦ n ⟧ < 0 ⟹ nth·xs·n = ⊥
proof (induction xs arbitrary: n)
  have [simp]: eq·n·0 = TT ⟷ (n::Integer) = 0 for n
    by (cases n, auto simp add: zero-Integer-def)
  case (Cons a xs n)
  have eq·n·0 = FF
    using Cons.prems
    by (cases eq·n·0) auto
  then show ?case
    using Cons.prems
    by (auto intro: Cons.IH)
qed simp-all

lemma nth-Cons-simp [simp]:
  defined n ⟹ ⟦ n ⟧ = 0 ⟹ nth·(x : xs)·n = x
```

defined $n \Longrightarrow [\![n]\!] > 0 \Longrightarrow nth{\cdot}(x : xs){\cdot}n = nth{\cdot}xs{\cdot}(n - 1)$
proof −
assume *defined* n **and** $[\![n]\!] = 0$
then have $n = 0$ **by** (*cases* n) *auto*
then show $nth{\cdot}(x : xs){\cdot}n = x$ **by** *simp*
next
assume *defined* n **and** $[\![n]\!] > 0$
then have $eq{\cdot}n{\cdot}0 = FF$ **by** (*cases* $eq{\cdot}n{\cdot}0$) *auto*
then show $nth{\cdot}(x : xs){\cdot}n = nth{\cdot}xs{\cdot}(n - 1)$ **by** *simp*
qed

end

11 List Comprehension

theory *List-Comprehension*
imports *Data-List*
begin

no-notation
disj (**infixr** | *30*)

nonterminal *llc-qual* **and** *llc-quals*

syntax
-llc :: $'a \Rightarrow$ *llc-qual* $\Rightarrow$ *llc-quals* $\Rightarrow$ [$'a$] ([- | --)
-llc-gen :: $'a \Rightarrow$ [$'a$] $\Rightarrow$ *llc-qual* (- <− -)
-llc-guard :: *tr* $\Rightarrow$ *llc-qual* (-)
-llc-let :: *letbinds* $\Rightarrow$ *llc-qual* (*let* -)
-llc-quals :: *llc-qual* $\Rightarrow$ *llc-quals* $\Rightarrow$ *llc-quals* (, --)
-llc-end :: *llc-quals* (])
-llc-abs :: $'a \Rightarrow$ [$'a$] $\Rightarrow$ [$'a$]

translations
[*e* | *p* <− *xs*] => *CONST concatMap*·(*-llc-abs p* [*e*])·*xs*
-llc e (*-llc-gen p xs*) (*-llc-quals q qs*)
=> *CONST concatMap*·(*-llc-abs p* (*-llc e q qs*))·*xs*
[*e* | *b*] => *If b then* [*e*] *else* []
-llc e (*-llc-guard b*) (*-llc-quals q qs*)
=> *If b then* (*-llc e q qs*) *else* []
-llc e (*-llc-let b*) (*-llc-quals q qs*)
=> *-Let b* (*-llc e q qs*)

parse-translation ‹
let open HOLCF-Library in
let
val NilC = Syntax.const @{*const-syntax Nil*};

fun Lam x = Syntax.const @{*const-syntax Abs-cfun*} $ *x*;

```
    fun fresh-var ts ctxt =
      let
        val ctxt' = fold Variable.declare-term ts ctxt
      in
        singleton (Variable.variant-frees ctxt' []) (x, dummyT)
      end

    fun pat-tr ctxt p e = (* %x. case x of p => e | - => [] *)
      let
        val x = Free (fresh-var [p, e] ctxt);
        val case1 = Syntax.const @{syntax-const -case1} $ p $ e;
        val case2 =
          Syntax.const @{syntax-const -case1} $
            Syntax.const @{const-syntax Pure.dummy-pattern} $ NilC;
        val cs = Syntax.const @{syntax-const -case2} $ case1 $ case2;
      in
        (* FIXME: handle HOLCF patterns correctly *)
        Syntax-Trans.abs-tr [x, Case-Translation.case-tr false ctxt [x, cs]]
        |> Lam
      end

    fun abs-tr ctxt [p, e] =
      (case Term-Position.strip-positions p of
        Free (s, T) => Lam (Syntax-Trans.abs-tr [p, e])
      | - => pat-tr ctxt p e)

  in [(@{syntax-const -llc-abs}, abs-tr)] end
end
›

lemma concatMap-singleton [simp]:
  concatMap·(Λ x. [f·x])·xs = map·f·xs
  by (induct xs) simp-all

lemma listcompr-filter [simp]:
  [x | x <- xs, P·x] = filter·P·xs
proof (induct xs)
  case (Cons a xs)
  then show ?case by (cases P·a; simp)
qed simp-all

lemma [y | let y = x*2; z = y, x <- xs] = A
  apply simp
  oops

end
```

12 The Num Class

theory *Num-Class*
imports
Type-Classes
Data-Integer
Data-Tuple
begin

12.1 Num class

class *Num-syn* =
Eq +
plus +
minus +
times +
zero +
one +
fixes *negate* :: $'a \to 'a$
and *abs* :: $'a \to 'a$
and *signum* :: $'a \to 'a$
and *fromInteger* :: *Integer* $\to 'a$

class *Num* = *Num-syn* + *plus-cpo* + *minus-cpo* + *times-cpo*

class *Num-strict* = *Num* +
assumes *plus-strict*[*simp*]:
$x + \bot = (\bot::'a::Num)$
$\bot + x = \bot$
assumes *minus-strict*[*simp*]:
$x - \bot = \bot$
$\bot - x = \bot$
assumes *mult-strict*[*simp*]:
$x * \bot = \bot$
$\bot * x = \bot$
assumes *negate-strict*[*simp*]:
negate$\cdot\bot = \bot$
assumes *abs-strict*[*simp*]:
abs$\cdot\bot = \bot$
assumes *signum-strict*[*simp*]:
signum$\cdot\bot = \bot$
assumes *fromInteger-strict*[*simp*]:
fromInteger$\cdot\bot = \bot$

class *Num-faithful* =

Num-syn +

assumes *abs-signum-eq*: $(eq{\cdot}((abs{\cdot}x) * (signum{\cdot}x)){\cdot}(x::'a::\{Num\text{-}syn\})) \sqsubseteq TT$

class *Integral* =
Num +

fixes *div mod* :: $'a \to 'a \to ('a::Num)$
fixes *toInteger* :: $'a \to Integer$
begin

fixrec *divMod* :: $'a \to 'a \to \langle 'a, 'a\rangle$ **where** $divMod{\cdot}x{\cdot}y = \langle div{\cdot}x{\cdot}y, mod{\cdot}x{\cdot}y\rangle$

fixrec *even* :: $'a \to tr$ **where** $even{\cdot}x = eq{\cdot}(div{\cdot}x{\cdot}(fromInteger{\cdot}2)){\cdot}0$
fixrec *odd* :: $'a \to tr$ **where** $odd{\cdot}x = neg{\cdot}(even{\cdot}x)$
end

class *Integral-strict* = *Integral* +
assumes *div-strict*[*simp*]:
$div{\cdot}x{\cdot}\bot = (\bot::'a::Integral)$
$div{\cdot}\bot{\cdot}x = \bot$
assumes *mod-strict*[*simp*]:
$mod{\cdot}x{\cdot}\bot = \bot$
$mod{\cdot}\bot{\cdot}x = \bot$
assumes *toInteger-strict*[*simp*]:
$toInteger{\cdot}\bot = \bot$

class *Integral-faithful* =
Integral +
Num-faithful +

assumes $eq{\cdot}y{\cdot}0 = FF \Longrightarrow div{\cdot}x{\cdot}y * y + mod{\cdot}x{\cdot}y = (x::'a::\{Integral\})$

12.2 Instances for Integer

instantiation *Integer* :: *Num-syn*
begin
definition *negate* = $(\Lambda\ (MkI{\cdot}x).\ MkI{\cdot}(uminus\ x))$
definition *abs* = $(\Lambda\ (MkI{\cdot}x)\ .\ MkI{\cdot}(|x|))$
definition *signum* = $(\Lambda\ (MkI{\cdot}x)\ .\ MkI{\cdot}(sgn\ x))$
definition *fromInteger* = $(\Lambda\ x.\ x)$
instance..
end

instance *Integer* :: *Num*
by *standard*

instance *Integer* :: *Num-faithful*
apply *standard*
apply (*rename-tac x*, *case-tac x*)
apply *simp*
apply (*simp add*: *signum-Integer-def abs-Integer-def*)
apply (*metis mult.commute mult-sgn-abs*)
done

instance *Integer* :: *Num-strict*
apply *standard*
unfolding *signum-Integer-def abs-Integer-def negate-Integer-def fromInteger-Integer-def*
by *simp-all*

instantiation *Integer* :: *Integral*
begin
definition *div* = (Λ (*MkI*·*x*) (*MkI*·*y*). *MkI*·(*Rings.divide x y*))
definition *mod* = (Λ (*MkI*·*x*) (*MkI*·*y*). *MkI*·(*Rings.modulo x y*))
definition *toInteger* = (Λ *x*. *x*)
instance ..
end

instance *Integer* :: *Integral-strict*
apply *standard*
unfolding *div-Integer-def mod-Integer-def toInteger-Integer-def*
apply *simp-all*
apply (*rename-tac x*, *case-tac x*, *simp*, *simp*)+
done

instance *Integer* :: *Integral-faithful*
apply *standard*
unfolding *div-Integer-def mod-Integer-def*
apply (*rename-tac y x*)
apply (*case-tac y*, *simp*)
apply (*case-tac x*, *simp*)
apply *simp*
done

lemma *Integer-Integral-simps*[*simp*]:
div·(*MkI*·*x*)·(*MkI*·*y*) = *MkI*·(*Rings.divide x y*)
mod·(*MkI*·*x*)·(*MkI*·*y*) = *MkI*·(*Rings.modulo x y*)
fromInteger·*i* = *i*
unfolding *mod-Integer-def div-Integer-def fromInteger-Integer-def* **by** *simp-all*

end
theory *HOLCF-Prelude*
imports
HOLCF-Main
Type-Classes

Numeral-Cpo
Data-Function
Data-Bool
Data-Tuple
Data-Integer
Data-List
Data-Maybe
begin
end
theory *Fibs*
imports
../HOLCF-Prelude
../Definedness
begin

13 Fibonacci sequence

In this example, we show that the self-recursive lazy definition of the fibonacci sequence is actually defined and correct.

fixrec *fibs* :: [*Integer*] **where**
[*simp del*]: *fibs* = *0* : *1* : *zipWith*·(+)·*fibs*·(*tail*·*fibs*)

fun *fib* :: *int* ⇒ *int* **where**
fib n = (*if n* ≤ *0 then 0 else if n* = *1 then 1 else fib* (*n* − *1*) + *fib* (*n* − *2*))

declare *fib.simps* [*simp del*]

lemma *fibs-0* [*simp*]:
fibs !! *0* = *0*
by (*subst fibs.simps*) *simp*

lemma *fibs-1* [*simp*]:
fibs !! *1* = *1*
by (*subst fibs.simps*) *simp*

And the proof that *fibs* !! *i* is defined and the fibs value.

lemma [*simp*]:−*1* + ⟦*i*⟧ = ⟦ *i* ⟧ − *1* **by** *simp*
lemma [*simp*]:−*2* + ⟦*i*⟧ = ⟦ *i* ⟧ − *2* **by** *simp*

lemma *nth-fibs*:
assumes *defined i* **and** ⟦ *i* ⟧ ≥ *0* **shows** *defined* (*fibs* !! *i*) **and** ⟦ *fibs* !! *i* ⟧ = *fib* ⟦ *i* ⟧
using *assms*
proof(*induction i rule*:*nonneg-full-Int-induct*)
case (*Suc i*)
case *1*
with *Suc* **show** *?case*
apply (*cases* ⟦*i*⟧ = *0*)

apply (*subst fibs.simps*, (*subst fib.simps*)*?*, *simp add*: *nth-zipWith nth-tail*)
apply (*cases* ⟦*i*⟧ = *1*)
apply (*subst fibs.simps*, (*subst fib.simps*)*?*, *simp add*: *nth-zipWith nth-tail*)
apply (*subst fibs.simps*, (*subst fib.simps*)*?*, *simp add*: *nth-zipWith nth-tail*)
done
qed (*subst fibs.simps*, (*subst fib.simps*)*?*, *simp add*: *nth-zipWith nth-tail*)+

end
theory *Sieve-Primes*
imports
HOL−Computational-Algebra.Primes
../Num-Class
../HOLCF-Prelude

begin

14 The Sieve of Eratosthenes

declare [[*coercion int*]]
declare [[*coercion-enabled*]]

This example proves that the well-known Haskell two-liner that lazily calculates the list of all primes does indeed do so. This proof is using coinduction.

We need to hide some constants again since we imported something from HOL not via *HOLCF−Prelude.HOLCF-Main*.

no-notation
Rings.divide (**infixl** *div 70*) **and**
Rings.modulo (**infixl** *mod 70*)

no-notation
Set.member ((:)) **and**
Set.member ((-/ : -) [*51*, *51*] *50*)

This is the implementation. We also need a modulus operator.

fixrec *sieve* :: [*Integer*] → [*Integer*] **where**
sieve·(*p* : *xs*) = *p* : (*sieve*·(*filter*·(Λ *x*. *neg*·(*eq*·(*mod*·*x*·*p*)·*0*))·*xs*))

fixrec *primes* :: [*Integer*] **where**
primes = *sieve*·[*2*..]

Simplification rules for modI:

definition *MkI′* :: *int* ⇒ *Integer* **where**
MkI′ x = *MkI*·*x*

lemma *MkI′-simps* [*simp*]:
shows *MkI′ 0* = *0* **and** *MkI′ 1* = *1* **and** *MkI′* (*numeral k*) = *numeral k*
unfolding *MkI′-def zero-Integer-def one-Integer-def numeral-Integer-eq*

by *rule+*

lemma *modI-numeral-numeral* [*simp*]:
mod·(*numeral i*)·(*numeral j*) = *MkI'* (*Rings.modulo* (*numeral i*) (*numeral j*))
unfolding *numeral-Integer-eq mod-Integer-def MkI'-def* **by** *simp*

Some lemmas demonstrating evaluation of our list:

lemma *primes* !! *0* = *2*
unfolding *primes.simps*
apply (*simp only*: *enumFrom-intsFrom-conv*)
apply (*subst intsFrom.simps*)
apply *simp*
done

lemma *primes* !! *1* = *3*
unfolding *primes.simps*
apply (*simp only*: *enumFrom-intsFrom-conv*)
apply (*subst intsFrom.simps*)
apply (*subst intsFrom.simps*)
apply *simp*
done

lemma *primes* !! *2* = *5*
unfolding *primes.simps*
apply (*simp only*: *enumFrom-intsFrom-conv*)
apply (*subst intsFrom.simps*)
apply (*subst intsFrom.simps*)
apply (*subst intsFrom.simps*)
apply (*subst intsFrom.simps*)
apply *simp*
done

lemma *primes* !! *3* = *7*
unfolding *primes.simps*
apply (*simp only*: *enumFrom-intsFrom-conv*)
apply (*subst intsFrom.simps*)
apply (*subst intsFrom.simps*)
apply (*subst intsFrom.simps*)
apply (*subst intsFrom.simps*)
apply (*subst intsFrom.simps*)
apply (*subst intsFrom.simps*)
apply (*simp del*: *filter-FF filter-TT*)

done

Auxiliary lemmas about prime numbers

lemma *find-next-prime-nat*:
fixes *n* :: *nat*
assumes *prime n*

shows $\exists\ n'.\ n' > n \wedge \mathit{prime}\ n' \wedge (\forall k.\ n < k \longrightarrow k < n' \longrightarrow \neg\ \mathit{prime}\ k)$
using *ex-least-nat-le*[*of* $\lambda\ k\ .\ k > n \wedge \mathit{prime}\ k$]
by (*metis bigger-prime not-prime-0*)

Simplification for andalso:

lemma *andAlso-Def*[*simp*]: $((\mathit{Def}\ x)\ \mathit{andalso}\ (\mathit{Def}\ y)) = \mathit{Def}\ (x \wedge y)$
by (*metis Def-bool2 Def-bool4 andalso-thms*(*1*) *andalso-thms*(*2*))

This defines the bisimulation and proves it to be a list bisimulation.

definition *prim-bisim*:
prim-bisim x1 x2 = ($\exists\ n\ .\ \mathit{prime}\ n\ \wedge$
x1 = *sieve*·(*filter*·(Λ (*MkI*·*i*). *Def* ($\forall\ d.\ d > 1 \longrightarrow d < n \longrightarrow \neg$ (*d dvd i*))))·[*MkI*·*n*..]) $\wedge$
x2 = *filter*·(Λ (*MkI*·*i*). *Def* (*prime* (*nat* |*i*|)))·[*MkI*·*n*..])

lemma *prim-bisim-is-bisim*: *list-bisim prim-bisim*
proof −
{
fix *xs ys*
assume *prim-bisim xs ys*
then obtain *n* :: *nat* **where**
prime n **and**
n > *1* **and**
xs = *sieve*·(*filter*·(Λ (*MkI*·*i*). *Def* ($\forall\ d$::*int*. *d* > *1* $\longrightarrow$ *d* < *n* $\longrightarrow \neg$ (*d dvd i*))))·[*MkI*·*n*..]) (**is** - = *sieve*·*?xs*) **and**
ys = *filter*·(Λ (*MkI*·*i*). *Def* (*prime* (*nat* |*i*|)))·[*MkI*·*n*..]

proof −
assume *a1*: $\bigwedge n.$ ⟦*prime n*; *1* < *n*; *xs* = *sieve*· (*Data-List.filter*· (Λ (*MkI*·*i*). *Def* ($\forall$ *d*>*1*. *d* < *int n* $\longrightarrow \neg$ *d dvd i*))· [*MkI*·(*int n*)..]); *ys* = *Data-List.filter*· (Λ (*MkI*·*i*). *Def* (*prime* (*nat* |*i*|)))· [*MkI*·(*int n*)..]⟧ $\Longrightarrow$ *thesis*
obtain *ii* **where**
f2: $\forall$ *is isa*. ($\neg$ *prim-bisim is isa* $\vee$ *prime* (*ii is isa*) $\wedge$ *is* = *sieve*· (*Data-List.filter*· (Λ (*MkI*·*i*). *Def* ($\forall$ *ia*>*1*. *ia* < *ii is isa* $\longrightarrow \neg$ *ia dvd i*))· [*MkI*·(*ii is isa*)..]) $\wedge$ *isa* = *Data-List.filter*· (Λ (*MkI*·*i*). *Def* (*prime* (*nat* |*i*|)))· [*MkI*·(*ii is isa*)..]) $\wedge$ (($\forall$ *i*. $\neg$ *prime i* $\vee$ *is* ≠ *sieve*· (*Data-List.filter*· (Λ (*MkI*·*ia*). *Def* ($\forall$ *ib*>*1*. *ib* < *i* $\longrightarrow \neg$ *ib dvd ia*))· [*MkI*·*i*..]) $\vee$ *isa* ≠ *Data-List.filter*· (Λ (*MkI*·*i*). *Def* (*prime* (*nat* |*i*|)))· [*MkI*·*i*..]) $\vee$ *prim-bisim is isa*)
using *prim-bisim* **by** *moura*
then have *f3*: *prime* (*ii xs ys*)
using ‹*prim-bisim xs ys*› **by** *blast*
then obtain *nn* :: *int* ⇒ *nat* **where**
f4: *int* (*nn* (*ii xs ys*)) = *ii xs ys*
by (*metis* (*no-types*) *prime-gt-0-int zero-less-imp-eq-int*)
then have *prime* (*nn* (*ii xs ys*))
using *f3* **by** (*metis* (*no-types*) *prime-nat-int-transfer*)
then show *?thesis*
using *f4 f2 a1* ‹*prim-bisim xs ys*› *prime-gt-1-nat* **by** *presburger*
qed

obtain n' **where** $n' > n$ **and** *prime* n' **and** *not-prime*: $\forall k.\ n < k \longrightarrow k < n' \longrightarrow \neg$ *prime* k
using *find-next-prime-nat*[*OF* ‹*prime* n›] **by** *auto*

{
fix k :: *int*
assume $n < k$ **and** $k < n'$

have $k > 1$ **using** ‹$n < k$› ‹$n > 1$› **by** *auto*
then obtain k' :: *nat* **where** $k =$ *int* k' **by** (*cases* k) *auto*
then obtain p **where** *prime* p **and** p *dvd* k
using ‹$k > 1$› ‹$k =$ *int* k'›
by (*metis* (*full-types*) *less-numeral-extra*(*4*) *of-nat-1 of-nat-dvd-iff prime-factor-nat prime-nat-int-transfer*)
then have $p < n'$ **using** ‹$k < n'$› ‹$k > 1$›
using *zdvd-imp-le* [*of p k*] **by** *simp*
then have $p \leq n$ **using** ‹*prime* p› *not-prime*
using *not-le prime-gt-0-int zero-less-imp-eq-int*
by (*metis of-nat-less-iff prime-nat-int-transfer*)
then have $\exists$ *d*::*int*>*1*. $d \leq n \wedge d$ *dvd* k **using** ‹p *dvd* k› ‹*prime* p› *of-nat-le-iff prime-gt-1-nat*
prime-gt-1-int **by** *auto*
}
then have *between-have-divisors*: $\bigwedge k$::*int*. $n < k \Longrightarrow k < n' \Longrightarrow \exists$ *d*::*int*>*1*. $d \leq n \wedge d$ *dvd* k**.**

{
fix i
{
assume *small*: $\forall$ *d*::*int*>*1*. $d \leq n \longrightarrow \neg$ d *dvd* i
fix d
assume $1 < d$ **and** d *dvd* i **and** $d < n'$
with *small* **have** $d > n$ **by** *auto*

obtain d'::*int* **where** $d' > 1$ **and** $d' \leq n$ **and** d' *dvd* d **using** *between-have-divisors*[*OF* ‹$n < d$› ‹$d < n'$›] **by** *auto*
with ‹d *dvd* i› *small* **have** *False* **by** (*metis* (*full-types*) *dvd-trans*)
}
then have ($\forall$ *d*::*int*. $d > 1 \longrightarrow d \leq n \longrightarrow \neg$ (d *dvd* i)) = ($\forall$ *d*::*int*. $d > 1 \longrightarrow d < n' \longrightarrow \neg$ (d *dvd* i))
using ‹$n' > n$› **by** *auto*
}
then have *between-not-relvant*: $\bigwedge$ *i*. ($\forall$ *d*::*int*. $d > 1 \longrightarrow d \leq n \longrightarrow \neg$ (d *dvd* i)) = ($\forall$ *d*::*int*. $d > 1 \longrightarrow d < n' \longrightarrow \neg$ (d *dvd* i)) **.**

from ‹*prime* n›
have $\forall$ *d*::*int* >*1*. $d < n \longrightarrow \neg$ d *dvd* n
unfolding *prime-int-altdef* **using** *int-one-le-iff-zero-less le-less*

by (*simp add*: *prime-int-not-dvd*)
then
obtain *xs'* **where** *?xs* = (*MkI*·*n*) : *xs'* **and** *xs'* = *filter*·(Λ (*MkI*·*i*). *Def* ((∀ *d*::*int*. *d* > *1* ⟶ *d* < *n* ⟶ ¬ (*d dvd i*)))·[*MkI*·(*n*+*1*)..]
by (*subst* (*asm*) *intsFrom.simps*[*unfolded enumFrom-intsFrom-conv*[*symmetric*]], *simp add*: *one-Integer-def TT-def*[*symmetric*] *add.commute*)

{
have *filter*·(Λ *x*. *neg*·(*eq*·(*mod*·*x*·(*MkI*·*n*))·*0*))·(*filter*·(Λ (*MkI*·*i*). *Def* ((∀ *d*::*int*. *d* > *1* ⟶ *d* < *n* ⟶ ¬ (*d dvd i*)))·[*MkI*·(*n*+*1*)..])
= *filter*·(Λ (*MkI*·*x*). *Def* (¬ *n dvd x*))·(*filter*·(Λ (*MkI*·*i*). *Def* ((∀ *d*::*int*. *d* > *1* ⟶ *d* < *n* ⟶ ¬ (*d dvd i*)))·[*MkI*·(*n*+*1*)..])
apply (*rule filter-cong*[*rule-format*])
apply (*rename-tac x*, *case-tac x*)
apply (*simp*)
apply (*auto simp add*: *zero-Integer-def*)
apply (*rule FF-def*)
apply (*simp add*: *TT-def*)
by (*metis dvd-eq-mod-eq-0*)
also have ... = *filter*·(Λ (*MkI*·*i*). *Def* ((¬ *n dvd i*) ∧ (∀ *d*::*int*. *d* > *1* ⟶ *d* < *n* ⟶ ¬ (*d dvd i*)))·[*MkI*·(*n*+*1*)..]
by (*auto intro*!: *filter-cong*[*rule-format*])
also have ... = *filter*·(Λ (*MkI*·*i*). *Def* ((∀ *d*::*int*. *d* > *1* ⟶ *d* ≤ *n* ⟶ ¬ (*d dvd i*)))·[*MkI*·(*n*+*1*)..]
apply (*rule filter-cong*[*rule-format*])
apply (*rename-tac x*, *case-tac x*)
using ‹*n* > *1*›
apply *auto*
done
also have ... = *filter*·(Λ (*MkI*·*i*). *Def* ((∀ *d*::*int*. *d* > *1* ⟶ *d* ≤ *n* ⟶ ¬ (*d dvd i*)))·[*MkI*·(*int n*+*1*)..]
by (*metis* (*no-types*, *lifting*) *of-nat-1 of-nat-add*)
also have ... = *filter*·(Λ (*MkI*·*i*). *Def* ((∀ *d*::*int*. *d* > *1* ⟶ *d* ≤ *n* ⟶ ¬ (*d dvd i*)))·[*MkI*·*n'*..]
apply (*rule filter-fast-forward*[*of n n'*]) **using** ‹*n'* > *n*›
apply (*auto simp add*: *between-have-divisors*)
done
also have ... = *filter*·(Λ (*MkI*·*i*). *Def* ((∀ *d*::*int*. *d* > *1* ⟶ *d* < *n'* ⟶ ¬ (*d dvd i*)))·[*MkI*·*n'*..]
by (*auto intro*: *filter-cong*[*rule-format*] *simp add*: *between-not-relvant*)
also note *calculation*
}
note *tmp* = *this*

{
have *xs* = *sieve*·*?xs* **by** *fact*
also have ... = *sieve*·((*MkI*·*n*) : *xs'*) **using** ‹*?xs* = -› **by** *simp*
also have ... = *sieve*·((*MkI*·*n*) : *filter*·(Λ (*MkI*·*i*). *Def* ((∀ *d*::*int*. *d* > *1* ⟶ *d* < *n* ⟶ ¬ (*d dvd i*)))·[*MkI*·(*n*+*1*)..]) **using** ‹*xs'* = -› **by** *simp*

```
    also have ... = (MkI·n) : sieve·(filter·(Λ (MkI·i). Def ((∀ d::int. d > 1 ⟶
d < n' ⟶ ¬ (d dvd i))))·[MkI·n'..])  using tmp by simp
    also note calculation
   }
   moreover
   {
    have ys = filter·(Λ (MkI·i). Def (prime (nat |i|)))·[MkI·n..] by fact
    also have ... = (MkI·n) : filter·(Λ (MkI·i). Def (prime (nat |i|)))·[MkI·(int
n+1)..]
      using ‹prime n›
     by (subst intsFrom.simps[unfolded enumFrom-intsFrom-conv[symmetric]])(simp
add: one-Integer-def TT-def[symmetric] add.commute)
    also have ... = (MkI·n) : filter·(Λ (MkI·i). Def (prime (nat |i|)))·[MkI·n'..]
      apply (subst filter-fast-forward[of n n']) using ‹n' > n› and not-prime
       apply auto
      apply (metis (full-types) ‹⋀k. ⟦int n < k; k < int n'⟧ ⟹ ∃ d>1. d ≤ int
n ∧ d dvd k› le-less not-le prime-gt-0-int prime-int-not-dvd zdvd-imp-le)
      done
    also note calculation
   }
   moreover
   note ‹prime n'›
   ultimately
   have ∃ p xs' ys'. xs = p : xs' ∧ ys = p : ys' ∧ prim-bisim xs' ys' unfolding
prim-bisim
    using prime-nat-int-transfer by blast
  }
  then show ?thesis unfolding list.bisim-def by metis
qed
```

Now we apply coinduction:

```
lemma sieve-produces-primes:
  fixes n :: nat
  assumes prime n
  shows sieve·(filter·(Λ (MkI·i). Def ((∀ d::int. d > 1 ⟶ d < n ⟶ ¬ (d dvd
i))))·[MkI·n..])
    = filter·(Λ (MkI·i). Def (prime (nat |i|)))·[MkI·n..]
  using assms
  apply −
  apply (rule list.coinduct[OF prim-bisim-is-bisim], auto simp add: prim-bisim)
  using prime-nat-int-transfer by blast
```

And finally show the correctness of primes.

```
theorem primes:
  shows primes = filter·(Λ (MkI·i). Def (prime (nat |i|)))·[MkI·2..]
proof −
  have primes = sieve·[2 ..] by (rule primes.simps)
  also have ... = sieve·(filter·(Λ (MkI·i). Def ((∀ d::int. d > 1 ⟶ d < (int 2)
⟶ ¬ (d dvd i))))·[MkI·(int 2)..])
```

unfolding *numeral-Integer-eq*
by (*subst filter-TT*, *auto*)
also have ... = *filter*·(Λ (*MkI*·*i*). *Def* (*prime* (*nat* |*i*|)))·[*MkI*·(*int* 2)..]
by (*rule sieve-produces-primes*[*OF two-is-prime-nat*])
also have ... = *filter*·(Λ (*MkI*·*i*). *Def* (*prime* (*nat* |*i*|)))·[*MkI*·2..]
by *simp*
finally show *?thesis* **.**
qed

end

15 GHC's "fold/build" Rule

theory *GHC-Rewrite-Rules*
imports *../HOLCF-Prelude*
begin

15.1 Approximating the Rewrite Rule

The original rule looks as follows (see also [3]):

```
"fold/build"
  forall k z (g :: forall b. (a -> b -> b) -> b -> b).
  foldr k z (build g) = g k z
```

Since we do not have rank-2 polymorphic types in Isabelle/HOL, we try to imitate a similar statement by introducing a new type that combines possible folds with their argument lists, i.e., f below is a function that, in a way, represents the list *xs*, but where list constructors are functionally abstracted.

abbreviation (*input*) *abstract-list* **where**
abstract-list xs ≡ (Λ *c n*. *foldr*·*c*·*n*·*xs*)

cpodef ('*a*, '*b*) *listfun* =
{(*f* :: ('*a* → '*b* → '*b*) → '*b* → '*b*, *xs*). *f* = *abstract-list xs*}
by *auto*

definition *listfun* :: ('*a*, '*b*) *listfun* → ('*a* → '*b* → '*b*) → '*b* → '*b* **where**
listfun = (Λ *g*. *Product-Type.fst* (*Rep-listfun g*))

definition *build* :: ('*a*, '*b*) *listfun* → ['*a*] **where**
build = (Λ *g*. *Product-Type.snd* (*Rep-listfun g*))

definition *augment* :: ('*a*, '*b*) *listfun* → ['*a*] → ['*a*] **where**
augment = (Λ *g xs*. *build*·*g* ++ *xs*)

definition *listfun-comp* :: ('*a*, '*b*) *listfun* → ('*a*, '*b*) *listfun* → ('*a*, '*b*) *listfun* **where**
listfun-comp = (Λ *g h*.
Abs-listfun (Λ *c n*. *listfun*·*g*·*c*·(*listfun*·*h*·*c*·*n*), *build*·*g* ++ *build*·*h*))

abbreviation
listfun-comp-infix :: ('*a*, '*b*) *listfun* ⇒ ('*a*, '*b*) *listfun* ⇒ ('*a*, '*b*) *listfun* (**infixl** ∘*lf* 55)
where
g ∘*lf h* ≡ *listfun-comp*·*g*·*h*

fixrec *mapFB* :: ('*b* → '*c* → '*c*) → ('*a* → '*b*) → '*a* → '*c* → '*c* **where**
mapFB·*c*·*f* = (Λ *x ys*. *c*·(*f*·*x*)·*ys*)

15.2 Lemmas

lemma *cont-listfun-body* [*simp*]:
cont (λ*g*. *Product-Type.fst* (*Rep-listfun g*))
by (*simp add*: *cont-Rep-listfun*)

lemma *cont-build-body* [*simp*]:
cont (λ*g*. *Product-Type.snd* (*Rep-listfun g*))
by (*simp add*: *cont-Rep-listfun*)

lemma *build-Abs-listfun*:
assumes *abstract-list xs* = *f*
shows *build*·(*Abs-listfun* (*f*, *xs*)) = *xs*
using *assms* **and** *Abs-listfun-inverse* [*of* (*f*, *xs*)]
by (*simp add*: *build-def*)

lemma *listfun-Abs-listfun* [*simp*]:
assumes *abstract-list xs* = *f*
shows *listfun*·(*Abs-listfun* (*f*, *xs*)) = *f*
using *assms* **and** *Abs-listfun-inverse* [*of* (*f*, *xs*)]
by (*simp add*: *listfun-def*)

lemma *augment-Abs-listfun* [*simp*]:
assumes *abstract-list xs* = *f*
shows *augment*·(*Abs-listfun* (*f*, *xs*))·*ys* = *xs* ++ *ys*
using *assms* **and** *Abs-listfun-inverse* [*of* (*f*, *xs*)]
by (*simp add*: *augment-def build-Abs-listfun*)

lemma *cont-augment-body* [*simp*]:
cont (λ*g*. *Abs-cfun* ((++) (*Product-Type.snd* (*Rep-listfun g*))))
by (*simp add*: *cont-Rep-listfun*)

lemma *fold*/*build*:
fixes *g* :: ('*a*, '*b*) *listfun*
shows *foldr*·*k*·*z*·(*build*·*g*) = *listfun*·*g*·*k*·*z*
proof −

from *Rep-listfun* **obtain** *f xs* **where**
Rep-listfun g = (*f*, *xs*) **and** *f* = *abstract-list xs* **by** *blast*
then show *?thesis* **by** (*simp add*: *build-def listfun-def*)
qed

lemma *foldr/augment*:
fixes *g* :: ('*a*, '*b*) *listfun*
shows *foldr*·*k*·*z*·(*augment*·*g*·*xs*) = *listfun*·*g*·*k*·(*foldr*·*k*·*z*·*xs*)
proof −
from *Rep-listfun* **obtain** *f ys* **where**
Rep-listfun g = (*f*, *ys*) **and** *f* = *abstract-list ys* **by** *blast*
then show *?thesis*
by (*simp add*: *augment-def build-def listfun-def*)
qed

lemma *foldr/id*:
foldr·(:)·[] = (Λ *x*. *x*)
proof (*rule cfun-eqI*)
fix *xs* :: ['*a*]
show *foldr*·(:)·[]·*xs* = (Λ *x*. *x*)·*xs*
by (*induction xs*) *simp*+
qed

lemma *foldr/app*:
foldr·(:)·*ys* = (Λ *xs*. *xs* ++ *ys*)
proof (*rule cfun-eqI*)
fix *xs* :: ['*a*]
show *foldr*·(:)·*ys*·*xs* = (Λ *xs*. *xs* ++ *ys*)·*xs* **by** (*induct xs*) *simp*+
qed

lemma *foldr/cons*: *foldr*·*k*·*z*·(*x*:*xs*) = *k*·*x*·(*foldr*·*k*·*z*·*xs*) **by** *simp*
lemma *foldr/single*: *foldr*·*k*·*z*·[*x*] = *k*·*x*·*z* **by** *simp*
lemma *foldr/nil*: *foldr*·*k*·*z*·[] = *z* **by** *simp*

lemma *cont-listfun-comp-body1* [*simp*]:
cont (λ*h*. *Abs-listfun* (Λ *c n*. *listfun*·*g*·*c*·(*listfun*·*h*·*c*·*n*), *build*·*g* ++ *build*·*h*))
proof −
have ⋀*h*.
(Λ *c n*. *listfun*·*g*·*c*·(*listfun*·*h*·*c*·*n*), *build*·*g* ++ *build*·*h*) ∈ {(*f*, *xs*). *f* = *abstract-list xs*}
by (*simp add*: *fold/build*)
from *cont-Abs-listfun* [*OF this*, *of* λ*x*. *x*]
show *?thesis* **by** *simp*
qed

lemma *cont-listfun-comp-body2* [*simp*]:
cont (λ*g*. *Abs-listfun* (Λ *c n*. *listfun*·*g*·*c*·(*listfun*·*h*·*c*·*n*), *build*·*g* ++ *build*·*h*))
proof −
have ⋀*g*.

(Λ c n. listfun·g·c·(listfun·h·c·n), build·g ++ build·h) ∈ {(f, xs). f = abstract-list xs}
by (simp add: fold/build)
from cont-Abs-listfun [OF this, of λx. x]
show ?thesis by simp
qed

lemma *cont-listfun-comp-body* [*simp*]:
cont (λg. Λ h. *Abs-listfun* (Λ c n. *listfun·g·c·*(*listfun·h·c·n*), *build·g* ++ *build·h*))
by (*rule cont2cont-LAM*) *simp+*

lemma *abstract-list-build-append*:
abstract-list (*build·g* ++ *build·h*) = (Λ c n. *listfun·g·c·*(*listfun·h·c·n*))
by (*intro cfun-eqI*) (*simp add*: *fold/build*)

lemma *augment/build*:
augment·g·(*build·h*) = *build·*(*g* ∘*lf h*)
by (*simp add*: *listfun-comp-def augment-def build-Abs-listfun* [*OF abstract-list-build-append*])

lemma *augment/nil*:
augment·g·[] = *build·g*
by (*simp add*: *augment-def*)

lemma *build-listfun-comp* [*simp*]:
build·(*g* ∘*lf h*) = *build·g* ++ *build·h*
unfolding *augment/build* [*symmetric*]
by (*simp add*: *augment-def*)

lemma *augment-augment*:
augment·g·(*augment·h·xs*) = *augment·*(*g* ∘*lf h*)*·xs*
by (*simp add*: *augment-def*)

lemma *abstract-list-map* [*simp*]:
abstract-list (*map·f·xs*) = (Λ c n. *foldr·*(*mapFB·c·f*)*·n·xs*)
by (*intro cfun-eqI*, *induct xs*) *simp+*

lemma *map*:
map·f·xs = *build·*(*Abs-listfun* (Λ c n. *foldr·*(*mapFB·c·f*)*·n·xs*, *map·f·xs*))
by (*simp add*: *build-Abs-listfun*)

lemma *mapList*:
foldr·(*mapFB·*(:)*·f*)*·*[] = *map·f*
by (*rule cfun-eqI*, *rename-tac x*, *induct-tac x*) *simp+*

lemma *mapFB*:
mapFB·(*mapFB·c·f*)*·g* = *mapFB·c·*(*f oo g*)
by *simp*

lemma ++:

xs ++ ys = augment·(Abs-listfun (abstract-list xs, xs))·ys
by *simp*

15.3 Examples

fixrec *sum* :: [*Integer*] → *Integer* **where**
sum·xs = foldr·(+)·0·xs

fixrec *down′* :: *Integer* → (*Integer* → *'a* → *'a*) → *'a* → *'a* **where**
down′·v·c·n = If le·1·v then c·v·(down′·(v − 1)·c·n) else n
declare *down′.simps* [*simp del*]

lemma *down′-strict* [*simp*]: *down′·⊥ = ⊥* **by** (*fixrec-simp*)

definition *down* :: *'b itself* ⇒ *Integer* → [*Integer*] **where**
down C-type = (Λ v. build·(Abs-listfun (
(down′ :: *Integer* → (*Integer* → *'b* → *'b*) → *'b* → *'b*)·*v*,
down′·v·(:)·[])))

lemma *abstract-list-down′* [*simp*]:
abstract-list (down′·v·(:)·[]) = down′·v
proof (*intro cfun-eqI*)
fix *c* :: *Integer* → *'b* → *'b* **and** *n* :: *'b*
show (*abstract-list (down′·v·(:)·[]))·c·n = down′·v·c·n*
proof (*cases le·1·v*)
assume *le·1·v = ⊥* **then show** *?thesis* **by** *simp*
next
assume *le·1·v = FF* **then show** *?thesis*
by (*subst* (*1 2*) *down′.simps*) *simp*
next
have *le·(0*::*Integer)·1 = TT* **by** *simp*
moreover assume *le·1·v = TT*
ultimately have *le·0·v = TT* **by** (*rule le-trans*)
then show *?thesis*
by (*induct v rule*: *nonneg-Integer-induct*)
(*subst* (*1 2*) *down′.simps*, *simp*)+
qed
qed

lemma *cont-Abs-listfun-down′* [*simp*]:
cont (*λv. Abs-listfun* (*down′·v*, *down′·v·*(:)·[]))
proof −
have ⋀*v*. (*down′·v*, *down′·v·*(:)·[]) ∈ {(*f*, *xs*). *f = abstract-list xs*} **by** *simp*
from *cont-Abs-listfun* [*OF this*, *of id*] **show** *?thesis* **by** *simp*
qed

lemma *sum-down*:
sum·((down TYPE(*Integer*))·*v*) = *down′·v·*(+)·*0*
by (*simp add*: *down-def fold*/*build*)

end
theory *HLint*
imports
../HOLCF-Prelude
../List-Comprehension
begin

16 HLint

The tool `hlint` analyses Haskell code and, based on a data base of rewrite rules, suggests stylistic improvements to it. We verify a number of these rules using our implementation of the Haskell standard library.

16.1 Ord

```
x == a || x == b || x == c ==> x `elem` [a,b,c]
```

lemma (*eq*·(*x*::'*a*::*Eq-sym*)·*a orelse eq*·*x*·*b orelse eq*·*x*·*c*) = *elem*·*x*·[*a*, *b*, *c*]
by (*auto simp add*: *eq-sym*)

```
x /= a && x /= b && x /= c ==> x `notElem` [a,b,c]
```

lemma (*neq*·(*x*::'*a*::*Eq-sym*)·*a andalso neq*·*x*·*b andalso neq*·*x*·*c*) = *notElem*·*x*·[*a*, *b*, *c*]
by (*auto simp add*: *eq-sym*)

16.2 List

```
concat (map f x) ==> concatMap f x
```

lemma *concat*·(*map*·*f*·*x*) = *concatMap*·*f*·*x*
by (*auto simp add*: *concatMap-def*)

```
concat [a, b] ==> a ++ b
```

lemma *concat*·[*a*, *b*] = *a* ++ *b*
by *auto*

```
map f (map g x) ==> map (f . g) x
```

lemma *map*·*f*·(*map*·*g*·*x*) = *map*·(*f oo g*)·*x*
by *auto*

```
x !! 0 ==> head x
```

lemma *x* !! *0* = *head*·*x*
by (*cases x*) *auto*

```
take n (repeat x) ==> replicate n x
```

lemma *take*·*n*·(*repeat*·*x*) = *replicate*·*n*·*x*

by (*simp add*: *replicate-def*)

```
lemma "head\<cdot>(reverse\<cdot>x) = last\<cdot>x"
```

lemma *head*·(*reverse*·*x*) = *last*·*x*
proof (*cases finite-list x*)
case *True* **then show** *?thesis*
by (*induct x rule*: *reverse-induct*) (*auto simp add*: *last-append-singleton*)
next
case *False* **then show** *?thesis*
by (*simp add*: *last-spine-strict reverse-spine-strict*)
qed

```
head (drop n x) ==> x !! n where note = "if the index is non-negative"
```

lemma
assumes *le*·*0*·*n* ≠ *FF*
shows *head*·(*drop*·*n*·*x*) = *x* !! *n*
proof (*cases le*·*0*·*n*)
assume *le*·*0*·*n* = *FF* **with** *assms* **show** *?thesis*..
next
assume *le*·*0*·*n* = *TT*
then show *?thesis*
proof (*induction arbitrary*: *x rule*: *nonneg-Integer-induct*)
case *0*
show *?case* **by** (*cases x*) *auto*
next
case (*step i x*)
from *step.hyps* **have** [*simp*]:*le*·*i*·*0* = *FF* **by** (*cases i*, *auto simp add*: *one-Integer-def zero-Integer-def*)
from *step.hyps* **have** [*simp*]:*eq*·*i*·*0* = *FF* **by** (*cases i*, *auto simp add*: *one-Integer-def zero-Integer-def*)
show *?case*
using *step.IH* **by** (*cases x*)*auto*
qed
qed *simp*

```
reverse (tail (reverse x)) ==> init x
```

lemma *reverse*·(*tail*·(*reverse*·*x*)) ⊑ *init*·*x*
proof (*cases finite-list x*)
case *True*
then show *?thesis*
by (*induct x rule*: *reverse-induct*) (*auto simp add*: *init-append-singleton*)
next
case *False*
then show *?thesis*
by (*auto simp add*: *reverse-spine-strict*)
qed

```
take (length x - 1) x ==> init x
```

lemma

assumes $x \neq []$
shows $take{\cdot}(length{\cdot}x - 1){\cdot}x \sqsubseteq init{\cdot}x$
using *assms*
proof (*induct x*)
case (*Cons y ys*)
show *?case*
proof (*cases ys*)
note *IH* = *Cons*
case (*Cons z zs*)
show *?thesis*
using *IH*
by (*cases length·zs*)
(*auto simp*: *Cons one-Integer-def dest*: *length-ge-0*)
qed (*auto simp*: *one-Integer-def*)
qed *auto*

```
foldr (++) [] ==> concat
```

lemma *foldr-append-concat*:*foldr·append·*[] = *concat*
proof (*rule cfun-eqI*)
fix *xs* :: [['*a*]]
show *foldr·append·*[]·*xs* = *concat·xs*
by (*induct xs*) *auto*
qed

```
foldl (++) [] ==> concat
```

lemma *foldl·append·*[] ⊑ *concat*
proof (*rule cfun-belowI*)
fix *xs* :: [['*a*]]
show *foldl·append·*[]·*xs* ⊑ *concat·xs*
by (*cases finite-list xs*)
(*auto simp add*: *foldr-append-concat foldl-assoc-foldr foldl-spine-strict*)
qed

```
span (not . p) ==> break p
```

lemma *span·*(*neg oo p*) = *break·p*
by *simp*

```
break (not . p) ==> span p
```

lemma *break·*(*neg oo p*) = *span·p*
by (*unfold break.simps*) (*subst assoc-oo*, *simp*)

```
or (map p x) ==> any p x
```

lemma *the-or·*(*map·p·x*) = *any·p·x*
by *simp*

```
and (map p x) ==> all p x
```

lemma *the-and·*(*map·p·x*) = *all·p·x*
by *simp*

```
zipWith (,) ==> zip
```

lemma *zipWith*·⟨,⟩ = *zip*
by (*simp add*: *zip-def*)

```
zipWith3 (,,) ==> zip3
```

lemma *zipWith3*·⟨,,⟩ = *zip3*
by (*simp add*: *zip3-def*)

```
length x == 0 ==> null x where note = "increases laziness"
```

lemma *eq*·(*length*·*x*)·*0* ⊑ *null*·*x*
proof (*cases x*)
case (*Cons y ys*)
then show *?thesis*
by (*cases length*·*ys*)
(*auto dest*: *length-ge-0 simp*: *zero-Integer-def one-Integer-def*)
qed *simp*+

```
length x /= 0 ==> not (null x)
```

lemma *neq*·(*length*·*x*)·*0* ⊑ *neg*·(*null*·*x*)
proof (*cases x*)
case (*Cons y ys*)
then show *?thesis*
by (*cases length*·*ys*)
(*auto dest*: *length-ge-0 simp*: *zero-Integer-def one-Integer-def*)
qed *simp*+

```
map (uncurry f) (zip x y) ==> zipWith f x y
```

lemma *map*·(*uncurry*·*f*)·(*zip*·*x*·*y*) = *zipWith*·*f*·*x*·*y*
proof (*induct x arbitrary*: *y*)
case (*Cons x xs y*) **then show** *?case* **by** (*cases y*) *auto*
qed *auto*

```
map f (zip x y) ==> zipWith (curry f) x y where _ = isVar f
```

lemma *map*·*f*·(*zip*·*x*·*y*) = *zipWith*·(*curry*·*f*)·*x*·*y*
proof(*induct x arbitrary*: *y*)
case (*Cons x xs y*) **then show** *?case* **by** (*cases y*) *auto*
qed *auto*

```
not (elem x y) ==> notElem x y
```

lemma *neg*·(*elem*·*x*·*y*) = *notElem*·*x*·*y*
by (*induct y*) *auto*

```
foldr f z (map g x) ==> foldr (f . g) z x
```

lemma *foldr*·*f*·*z*·(*map*·*g*·*x*) = *foldr*·(*f oo g*)·*z*·*x*
by (*induct x*) *auto*

```
null (filter f x) ==> not (any f x)
```

lemma *null*·(*filter*·*f*·*x*) = *neg*·(*any*·*f*·*x*)
proof (*induct x*)
  case (*Cons x xs*)
  then show *?case* **by** (*cases f*·*x*) *auto*
qed *auto*

```
filter f x == [] ==> not (any f x)
```

lemma *eq*·(*filter*·*f*·*x*)·[] = *neg*·(*any*·*f*·*x*)
proof (*induct x*)
  case (*Cons x xs*)
  then show *?case* **by** (*cases f*·*x*) *auto*
qed *auto*

```
filter f x /= [] ==> any f x
```

lemma *neq*·(*filter*·*f*·*x*)·[] = *any*·*f*·*x*
proof (*induct x*)
  case (*Cons x xs*)
  then show *?case* **by** (*cases f*·*x*) *auto*
qed *auto*

```
any (== a) ==> elem a
```

lemma *any*·(Λ *z*. *eq*·*z*·*a*) = *elem*·*a*
proof (*rule cfun-eqI*)
  fix *xs*
  show *any*·(Λ *z*. *eq*·*z*·*a*)·*xs* = *elem*·*a*·*xs*
    by (*induct xs*) *auto*
qed

```
any ((==) a) ==> elem a
```

lemma *any*·(*eq*·(*a*::'*a*::*Eq-sym*)) = *elem*·*a*
proof (*rule cfun-eqI*)
  fix *xs*
  show *any*·(*eq*·*a*)·*xs* = *elem*·*a*·*xs*
    by (*induct xs*) (*auto simp*: *eq-sym*)
qed

```
any (a ==) ==> elem a
```

lemma *any*·(Λ *z*. *eq*·(*a*::'*a*::*Eq-sym*)·*z*) = *elem*·*a*
proof (*rule cfun-eqI*)
  fix *xs*
  show *any*·(Λ *z*. *eq*·*a*·*z*)·*xs* = *elem*·*a*·*xs*
    by (*induct xs*) (*auto simp*: *eq-sym*)
qed

```
all (/= a) ==> notElem a
```

lemma *all*·(Λ *z*. *neq*·*z*·(*a*::'*a*::*Eq-sym*)) = *notElem*·*a*
proof (*rule cfun-eqI*)
  fix *xs*

show *all*·(Λ *z*. *neq*·*z*·*a*)·*xs* = *notElem*·*a*·*xs*
by (*induct xs*) *auto*
qed

```
all (a /=) ==> notElem a
```

lemma *all*·(Λ *z*. *neq*·(*a*::'*a*::*Eq-sym*)·*z*) = *notElem*·*a*
proof (*rule cfun-eqI*)
fix *xs*
show *all*·(Λ *z*. *neq*·*a*·*z*)·*xs* = *notElem*·*a*·*xs*
by (*induct xs*) (*auto simp*: *eq-sym*)
qed

16.3 Folds

```
foldr  (&&) True ==> and
```

lemma *foldr*·*trand*·*TT* = *the-and*
by (*subst the-and.simps*, *rule*)

```
foldl  (&&) True ==> and
```

lemma *foldl-to-and*:*foldl*·*trand*·*TT* ⊑ *the-and*
proof (*rule cfun-belowI*)
fix *xs*
show *foldl*·*trand*·*TT*·*xs* ⊑ *the-and*·*xs*
by (*cases finite-list xs*) (*auto simp*: *foldl-assoc-foldr foldl-spine-strict*)
qed

```
foldr1 (&&)  ==> and
```

lemma *foldr1*·*trand* ⊑ *the-and*
proof (*rule cfun-belowI*)
fix *xs*
show *foldr1*·*trand*·*xs* ⊑ *the-and*·*xs*
proof (*induct xs*)
case (*Cons y ys*)
then show *?case* **by** (*cases ys*) (*auto elim*: *monofun-cfun-arg*)
qed *simp*+
qed

```
foldl1 (&&)  ==> and
```

lemma *foldl1*·*trand* ⊑ *the-and*
proof (*rule cfun-belowI*)
fix *x*
have *foldl1*·*trand*·*x* ⊑ *foldl*·*trand*·*TT*·*x*
by (*cases x*, *auto*)
also have ... ⊑ *the-and*·*x*
by (*rule monofun-cfun-fun*[*OF foldl-to-and*])
finally show *foldl1*·*trand*·*x* ⊑ *the-and*·*x* **.**
qed

```
foldr  (||) False ==> or
```

lemma *foldr·tror·FF = the-or*
by (*subst the-or.simps*, *rule*)

```
foldl  (||) False ==> or
```

lemma *foldl-to-or*: *foldl·tror·FF* $\sqsubseteq$ *the-or*
proof (*rule cfun-belowI*)
fix *xs*
show *foldl·tror·FF·xs* $\sqsubseteq$ *the-or·xs*
by (*cases finite-list xs*) (*auto simp*: *foldl-assoc-foldr foldl-spine-strict*)
qed

```
foldr1 (||)  ==> or
```

lemma *foldr1·tror* $\sqsubseteq$ *the-or*
proof (*rule cfun-belowI*)
fix *xs*
show *foldr1·tror·xs* $\sqsubseteq$ *the-or·xs*
proof (*induct xs*)
case (*Cons y ys*)
then show *?case* **by** (*cases ys*) (*auto elim*: *monofun-cfun-arg*)
qed *simp+*
qed

```
foldl1 (||)  ==> or
```

lemma *foldl1·tror* $\sqsubseteq$ *the-or*
proof(*rule cfun-belowI*)
fix *x*
have *foldl1·tror·x* $\sqsubseteq$ *foldl·tror·FF·x*
by (*cases x*, *auto*)
also have *...* $\sqsubseteq$ *the-or·x*
by (*rule monofun-cfun-fun*[*OF foldl-to-or*])
finally show *foldl1·tror·x* $\sqsubseteq$ *the-or·x* **.**
qed

16.4 Function

```
(\x -> x) ==> id
```

lemma (Λ *x*. *x*) = *ID*
by (*metis ID-def*)

```
(\x y -> x) ==> const
```

lemma (Λ *x y*. *x*) = *const*
by (*intro cfun-eqI*) *simp*

```
(\(x,y) -> y) ==> fst where _ = notIn x y
```

lemma (Λ $\langle$*x*, *y*$\rangle$. *x*) = *fst*
proof (*rule cfun-eqI*)

fix *p*
show (*case p of* ⟨*x*, *y*⟩ ⇒ *x*) = *fst* · *p*
proof (*cases p*)
case *bottom* **then show** *?thesis* **by** *simp*
next
case *Tuple2* **then show** *?thesis* **by** *simp*
qed
qed

```
(\(x,y) -> y) ==> snd where _ = notIn x y
```

lemma (Λ ⟨*x*, *y*⟩. *y*) = *snd*
proof (*rule cfun-eqI*)
fix *p*
show (*case p of* ⟨*x*, *y*⟩ ⇒ *y*) = *snd* · *p*
proof (*cases p*)
case *bottom* **then show** *?thesis* **by** *simp*
next
case *Tuple2* **then show** *?thesis* **by** *simp*
qed
qed

```
(\x y-> f (x,y)) ==> curry f where _ = notIn [x,y] f
```

lemma (Λ *x y*. *f*·⟨*x*, *y*⟩) = *curry*·*f*
by (*auto intro*!: *cfun-eqI*)

```
(\(x,y) -> f x y) ==> uncurry f where _ = notIn [x,y] f
```

lemma (Λ ⟨*x*, *y*⟩. *f*·*x*·*y*) ⊑ *uncurry*·*f*
by (*rule cfun-belowI*, *rename-tac x*, *case-tac x*, *auto*)

```
(\x -> y) ==> const y where _ = isAtom y && notIn x y
```

lemma (Λ *x*. *y*) = *const*·*y*
by (*intro cfun-eqI*) *simp*

lemma *flip*·*f*·*x*·*y* = *f*·*y*·*x* **by** *simp*

16.5 Bool

```
a == True ==> a
```

lemma *eq-true*:*eq*·*x*·*TT* = *x*
by (*cases x*, *auto*)

```
a == False ==> not a
```

lemma *eq-false*:*eq*·*x*·*FF* = *neg*·*x*
by (*cases x*, *auto*)

```
(if a then x else x) ==> x where note = "reduces strictness"
```

lemma *if-equal*:(*If a then x else x*) $\sqsubseteq$ *x*
by (*cases a*, *auto*)

```
(if a then True else False) ==> a
```

lemma (*If a then TT else FF*) = *a*
by (*cases a*, *auto*)

```
(if a then False else True) ==> not a
```

lemma (*If a then FF else TT*) = *neg*·*a*
by (*cases a*, *auto*)

```
(if a then t else (if b then t else f)) ==> if a || b then t else
f
```

lemma (*If a then t else* (*If b then t else f*)) = (*If a orelse b then t else f*)
by (*cases a*, *auto*)

```
(if a then (if b then t else f) else f) ==> if a && b then t else
f
```

lemma (*If a then* (*If b then t else f*) *else f*) = (*If a andalso b then t else f*)
by (*cases a*, *auto*)

```
(if x then True else y) ==> x || y where _ = notEq y False
```

lemma (*If x then TT else y*) = (*x orelse y*)
by (*cases x*, *auto*)

```
(if x then y else False) ==> x && y where _ = notEq y True
```

lemma (*If x then y else FF*) = (*x andalso y*)
by (*cases x*, *auto*)

```
(if c then (True, x) else (False, x)) ==> (c, x) where note = "reduces
strictness"
```

lemma (*If c then* ⟨*TT*, *x*⟩ *else* ⟨*FF*, *x*⟩) $\sqsubseteq$ ⟨*c*, *x*⟩
by (*cases c*, *auto*)

```
(if c then (False, x) else (True, x)) ==> (not c, x) where note
= "reduces strictness"
```

lemma (*If c then* ⟨*FF*, *x*⟩ *else* ⟨*TT*, *x*⟩) $\sqsubseteq$ ⟨*neg*·*c*, *x*⟩
by (*cases c*, *auto*)

```
or [x,y]  ==> x || y
```

lemma *the-or*·[*x*, *y*] = (*x orelse y*)
by (*fixrec-simp*)

```
or [x,y,z]  ==> x || y || z
```

lemma *the-or*·[*x*, *y*, *z*] = (*x orelse y orelse z*)
by (*fixrec-simp*)

```
and [x,y]  ==> x && y
```

lemma *the-and*·$[x,\ y] = (x\ \mathit{andalso}\ y)$
by (*fixrec-simp*)

```
and [x,y,z]  ==> x && y && z
```

lemma *the-and*·$[x,\ y,\ z] = (x\ \mathit{andalso}\ y\ \mathit{andalso}\ z)$
by (*fixrec-simp*)

16.6 Arrow

```
(fst x, snd x) ==>  x
```

lemma $x \sqsubseteq \langle \mathit{fst}{\cdot}x,\ \mathit{snd}{\cdot}x\rangle$
by (*cases* x, *auto*)

16.7 Seq

```
x `seq` x ==> x
```

lemma $\mathit{seq}{\cdot}x{\cdot}x = x$ **by** (*simp add*: *seq-def*)

16.8 Evaluate

```
True && x ==> x
```

lemma $(\mathit{TT}\ \mathit{andalso}\ x) = x$ **by** *auto*

```
False && x ==> False
```

lemma $(\mathit{FF}\ \mathit{andalso}\ x) = \mathit{FF}$ **by** *auto*

```
True || x ==> True
```

lemma $(\mathit{TT}\ \mathit{orelse}\ x) = \mathit{TT}$ **by** *auto*

```
False || x ==> x
```

lemma $(\mathit{FF}\ \mathit{orelse}\ x) = x$ **by** *auto*

```
not True ==> False
```

lemma $\mathit{neg}{\cdot}\mathit{TT} = \mathit{FF}$ **by** *auto*

```
not False ==> True
```

lemma $\mathit{neg}{\cdot}\mathit{FF} = \mathit{TT}$ **by** *auto*

```
fst (x,y) ==> x
```

lemma $\mathit{fst}{\cdot}\langle x,\ y\rangle = x$ **by** *auto*

```
snd (x,y) ==> y
```

lemma $\mathit{snd}{\cdot}\langle x,\ y\rangle = y$ **by** *auto*

```
f (fst p) (snd p) ==> uncurry f p
```

lemma $f{\cdot}(fst{\cdot}p){\cdot}(snd{\cdot}p) = uncurry{\cdot}f{\cdot}p$
by (*cases p*, *auto*)

```
init [x] ==> []
```

lemma $init{\cdot}[x] = []$ **by** *auto*

```
null [] ==> True
```

lemma $null{\cdot}[] = TT$ **by** *auto*

```
length [] ==> 0
```

lemma $length{\cdot}[] = 0$ **by** *auto*

```
foldl f z [] ==> z
```

lemma $foldl{\cdot}f{\cdot}z{\cdot}[] = z$ **by** *simp*

```
foldr f z [] ==> z
```

lemma $foldr{\cdot}f{\cdot}z{\cdot}[] = z$ **by** *auto*

```
foldr1 f [x] ==> x
```

lemma $foldr1{\cdot}f{\cdot}[x] = x$ **by** *simp*

```
scanr f z [] ==> [z]
```

lemma $scanr{\cdot}f{\cdot}z{\cdot}[] = [z]$ **by** *simp*

```
scanr1 f [] ==> []
```

lemma $scanr1{\cdot}f{\cdot}[] = []$ **by** *simp*

```
scanr1 f [x] ==> [x]
```

lemma $scanr1{\cdot}f{\cdot}[x] = [x]$ **by** *simp*

```
take n [] ==> []
```

lemma $take{\cdot}n{\cdot}[] \sqsubseteq []$ **by** (*cases n*, *auto*)

```
drop n [] ==> []
```

lemma $drop{\cdot}n{\cdot}[] \sqsubseteq []$
by (*subst drop.simps*) (*auto simp*: *if-equal*)

```
takeWhile p [] ==> []
```

lemma $takeWhile{\cdot}p{\cdot}[] = []$ **by** (*fixrec-simp*)

```
dropWhile p [] ==> []
```

lemma $dropWhile{\cdot}p{\cdot}[] = []$ **by** (*fixrec-simp*)

```
span p [] ==> ([],[])
```

lemma *span·p·[] = ⟨[], []⟩* **by** (*fixrec-simp*)

```
concat [a] ==> a
```

lemma *concat·[a] = a* **by** *auto*

```
concat [] ==> []
```

lemma *concat·[] = []* **by** *auto*

```
zip [] [] ==> []
```

lemma *zip·[]·[] = []* **by** *auto*

```
id x ==> x
```

lemma *ID·x = x* **by** *auto*

```
const x y ==> x
```

lemma *const·x·y = x* **by** *simp*

16.9 Complex hints

```
take (length t) s == t ==> t `Data.List.isPrefixOf` s
```

lemma
fixes *t* :: ['*a*::*Eq-sym*]
shows *eq·(take·(length·t)·s)·t ⊑ isPrefixOf·t·s*
by (*subst eq-sym*) (*rule eq-take-length-isPrefixOf*)

```
(take i s == t) ==> _eval_ ((i >= length t) && (t `Data.List.isPrefixOf`
s))
```

The hint is not true in general, as the following two lemmas show:

lemma
assumes *t = []* **and** *s = x : xs* **and** *i = 1*
shows *¬ (eq·(take·i·s)·t ⊑ (le·(length·t)·i andalso isPrefixOf·t·s))*
using *assms* **by** *simp*

lemma
assumes *le·0·i = TT* **and** *le·i·0 = FF*
and *s = ⊥* **and** *t = []*
shows *¬ ((le·(length·t)·i andalso isPrefixOf·t·s) ⊑ eq·(take·i·s)·t)*
using *assms* **by** (*subst take.simps*) *simp*

lemma *neg·(eq·a·b) = neq·a·b* **by** *auto*

```
not (a /= b) ==> a == b
```

lemma $neg{\cdot}(neq{\cdot}a{\cdot}b) = eq{\cdot}a{\cdot}b$ **by** *auto*

```
map id ==> id
```

lemma *map-id*:$map{\cdot}ID = ID$ **by** (*auto simp add*: *cfun-eq-iff*)

```
x == [] ==> null x
```

lemma $eq{\cdot}x{\cdot}[] = null{\cdot}x$ **by** (*cases x*, *auto*)

```
any id ==> or
```

lemma $any{\cdot}ID =$ *the-or* **by** (*auto simp add*:*map-id*)

```
all id ==> and
```

lemma $all{\cdot}ID =$ *the-and* **by** (*auto simp add*:*map-id*)

```
(if x then False else y) ==> (not x && y)
```

lemma (*If x then FF else y*) = (*neg*·*x andalso y*) **by** (*cases x*, *auto*)

```
(if x then y else True) ==> (not x || y)
```

lemma (*If x then y else TT*) = (*neg*·*x orelse y*) **by** (*cases x*, *auto*)

```
not (not x) ==> x
```

lemma $neg{\cdot}(neg{\cdot}x) = x$ **by** *auto*

```
(if c then f x else f y) ==> f (if c then x else y)
```

lemma (*If c then f*·*x else f*·*y*) $\sqsubseteq$ *f*·(*If c then x else y*) **by** (*cases c*, *auto*)

```
(\ x -> [x]) ==> (: [])
```

lemma (Λ *x*. [*x*]) = (Λ *z*. *z* : []) **by** *auto*

```
True == a ==> a
```

lemma $eq{\cdot}TT{\cdot}a = a$ **by** (*cases a*, *auto*)

```
False == a ==> not a
```

lemma $eq{\cdot}FF{\cdot}a = neg{\cdot}a$ **by** (*cases a*, *auto*)

```
a /= True ==> not a
```

lemma $neq{\cdot}a{\cdot}TT = neg{\cdot}a$ **by** (*cases a*, *auto*)

```
a /= False ==> a
```

lemma $neq{\cdot}a{\cdot}FF = a$ **by** (*cases a*, *auto*)

```
True /= a ==> not a
```

lemma $neq{\cdot}TT{\cdot}a = neg{\cdot}a$ **by** (*cases a*, *auto*)

```
False /= a ==> a
```

lemma *neq·FF·a = a* **by** (*cases a*, *auto*)

```
not (isNothing x) ==> isJust x
```

lemma *neg·(isNothing·x) = isJust·x* **by** *auto*

```
not (isJust x) ==> isNothing x
```

lemma *neg·(isJust·x) = isNothing·x* **by** *auto*

```
x == Nothing ==> isNothing x
```

lemma *eq·x·Nothing = isNothing·x* **by** (*cases x*, *auto*)

```
Nothing == x ==> isNothing x
```

lemma *eq·Nothing·x = isNothing·x* **by** (*cases x*, *auto*)

```
x /= Nothing ==> Data.Maybe.isJust x
```

lemma *neq·x·Nothing = isJust·x* **by** (*cases x*, *auto*)

```
Nothing /= x ==> Data.Maybe.isJust x
```

lemma *neq·Nothing·x = isJust·x* **by** (*cases x*, *auto*)

```
(if isNothing x then y else fromJust x) ==> fromMaybe y x
```

lemma (*If isNothing·x then y else fromJust·x*) = *fromMaybe·y·x* **by** (*cases x*, *auto*)

```
(if isJust x then fromJust x else y) ==> fromMaybe y x
```

lemma (*If isJust·x then fromJust·x else y*) = *fromMaybe·y·x* **by** (*cases x*, *auto*)

```
(isJust x && (fromJust x == y)) ==> x == Just y
```

lemma (*isJust·x andalso* (*eq·*(*fromJust·x*)*·y*)) = *eq·x·*(*Just·y*) **by** (*cases x*, *auto*)

```
elem True ==> or
```

lemma *elem·TT = the-or*
proof (*rule cfun-eqI*)
  fix *xs*
  show *elem·TT·xs = the-or·xs*
    by (*induct xs*) (*auto simp*: *eq-true*)
qed

```
notElem False ==> and
```

lemma *notElem·FF = the-and*
proof (*rule cfun-eqI*)
  fix *xs*
  show *notElem·FF·xs = the-and·xs*
    by (*induct xs*) (*auto simp*: *eq-false*)
qed

```
all ((/=) a) ==> notElem a
```

lemma *all*·(*neq*·(*a*::'*a*::*Eq-sym*)) = *notElem*·*a*
proof (*rule cfun-eqI*)
  fix *xs*
  show *all*·(*neq*·*a*)·*xs* = *notElem*·*a*·*xs*
    by (*induct xs*) (*auto simp*: *eq-sym*)
qed

```
maybe x id ==> Data.Maybe.fromMaybe x
```

lemma *maybe*·*x*·*ID* = *fromMaybe*·*x*
proof (*rule cfun-eqI*)
  fix *xs*
  show *maybe*·*x*·*ID*·*xs* = *fromMaybe*·*x*·*xs*
    by (*cases xs*) *auto*
qed

```
maybe False (const True) ==> Data.Maybe.isJust
```

lemma *maybe*·*FF*·(*const*·*TT*) = *isJust*
proof (*rule cfun-eqI*)
  fix *x* :: '*a Maybe*
  show *maybe*·*FF*·(*const*·*TT*)·*x* = *isJust*·*x*
    by (*cases x*) *simp*+
qed

```
maybe True (const False) ==> Data.Maybe.isNothing
```

lemma *maybe*·*TT*·(*const*·*FF*) = *isNothing*
proof (*rule cfun-eqI*)
  fix *x* :: '*a Maybe*
  show *maybe*·*TT*·(*const*·*FF*)·*x* = *isNothing*·*x*
    by (*cases x*) *simp*+
qed

```
maybe [] (: []) ==> maybeToList
```

lemma *maybe*·[]·(Λ *z*. *z* : []) = *maybeToList*
proof (*rule cfun-eqI*)
  fix *x* :: '*a Maybe*
  show *maybe*·[]·(Λ *z*. *z* : [])·*x* = *maybeToList*·*x*
    by (*cases x*) *simp*+
qed

```
catMaybes (map f x) ==> mapMaybe f x
```

lemma *catMaybes*·(*map*·*f*·*x*) = *mapMaybe*·*f*·*x* **by** *auto*

```
(if isNothing x then y else f (fromJust x)) ==> maybe y f x
```

lemma (*If isNothing*·*x then y else f*·(*fromJust*·*x*)) = *maybe*·*y*·*f*·*x* **by** (*cases x*, *auto*)

```
(if isJust x then f (fromJust x) else y) ==> maybe y f x
```

lemma (*If isJust*·*x then f*·(*fromJust*·*x*) *else y*) = *maybe*·*y*·*f*·*x* **by** (*cases x*, *auto*)

```
(map fromJust . filter isJust) ==> Data.Maybe.catMaybes
```

lemma (*map·fromJust oo filter·isJust*) = *catMaybes*
proof (*rule cfun-eqI*)
  fix *xs* :: ['*a Maybe*]
  show (*map·fromJust oo filter·isJust*)·*xs* = *catMaybes·xs*
  proof (*induct xs*)
    case (*Cons y ys*)
    then show *?case* **by** (*cases y*) *simp+*
  qed *simp+*
qed

```
concatMap (maybeToList . f) ==> Data.Maybe.mapMaybe f
```

lemma *concatMap*·(*maybeToList oo f*) = *mapMaybe·f*
proof (*rule cfun-eqI*)
  fix *xs*
  show *concatMap*·(*maybeToList oo f*)·*xs* = *mapMaybe·f·xs*
    by (*induct xs*) *auto*
qed

```
concatMap maybeToList ==> catMaybes
```

lemma *concatMap·maybeToList* = *catMaybes* **by** *auto*

```
mapMaybe f (map g x) ==> mapMaybe (f . g) x
```

lemma *mapMaybe·f·*(*map·g·x*) = *mapMaybe·*(*f oo g*)·*x* **by** *auto*

```
(($) . f) ==> f
```

lemma (*dollar oo f*) = *f* **by** (*auto simp add*:*cfun-eq-iff*)

```
(f $) ==> f
```

lemma (Λ *z*. *dollar·f·z*) = *f* **by** (*auto simp add*:*cfun-eq-iff*)

```
(\ a b -> g (f a) (f b)) ==> g `Data.Function.on` f
```

lemma (Λ *a b*. *g*·(*f·a*)·(*f·b*)) = *on·g·f* **by** (*auto simp add*:*cfun-eq-iff*)

```
id $! x ==> x
```

lemma *dollarBang·ID·x* = *x* **by** (*auto simp add*:*seq-def*)

```
[x | x <- y] ==> y
```

lemma [*x* | *x* <− *y*] = *y* **by** (*induct y*, *auto*)

```
isPrefixOf (reverse x) (reverse y) ==> isSuffixOf x y
```

lemma *isPrefixOf*·(*reverse·x*)·(*reverse·y*) = *isSuffixOf·x·y* **by** *auto*

```
concat (intersperse x y) ==> intercalate x y
```

lemma *concat*·(*intersperse·x·y*) = *intercalate·x·y* **by** *auto*

```
x `seq` y ==> y
```

lemma
  assumes $x \neq \bot$ **shows** $seq{\cdot}x{\cdot}y = y$
  using *assms* **by** (*simp add*: *seq-def*)

```
f $! x ==> f x
```

lemma assumes $x \neq \bot$ **shows** $dollarBang{\cdot}f{\cdot}x = f{\cdot}x$
  using *assms* **by** (*simp add*: *seq-def*)

```
maybe (f x) (f . g) ==> (f . maybe x g)
```

lemma $maybe{\cdot}(f{\cdot}x){\cdot}(f\ oo\ g) \sqsubseteq (f\ oo\ maybe{\cdot}x{\cdot}g)$
proof (*rule cfun-belowI*)
  fix y
  show $maybe{\cdot}(f{\cdot}x){\cdot}(f\ oo\ g){\cdot}y \sqsubseteq (f\ oo\ maybe{\cdot}x{\cdot}g){\cdot}y$
    by (*cases y*) *auto*
qed

end

Acknowledgments

We thank Lars Hupel for his help with the final AFP submission.

References

[1] J. Breitner, B. Huffman, N. Mitchell, and C. Sternagel. Certified HLints with Isabelle/HOLCF-Prelude, June 2013. Haskell And Rewriting Techniques (HART).

[2] S. Peyton Jones. Haskell 98 - Standard Prelude. *Journal of Functional Programming*, 13(1):103–124, 2003. doi:10.1017/S0956796803001011.

[3] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimization technique in GHC. In *the ACM SIGPLAN Haskell Workshop, Haskell'01*, pages 203–233, 2001.