Isabelle/HOLCF-Prelude

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

October 11, 2017

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
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 (*op* :) **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* :: $'a \to 'a \to tr$

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* $\to 'a \to tr$ **where**
  $neq{\cdot}x{\cdot}y = neg{\cdot}(eq{\cdot}x{\cdot}y)$

class *Eq-strict* = *Eq* +
  assumes *eq-strict* [*simp*]:
    $eq{\cdot}x{\cdot}\bot = \bot$
    $eq{\cdot}\bot{\cdot}y = \bot$

class *Eq-sym* = *Eq-strict* +
  assumes *eq-sym*: $eq{\cdot}x{\cdot}y = eq{\cdot}y{\cdot}x$

class *Eq-equiv* = *Eq-sym* +
  assumes *eq-self-neq-FF* [*simp*]: $eq{\cdot}x{\cdot}x \neq FF$
    and *eq-trans*: $eq{\cdot}x{\cdot}y = TT \Longrightarrow eq{\cdot}y{\cdot}z = TT \Longrightarrow eq{\cdot}x{\cdot}z = TT$
begin

lemma *eq-refl*: $eq{\cdot}x{\cdot}x \neq \bot \Longrightarrow eq{\cdot}x{\cdot}x = TT$
  ⟨*proof*⟩

end

class *Eq-eq* = *Eq-sym* +
  assumes *eq-self-neq-FF'*: $eq{\cdot}x{\cdot}x \neq FF$
    and *eq-TT-dest*: $eq{\cdot}x{\cdot}y = TT \Longrightarrow x = y$
begin

subclass *Eq-equiv*
  ⟨*proof*⟩

lemma *eqD* [*dest*]:
$eq{\cdot}x{\cdot}y = TT \Longrightarrow x = y$
$eq{\cdot}x{\cdot}y = FF \Longrightarrow x \neq y$
⟨*proof*⟩

end

2.1.1 Class instances

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

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

instance
⟨*proof*⟩

end

lemma *eq-ONE-ONE* [*simp*]: $eq{\cdot}ONE{\cdot}ONE = TT$
⟨*proof*⟩

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*]:
$oppOrdering{\cdot}LT = GT$
$oppOrdering{\cdot}EQ = EQ$
$oppOrdering{\cdot}GT = LT$
$oppOrdering{\cdot}\bot = \bot$
⟨*proof*⟩

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*: $lt{\cdot}x{\cdot}y = TT \longleftrightarrow compare{\cdot}x{\cdot}y = LT$
⟨*proof*⟩

end

class *Ord-strict* = *Ord* +
  assumes *compare-strict* [*simp*]:
    $compare{\cdot}\bot{\cdot}y = \bot$
    $compare{\cdot}x{\cdot}\bot = \bot$
begin

lemma *lt-strict* [*simp*]:
  shows $lt{\cdot}\bot{\cdot}x = \bot$
    and $lt{\cdot}x{\cdot}\bot = \bot$
  ⟨*proof*⟩

lemma *le-strict* [*simp*]:
  shows $le{\cdot}\bot{\cdot}x = \bot$
    and $le{\cdot}x{\cdot}\bot = \bot$
  ⟨*proof*⟩

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*: $eq{\cdot}x{\cdot}y = is\text{-}EQ{\cdot}(compare{\cdot}x{\cdot}y)$
    and *oppOrdering-compare* [*simp*]:
    $oppOrdering{\cdot}(compare{\cdot}x{\cdot}y) = compare{\cdot}y{\cdot}x$
    and *compare-EQ-dest*: $compare{\cdot}x{\cdot}y = EQ \Longrightarrow x = y$
    and *compare-self-below-EQ*: $compare{\cdot}x{\cdot}x \sqsubseteq EQ$
    and *compare-LT-trans*:
    $compare{\cdot}x{\cdot}y = LT \Longrightarrow compare{\cdot}y{\cdot}z = LT \Longrightarrow compare{\cdot}x{\cdot}z = LT$

begin

lemma *eq-TT-dest*: $eq{\cdot}x{\cdot}y = TT \Longrightarrow x = y$
  ⟨*proof*⟩

lemma *le-iff-lt-or-eq*:
  $le{\cdot}x{\cdot}y = TT \longleftrightarrow lt{\cdot}x{\cdot}y = TT \vee eq{\cdot}x{\cdot}y = TT$
  ⟨*proof*⟩

lemma *compare-sym*:
  $compare{\cdot}x{\cdot}y = (case\ compare{\cdot}y{\cdot}x\ of\ LT \Rightarrow GT \mid EQ \Rightarrow EQ \mid GT \Rightarrow LT)$
  ⟨*proof*⟩

lemma *compare-self-neq-LT* [*simp*]: $compare{\cdot}x{\cdot}x \neq LT$
  ⟨*proof*⟩

lemma *compare-self-neq-GT* [*simp*]: $compare{\cdot}x{\cdot}x \neq GT$

⟨*proof*⟩

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

lemma *lt-trans*: $lt{\cdot}x{\cdot}y = TT \Longrightarrow lt{\cdot}y{\cdot}z = TT \Longrightarrow lt{\cdot}x{\cdot}z = TT$
⟨*proof*⟩

lemma *compare-GT-iff-LT*: $compare{\cdot}x{\cdot}y = GT \longleftrightarrow compare{\cdot}y{\cdot}x = LT$
⟨*proof*⟩

lemma *compare-GT-trans*:
$compare{\cdot}x{\cdot}y = GT \Longrightarrow compare{\cdot}y{\cdot}z = GT \Longrightarrow compare{\cdot}x{\cdot}z = GT$
⟨*proof*⟩

lemma *compare-EQ-iff-eq-TT*:
$compare{\cdot}x{\cdot}y = EQ \longleftrightarrow eq{\cdot}x{\cdot}y = TT$
⟨*proof*⟩

lemma *compare-EQ-trans*:
$compare{\cdot}x{\cdot}y = EQ \Longrightarrow compare{\cdot}y{\cdot}z = EQ \Longrightarrow compare{\cdot}x{\cdot}z = EQ$
⟨*proof*⟩

lemma *le-trans*:
$le{\cdot}x{\cdot}y = TT \Longrightarrow le{\cdot}y{\cdot}z = TT \Longrightarrow le{\cdot}x{\cdot}z = TT$
⟨*proof*⟩

lemma *neg-lt*: $neg{\cdot}(lt{\cdot}x{\cdot}y) = le{\cdot}y{\cdot}x$
⟨*proof*⟩

lemma *neg-le*: $neg{\cdot}(le{\cdot}x{\cdot}y) = lt{\cdot}y{\cdot}x$
⟨*proof*⟩

subclass *Eq-eq*
⟨*proof*⟩

end

A combinator for defining Ord instances for datatypes.

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

lemma *thenOrdering-simps* [*simp*]:
$thenOrdering{\cdot}LT{\cdot}y = LT$
$thenOrdering{\cdot}EQ{\cdot}y = y$
$thenOrdering{\cdot}GT{\cdot}y = GT$
$thenOrdering{\cdot}\bot{\cdot}y = \bot$
⟨*proof*⟩

lemma *thenOrdering-LT-iff* [*simp*]:

thenOrdering·*x*·*y* = *LT* ⟷ *x* = *LT* ∨ *x* = *EQ* ∧ *y* = *LT*
⟨*proof*⟩

lemma *thenOrdering-EQ-iff* [*simp*]:
thenOrdering·*x*·*y* = *EQ* ⟷ *x* = *EQ* ∧ *y* = *EQ*
⟨*proof*⟩

lemma *thenOrdering-GT-iff* [*simp*]:
thenOrdering·*x*·*y* = *GT* ⟷ *x* = *GT* ∨ *x* = *EQ* ∧ *y* = *GT*
⟨*proof*⟩

lemma *thenOrdering-below-EQ-iff* [*simp*]:
thenOrdering·*x*·*y* ⊑ *EQ* ⟷ *x* ⊑ *EQ* ∧ (*x* = ⊥ ∨ *y* ⊑ *EQ*)
⟨*proof*⟩

lemma *is-EQ-thenOrdering* [*simp*]:
is-EQ·(*thenOrdering*·*x*·*y*) = (*is-EQ*·*x andalso is-EQ*·*y*)
⟨*proof*⟩

lemma *oppOrdering-thenOrdering*:
oppOrdering·(*thenOrdering*·*x*·*y*) =
thenOrdering·(*oppOrdering*·*x*)·(*oppOrdering*·*y*)
⟨*proof*⟩

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*⟩

end

lemma *lt-le*:
lt·(*x*::′*a*::*Ord-linear*)·*y* = (*le*·*x*·*y andalso neq*·*x*·*y*)
⟨*proof*⟩

end

3 Cpo for Numerals

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

class *plus-cpo* = *plus* + *cpo* +
assumes *cont-plus1*: *cont* (λ*x*::′*a*::{*plus*,*cpo*}. *x* + *y*)

assumes *cont-plus2*: *cont* ($\lambda y{::}'a{::}\{plus,cpo\}.\ x + y$)
begin

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

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* :: $'a \Rightarrow 'a \to 'a$ ($'(\text{--}')$) **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* ($\lambda x.\ f\ x + g\ x :: 'a{::}plus\text{-}cpo$)
⟨*proof*⟩

lemma *cont2cont-minus* [*simp*, *cont2cont*]:
assumes *cont* ($\lambda x.\ f\ x$) **and** *cont* ($\lambda x.\ g\ x$)
shows *cont* ($\lambda x.\ f\ x - g\ x :: 'a{::}minus\text{-}cpo$)
⟨*proof*⟩

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*)
⟨*proof*⟩

instantiation *u* :: ({*zero*,*cpo*}) *zero*
begin
definition *zero-u* = *up*·(*0*::′*a*)
instance ⟨*proof*⟩
end

instantiation *u* :: ({*one*,*cpo*}) *one*
begin
definition *one-u* = *up*·(*1*::′*a*)
instance ⟨*proof*⟩
end

instantiation *u* :: (*plus-cpo*) *plus*
begin
definition *plus-u x y* = (Λ(*up*·*a*) (*up*·*b*). *up*·(*a* + *b*))·*x*·*y*
instance ⟨*proof*⟩
end

instantiation *u* :: (*minus-cpo*) *minus*
begin
definition *minus-u x y* = (Λ(*up*·*a*) (*up*·*b*). *up*·(*a* − *b*))·*x*·*y*
instance ⟨*proof*⟩
end

instantiation *u* :: (*times-cpo*) *times*
begin
definition *times-u x y* = (Λ(*up*·*a*) (*up*·*b*). *up*·(*a* ∗ *b*))·*x*·*y*
instance ⟨*proof*⟩
end

lemma *plus-u-strict* [*simp*]:
fixes *x* :: - *u* **shows** *x* + ⊥ = ⊥ **and** ⊥ + *x* = ⊥
⟨*proof*⟩

lemma *minus-u-strict* [*simp*]:
fixes *x* :: - *u* **shows** *x* − ⊥ = ⊥ **and** ⊥ − *x* = ⊥
⟨*proof*⟩

lemma *times-u-strict* [*simp*]:
fixes *x* :: - *u* **shows** *x* ∗ ⊥ = ⊥ **and** ⊥ ∗ *x* = ⊥
⟨*proof*⟩

lemma *plus-up-up* [*simp*]: *up*·*x* + *up*·*y* = *up*·(*x* + *y*)
⟨*proof*⟩

lemma *minus-up-up* [*simp*]: $up{\cdot}x - up{\cdot}y = up{\cdot}(x - y)$
⟨*proof*⟩

lemma *times-up-up* [*simp*]: $up{\cdot}x * up{\cdot}y = up{\cdot}(x * y)$
⟨*proof*⟩

instance u :: (*plus-cpo*) *plus-cpo*
⟨*proof*⟩

instance u :: (*minus-cpo*) *minus-cpo*
⟨*proof*⟩

instance u :: (*times-cpo*) *times-cpo*
⟨*proof*⟩

instance u :: ({*semigroup-add*,*plus-cpo*}) *semigroup-add*
⟨*proof*⟩

instance u :: ({*ab-semigroup-add*,*plus-cpo*}) *ab-semigroup-add*
⟨*proof*⟩

instance u :: ({*monoid-add*,*plus-cpo*}) *monoid-add*
⟨*proof*⟩

instance u :: ({*comm-monoid-add*,*plus-cpo*}) *comm-monoid-add*
⟨*proof*⟩

instance u :: ({*numeral*, *plus-cpo*}) *numeral* ⟨*proof*⟩

instance *int* :: *plus-cpo*
⟨*proof*⟩

instance *int* :: *minus-cpo*
⟨*proof*⟩

end

4 Data: Functions

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

fixrec *flip* :: $('a \to 'b \to 'c) \to 'b \to 'a \to 'c$ **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 -> 'b) -> 'a -> 'b **where**
  dollar·*f*·*x* = *f*·*x*

fixrec *dollarBang* :: ('a -> 'b) -> 'a -> 'b **where**
  dollarBang·*f*·*x* = *seq*·*x*·(*f*·*x*)

fixrec *on* :: ('b -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'a -> 'c **where**
  on·*g*·*f*·*x*·*y* = *g*·(*f*·*x*)·(*f*·*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*]:
  $(x = \bot \Longrightarrow y = \bot) \Longrightarrow (y = \bot \Longrightarrow x = \bot) \Longrightarrow (x = TT \Longrightarrow y = TT) \Longrightarrow (y = TT \Longrightarrow x = TT) \Longrightarrow x = y$
⟨*proof*⟩

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*
  ⟨*proof*⟩

Ord

lemma *compare-tr-simps* [*simp*]:
  compare·*FF*·*FF* = *EQ*
  compare·*FF*·*TT* = *LT*
  compare·*TT*·*FF* = *GT*
  compare·*TT*·*TT* = *EQ*
  ⟨*proof*⟩

5.2 Lemmas

lemma *andalso-eq-TT-iff* [*simp*]:
  (*x andalso y*) = *TT* ⟷ *x* = *TT* ∧ *y* = *TT*
  ⟨*proof*⟩

lemma *andalso-eq-FF-iff* [*simp*]:
  (*x andalso y*) = *FF* ⟷ *x* = *FF* ∨ (*x* = *TT* ∧ *y* = *FF*)
  ⟨*proof*⟩

lemma *andalso-eq-bottom-iff* [*simp*]:
$(x\ andalso\ y) = \bot \longleftrightarrow x = \bot \vee (x = TT \wedge y = \bot)$
⟨*proof*⟩

lemma *orelse-eq-FF-iff* [*simp*]:
$(x\ orelse\ y) = FF \longleftrightarrow x = FF \wedge y = FF$
⟨*proof*⟩

lemma *orelse-assoc* [*simp*]:
$((x\ orelse\ y)\ orelse\ z) = (x\ orelse\ y\ orelse\ z)$
⟨*proof*⟩

lemma *andalso-assoc* [*simp*]:
$((x\ andalso\ y)\ andalso\ z) = (x\ andalso\ y\ andalso\ z)$
⟨*proof*⟩

lemma *neg-orelse* [*simp*]:
$neg{\cdot}(x\ orelse\ y) = (neg{\cdot}x\ andalso\ neg{\cdot}y)$
⟨*proof*⟩

lemma *neg-andalso* [*simp*]:
$neg{\cdot}(x\ andalso\ y) = (neg{\cdot}x\ orelse\ neg{\cdot}y)$
⟨*proof*⟩

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

lemma *neg-eq-simps*:
$neg{\cdot}x = TT \Longrightarrow x = FF$
$neg{\cdot}x = FF \Longrightarrow x = TT$
$neg{\cdot}x = \bot \Longrightarrow x = \bot$
⟨*proof*⟩

lemma *neg-eq-TT-iff* [*simp*]: $neg{\cdot}x = TT \longleftrightarrow x = FF$
⟨*proof*⟩

lemma *neg-eq-FF-iff* [*simp*]: $neg{\cdot}x = FF \longleftrightarrow x = TT$
⟨*proof*⟩

lemma *neg-eq-bottom-iff* [*simp*]: $neg{\cdot}x = \bot \longleftrightarrow x = \bot$
⟨*proof*⟩

lemma *neg-eq* [*simp*]:
$neg{\cdot}x = neg{\cdot}y \longleftrightarrow x = y$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *neg-comp-neg* [*simp*]:
neg oo neg = *ID*
⟨*proof*⟩

end

6 Data: Tuple

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

6.1 Datatype definitions

domain *Unit* (⟨⟩) = *Unit* (⟨⟩)

domain ('*a*, '*b*) *Tuple2* (⟨-, -⟩) =
Tuple2 (**lazy** *fst* :: '*a*) (**lazy** *snd* :: '*b*) (⟨-, -⟩)

notation *Tuple2* (⟨,⟩)

fixrec *uncurry* :: ('*a* → '*b* → '*c*) → ⟨'*a*, '*b*⟩ → '*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
⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

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

instance *Tuple2* :: (*Eq-sym*, *Eq-sym*) *Eq-sym*
⟨*proof*⟩

instance *Tuple2* :: (*Eq-equiv*, *Eq-equiv*) *Eq-equiv*
⟨*proof*⟩

instance *Tuple2* :: (*Eq-eq*, *Eq-eq*) *Eq-eq*
⟨*proof*⟩

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

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

instance
⟨*proof*⟩

end

lemma *compare-Tuple2-simps* [*simp*]:
compare·⟨*x1*, *y1*⟩·⟨*x2*, *y2*⟩ = *thenOrdering*·(*compare*·*x1*·*x2*)·(*compare*·*y1*·*y2*)
⟨*proof*⟩

instance *Tuple2* :: (*Ord-linear*, *Ord-linear*) *Ord-linear*
⟨*proof*⟩

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*⟩

end

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

instance *Tuple3* :: (*Eq-sym*, *Eq-sym*, *Eq-sym*) *Eq-sym*
⟨*proof*⟩

instance *Tuple3* :: (*Eq-equiv*, *Eq-equiv*, *Eq-equiv*) *Eq-equiv*
⟨*proof*⟩

instance *Tuple3* :: (*Eq-eq*, *Eq-eq*, *Eq-eq*) *Eq-eq*
⟨*proof*⟩

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
⟨*proof*⟩

end

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

instance *Tuple3* :: (*Ord-linear*, *Ord-linear*, *Ord-linear*) *Ord-linear*
⟨*proof*⟩

end

7 Data: Integers

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

begin

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

instance *Integer* :: *flat*
⟨*proof*⟩

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

definition *0* = *MkI*·*0*
definition *1* = *MkI*·*1*
definition $a + b = (\Lambda\ (MkI{\cdot}x)\ (MkI{\cdot}y).\ MkI{\cdot}(x + y)){\cdot}a{\cdot}b$
definition $a - b = (\Lambda\ (MkI{\cdot}x)\ (MkI{\cdot}y).\ MkI{\cdot}(x - y)){\cdot}a{\cdot}b$
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 ⟨*proof*⟩

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 $- \bot = (\bot{::}Integer)$
  ⟨*proof*⟩

lemma *Integer-arith-simps* [*simp*]:
  $MkI{\cdot}a + MkI{\cdot}b = MkI{\cdot}(a + b)$
  $MkI{\cdot}a * MkI{\cdot}b = MkI{\cdot}(a * b)$
  $MkI{\cdot}a - MkI{\cdot}b = MkI{\cdot}(a - b)$
  $- MkI{\cdot}a = MkI{\cdot}(uminus\ a)$
  ⟨*proof*⟩

lemma *plus-MkI-MkI*:
  $MkI{\cdot}x + MkI{\cdot}y = MkI{\cdot}(x + y)$
  ⟨*proof*⟩

instance *Integer* :: {*plus-cpo*,*minus-cpo*,*times-cpo*}
  ⟨*proof*⟩

instance *Integer* :: *comm-monoid-add*
⟨*proof*⟩

instance *Integer* :: *comm-monoid-mult*
⟨*proof*⟩

instance *Integer* :: *comm-semiring*
⟨*proof*⟩

instance *Integer* :: *semiring-numeral* ⟨*proof*⟩

lemma *Integer-add-diff-cancel* [*simp*]:
$b \neq \bot \Longrightarrow (a::Integer) + b - b = a$
⟨*proof*⟩

lemma *zero-Integer-neq-bottom* [*simp*]: $(0::Integer) \neq \bot$
⟨*proof*⟩

lemma *one-Integer-neq-bottom* [*simp*]: $(1::Integer) \neq \bot$
⟨*proof*⟩

lemma *plus-Integer-eq-bottom-iff* [*simp*]:
fixes $x\ y$ :: *Integer* **shows** $x + y = \bot \longleftrightarrow x = \bot \vee y = \bot$
⟨*proof*⟩

lemma *diff-Integer-eq-bottom-iff* [*simp*]:
fixes $x\ y$ :: *Integer* **shows** $x - y = \bot \longleftrightarrow x = \bot \vee y = \bot$
⟨*proof*⟩

lemma *mult-Integer-eq-bottom-iff* [*simp*]:
fixes $x\ y$ :: *Integer* **shows** $x * y = \bot \longleftrightarrow x = \bot \vee y = \bot$
⟨*proof*⟩

lemma *minus-Integer-eq-bottom-iff* [*simp*]:
fixes x :: *Integer* **shows** $- x = \bot \longleftrightarrow x = \bot$
⟨*proof*⟩

lemma *numeral-Integer-eq*: *numeral* k = *MkI*·(*numeral* k)
⟨*proof*⟩

lemma *numeral-Integer-neq-bottom* [*simp*]: (*numeral* k::*Integer*) $\neq \bot$
⟨*proof*⟩

Symmetric versions are also needed, because the reorient simproc does not apply to these comparisons.

lemma *bottom-neq-zero-Integer* [*simp*]: $(\bot::Integer) \neq 0$
⟨*proof*⟩

lemma *bottom-neq-one-Integer* [*simp*]: $(\bot::Integer) \neq 1$
⟨*proof*⟩

lemma *bottom-neq-numeral-Integer* [*simp*]: (⊥::*Integer*) ≠ *numeral* k
⟨*proof*⟩

instantiation *Integer* :: *Ord-linear*

begin

definition
$eq = (\Lambda\ (MkI{\cdot}x)\ (MkI{\cdot}y).\ if\ x = y\ then\ TT\ else\ FF)$

definition
$compare = (\Lambda\ (MkI{\cdot}x)\ (MkI{\cdot}y).$
$\quad if\ x < y\ then\ LT\ else\ if\ x > y\ then\ GT\ else\ EQ)$

instance ⟨*proof*⟩

end

lemma *eq-MkI-MkI* [*simp*]:
$eq{\cdot}(MkI{\cdot}m){\cdot}(MkI{\cdot}n) = (if\ m = n\ then\ TT\ else\ FF)$
⟨*proof*⟩

lemma *compare-MkI-MkI* [*simp*]:
$compare{\cdot}(MkI{\cdot}x){\cdot}(MkI{\cdot}y) = (if\ x < y\ then\ LT\ else\ if\ x > y\ then\ GT\ else\ EQ)$
⟨*proof*⟩

lemma *lt-MkI-MkI* [*simp*]:
$lt{\cdot}(MkI{\cdot}x){\cdot}(MkI{\cdot}y) = (if\ x < y\ then\ TT\ else\ FF)$
⟨*proof*⟩

lemma *le-MkI-MkI* [*simp*]:
$le{\cdot}(MkI{\cdot}x){\cdot}(MkI{\cdot}y) = (if\ x \leq y\ then\ TT\ else\ FF)$
⟨*proof*⟩

lemma *eq-Integer-bottom-iff* [*simp*]:
fixes $x\ y$ :: *Integer* **shows** $eq{\cdot}x{\cdot}y = \bot \longleftrightarrow x = \bot \vee y = \bot$
⟨*proof*⟩

lemma *compare-Integer-bottom-iff* [*simp*]:
fixes $x\ y$ :: *Integer* **shows** $compare{\cdot}x{\cdot}y = \bot \longleftrightarrow x = \bot \vee y = \bot$
⟨*proof*⟩

lemma *lt-Integer-bottom-iff* [*simp*]:
fixes $x\ y$ :: *Integer* **shows** $lt{\cdot}x{\cdot}y = \bot \longleftrightarrow x = \bot \vee y = \bot$
⟨*proof*⟩

lemma *le-Integer-bottom-iff* [*simp*]:
fixes $x\ y$ :: *Integer* **shows** $le{\cdot}x{\cdot}y = \bot \longleftrightarrow x = \bot \vee y = \bot$
⟨*proof*⟩

lemma *compare-refl-Integer* [*simp*]:
$(x{::}Integer) \neq \bot \Longrightarrow compare{\cdot}x{\cdot}x = EQ$
⟨*proof*⟩

lemma *eq-refl-Integer* [*simp*]:
$(x::Integer) \neq \bot \Longrightarrow eq\cdot x\cdot x = TT$
⟨*proof*⟩

lemma *lt-refl-Integer* [*simp*]:
$(x::Integer) \neq \bot \Longrightarrow lt\cdot x\cdot x = FF$
⟨*proof*⟩

lemma *le-refl-Integer* [*simp*]:
$(x::Integer) \neq \bot \Longrightarrow le\cdot x\cdot x = TT$
⟨*proof*⟩

lemma *eq-Integer-numeral-simps* [*simp*]:
$eq\cdot(0::Integer)\cdot 0 = TT$
$eq\cdot(0::Integer)\cdot 1 = FF$
$eq\cdot(1::Integer)\cdot 0 = FF$
$eq\cdot(1::Integer)\cdot 1 = TT$
$eq\cdot(0::Integer)\cdot(numeral\ k) = FF$
$eq\cdot(numeral\ k)\cdot(0::Integer) = FF$
$k \neq Num.One \Longrightarrow eq\cdot(1::Integer)\cdot(numeral\ k) = FF$
$k \neq Num.One \Longrightarrow eq\cdot(numeral\ k)\cdot(1::Integer) = FF$
$eq\cdot(numeral\ k::Integer)\cdot(numeral\ l) = (if\ k = l\ then\ TT\ else\ FF)$
⟨*proof*⟩

lemma *compare-Integer-numeral-simps* [*simp*]:
$compare\cdot(0::Integer)\cdot 0 = EQ$
$compare\cdot(0::Integer)\cdot 1 = LT$
$compare\cdot(1::Integer)\cdot 0 = GT$
$compare\cdot(1::Integer)\cdot 1 = EQ$
$compare\cdot(0::Integer)\cdot(numeral\ k) = LT$
$compare\cdot(numeral\ k)\cdot(0::Integer) = GT$
$Num.One < k \Longrightarrow compare\cdot(1::Integer)\cdot(numeral\ k) = LT$
$Num.One < k \Longrightarrow compare\cdot(numeral\ k)\cdot(1::Integer) = GT$
$compare\cdot(numeral\ k::Integer)\cdot(numeral\ l) =$
$(if\ k < l\ then\ LT\ else\ if\ k > l\ then\ GT\ else\ EQ)$
⟨*proof*⟩

lemma *lt-Integer-numeral-simps* [*simp*]:
$lt\cdot(0::Integer)\cdot 0 = FF$
$lt\cdot(0::Integer)\cdot 1 = TT$
$lt\cdot(1::Integer)\cdot 0 = FF$
$lt\cdot(1::Integer)\cdot 1 = FF$
$lt\cdot(0::Integer)\cdot(numeral\ k) = TT$
$lt\cdot(numeral\ k)\cdot(0::Integer) = FF$
$Num.One < k \Longrightarrow lt\cdot(1::Integer)\cdot(numeral\ k) = TT$
$lt\cdot(numeral\ k)\cdot(1::Integer) = FF$
$lt\cdot(numeral\ k::Integer)\cdot(numeral\ l) = (if\ k < l\ then\ TT\ else\ FF)$
⟨*proof*⟩

lemma *le-Integer-numeral-simps* [*simp*]:
le·(0::Integer)·0 = TT
le·(0::Integer)·1 = TT
le·(1::Integer)·0 = FF
le·(1::Integer)·1 = TT
le·(0::Integer)·(numeral k) = TT
le·(numeral k)·(0::Integer) = FF
le·(1::Integer)·(numeral k) = TT
Num.One < k ⟹ le·(numeral k)·(1::Integer) = FF
le·(numeral k::Integer)·(numeral l) = (if k ≤ l then TT else FF)
⟨*proof*⟩

lemma *MkI-eq-0-iff* [*simp*]: *MkI·n = 0 ⟷ n = 0*
⟨*proof*⟩

lemma *MkI-eq-1-iff* [*simp*]: *MkI·n = 1 ⟷ n = 1*
⟨*proof*⟩

lemma *MkI-eq-numeral-iff* [*simp*]: *MkI·n = numeral k ⟷ n = numeral k*
⟨*proof*⟩

lemma *MkI-0*: *MkI·0 = 0*
⟨*proof*⟩

lemma *MkI-1*: *MkI·1 = 1*
⟨*proof*⟩

lemma *le-plus-1*:
fixes *m* :: *Integer*
assumes *le·m·n = TT*
shows *le·m·(n + 1) = TT*
⟨*proof*⟩

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*⟩

end

8 Data: List

theory *Data-List*
imports

Type-Classes
Data-Function
Data-Bool
Data-Tuple
Data-Integer
Numeral-Cpo
begin

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
  (:) == *CONST Cons*
  (x:) == (*CONST Rep-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-list·[]·[] = *TT* |
  eq-list·(x : xs)·[] = *FF* |
  eq-list·[]·(y : ys) = *FF* |
  eq-list·(x : xs)·(y : ys) = (*eq*·x·y *andalso* *eq-list*·xs·ys)

instance ⟨*proof*⟩

end

instance *list* :: (*Eq-sym*) *Eq-sym*

⟨*proof*⟩

instance *list* :: (*Eq-equiv*) *Eq-equiv*
⟨*proof*⟩

instance *list* :: (*Eq-eq*) *Eq-eq*
⟨*proof*⟩

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
⟨*proof*⟩

end

instance *list* :: (*Ord-linear*) *Ord-linear*
⟨*proof*⟩

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* → '*b* → '*c* → '*d*) → ['*a*] → ['*b*] → ['*c*] → ['*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*] → ['*b*] → ['*c*] → [⟨'*a*, '*b*, '*c*⟩] **where**
zip3 = *zipWith3*·⟨,,⟩

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

fixrec *filter* :: ('*a* → *tr*) → ['*a*] → ['*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*]: $repeat{\cdot}x = x : repeat{\cdot}x$

fixrec *takeWhile* :: $('a \to tr) \to ['a] \to ['a]$ **where**
 $takeWhile{\cdot}p{\cdot}[] \quad = [] \mid$
 $takeWhile{\cdot}p{\cdot}(x{:}xs) = If\ p{\cdot}x\ then\ x : takeWhile{\cdot}p{\cdot}xs\ else\ []$

fixrec *dropWhile* :: $('a \to tr) \to ['a] \to ['a]$ **where**
 $dropWhile{\cdot}p{\cdot}[] \quad = [] \mid$
 $dropWhile{\cdot}p{\cdot}(x{:}xs) = If\ p{\cdot}x\ then\ dropWhile{\cdot}p{\cdot}xs\ else\ (x{:}xs)$

fixrec *span* :: $('a$ -> $tr)$ -> $['a]$ -> $\langle['a],['a]\rangle$ **where**
 $span{\cdot}p{\cdot}[] \quad = \langle[],[]\rangle \mid$
 $span{\cdot}p{\cdot}(x{:}xs) = If\ p{\cdot}x\ then\ (case\ span{\cdot}p{\cdot}xs\ of\ \langle ys,\ zs\rangle \Rightarrow \langle x{:}ys,zs\rangle)\ else\ \langle[],\ x{:}xs\rangle$

fixrec *break* :: $('a$ -> $tr)$ -> $['a]$ -> $\langle['a],['a]\rangle$ **where**
 $break{\cdot}p = span{\cdot}(neg\ oo\ p)$

fixrec *nth* :: $['a] \to Integer \to 'a$ **where**
  $nth{\cdot}[]{\cdot}n = \bot \mid$
  nth-Cons [*simp del*]:
  $nth{\cdot}(x : xs){\cdot}n = If\ eq{\cdot}n{\cdot}0\ then\ x\ else\ nth{\cdot}xs{\cdot}(n - 1)$

abbreviation *nth-syn* :: $['a] \Rightarrow Integer \Rightarrow 'a$ (**infixl** !! *100*) **where**
  $xs\ !!\ n \equiv nth{\cdot}xs{\cdot}n$

definition *partition* :: $('a \to tr) \to ['a] \to \langle['a],\ ['a]\rangle$ **where**
  $partition = (\Lambda\ P\ xs.\ \langle filter{\cdot}P{\cdot}xs,\ filter{\cdot}(neg\ oo\ P){\cdot}xs\rangle)$

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

fixrec *foldl* :: $('a$ -> $'b$ -> $'a)$ -> $'a$ -> $['b]$ -> $'a$ **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* :: $('a$ -> $'a$ -> $'a)$ -> $['a]$ -> $'a$ **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 $append\text{-}syn :: ['a] \Rightarrow ['a] \Rightarrow ['a]$ (**infixr** $++\ 65$) **where**
$xs\ ++\ ys \equiv append{\cdot}xs{\cdot}ys$

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

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

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

fixrec $the\text{-}and :: [tr] \to tr$ **where**
$the\text{-}and = foldr{\cdot}trand{\cdot}TT$

fixrec $the\text{-}or :: [tr] \to tr$ **where**
$the\text{-}or = foldr{\cdot}tror{\cdot}FF$

fixrec $all :: ('a \to tr) \to ['a] \to tr$ **where**
$all{\cdot}P = the\text{-}and\ oo\ (map{\cdot}P)$

fixrec $any :: ('a \to tr) \to ['a] \to tr$ **where**
$any{\cdot}P = the\text{-}or\ oo\ (map{\cdot}P)$

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

```
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* :: $'a \to 'a$ **where**
succ = *toEnum oo* (+*1*) *oo fromEnum*

definition *pred* :: $'a \to 'a$ **where**
pred = *toEnum oo* (−*1*) *oo fromEnum*

definition *enumFrom* :: $'a \to ['a]$ **where**
enumFrom = (Λ *x*. *map*·*toEnum*·(*intsFrom*·(*fromEnum*·*x*)))

definition *enumFromTo* :: $'a \to 'a \to ['a]$ **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 ⟨*proof*⟩
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* :: $'a \to ['a] \to ['a]$ **where**
intersperse·*sep*·[] = [] |
intersperse·*sep*·[*x*] = [*x*] |
intersperse·*sep*·(*x*:*y*:*xs*) = *x*:*sep*:*intersperse*·*sep*·(*y*:*xs*)

fixrec *intercalate* :: $['a] \to [['a]] \to ['a]$ **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*
⟨*proof*⟩

lemma *adm-finite-list* [*simp*]: *adm finite-list*
⟨*proof*⟩

lemma *bot-not-finite-list* [*simp*]:
finite-list ⊥ = *False*
⟨*proof*⟩

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

lemma *listmem-simps* [*simp*]:

shows $\neg$ *listmem x* $\bot$ **and** $\neg$ *listmem x* []
and *listmem x* (*y* : *ys*) $\longleftrightarrow$ *x* = *y* $\vee$ *listmem x ys*
⟨*proof*⟩

definition *set* :: ['*a*] $\Rightarrow$ '*a set* **where**
set xs = {*x*. *listmem x xs*}

lemma *set-simps* [*simp*]:
shows *set* $\bot$ = {} **and** *set* [] = {}
and *set* (*x* : *xs*) = *insert x* (*set xs*)
⟨*proof*⟩

inductive *distinct* :: ['*a*] $\Rightarrow$ *bool* **where**
Nil [*intro*!, *simp*]: *distinct* [] |
Cons [*intro*!, *simp*]: $\bigwedge$*x xs*. *distinct xs* $\Longrightarrow$ *x* $\notin$ *set xs* $\Longrightarrow$ *distinct* (*x* : *xs*)

8.4 Properties

lemma *map-strict* [*simp*]:
map·*P*·$\bot$ = $\bot$
⟨*proof*⟩

lemma *map-ID* [*simp*]:
map·*ID*·*xs* = *xs*
⟨*proof*⟩

lemma *enumFrom-intsFrom-conv* [*simp*]:
enumFrom = *intsFrom*
⟨*proof*⟩

lemma *enumFromTo-upto-conv* [*simp*]:
enumFromTo = *upto*
⟨*proof*⟩

lemma *zipWith-strict* [*simp*]:
zipWith·*f*·$\bot$·*ys* = $\bot$
zipWith·*f*·(*x* : *xs*)·$\bot$ = $\bot$
⟨*proof*⟩

lemma *zip-simps* [*simp*]:
zip·(*x* : *xs*)·(*y* : *ys*) = ⟨*x*, *y*⟩ : *zip*·*xs*·*ys*
zip·(*x* : *xs*)·[] = []
zip·(*x* : *xs*)·$\bot$ = $\bot$
zip·[]·*ys* = []
zip·$\bot$·*ys* = $\bot$
⟨*proof*⟩

lemma *zip-Nil2* [*simp*]:
xs $\neq$ $\bot$ $\Longrightarrow$ *zip*·*xs*·[] = []

⟨*proof*⟩

lemma *nth-strict* [*simp*]:
nth·⊥·*n* = ⊥
nth·*xs*·⊥ = ⊥
⟨*proof*⟩

lemma *upto-strict* [*simp*]:
upto·⊥·*y* = ⊥
upto·*x*·⊥ = ⊥
⟨*proof*⟩

lemma *upto-simps* [*simp*]:
n < *m* ⟹ *upto*·(*MkI*·*m*)·(*MkI*·*n*) = []
m ≤ *n* ⟹ *upto*·(*MkI*·*m*)·(*MkI*·*n*) = *MkI*·*m* : [*MkI*·*m*+*1*..*MkI*·*n*]
⟨*proof*⟩

lemma *filter-strict* [*simp*]:
filter·*P*·⊥ = ⊥
⟨*proof*⟩

lemma *nth-Cons-simp* [*simp*]:
eq·*n*·*0* = *TT* ⟹ *nth*·(*x* : *xs*)·*n* = *x*
eq·*n*·*0* = *FF* ⟹ *nth*·(*x* : *xs*)·*n* = *nth*·*xs*·(*n* − *1*)
⟨*proof*⟩

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*)))

⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *take-strict* [*simp*]:
take·⊥·*xs* = ⊥
⟨*proof*⟩

lemma *take-strict-2* [*simp*]:
le·*1*·*i* = *TT* ⟹ *take*·*i*·⊥ = ⊥
⟨*proof*⟩

lemma *drop-strict* [*simp*]:
$drop{\cdot}\bot{\cdot}xs = \bot$
⟨*proof*⟩

lemma *isPrefixOf-strict* [*simp*]:
$isPrefixOf{\cdot}\bot{\cdot}xs = \bot$
$isPrefixOf{\cdot}(x{:}xs){\cdot}\bot = \bot$
⟨*proof*⟩

lemma *last-strict*[*simp*]:
$last{\cdot}\bot = \bot$
$last{\cdot}(x{:}\bot) = \bot$
⟨*proof*⟩

lemma *last-nil* [*simp*]:
$last{\cdot}[] = \bot$
⟨*proof*⟩

lemma *last-spine-strict*: $\neg$ *finite-list xs* $\Longrightarrow$ $last{\cdot}xs = \bot$
⟨*proof*⟩

lemma *init-strict* [*simp*]:
$init{\cdot}\bot = \bot$
$init{\cdot}(x{:}\bot) = \bot$
⟨*proof*⟩

lemma *init-nil* [*simp*]:
$init{\cdot}[] = \bot$
⟨*proof*⟩

lemma *strict-foldr-strict2* [*simp*]:
$(\bigwedge x.\ f{\cdot}x{\cdot}\bot = \bot) \Longrightarrow foldr{\cdot}f{\cdot}\bot{\cdot}xs = \bot$
⟨*proof*⟩

lemma *foldr-strict* [*simp*]:
$foldr{\cdot}f{\cdot}d{\cdot}\bot = \bot$
$foldr{\cdot}f{\cdot}\bot{\cdot}[] = \bot$
$foldr{\cdot}\bot{\cdot}d{\cdot}(x : xs) = \bot$
⟨*proof*⟩

lemma *foldr-Cons-Nil* [*simp*]:
$foldr{\cdot}(:){\cdot}[]{\cdot}xs = xs$
⟨*proof*⟩

lemma *append-strict1* [*simp*]:
$\bot +\!+ ys = \bot$
⟨*proof*⟩

lemma *foldr-append* [*simp*]:
foldr·*f*·*a*·(*xs* ++ *ys*) = *foldr*·*f*·(*foldr*·*f*·*a*·*ys*)·*xs*
⟨*proof*⟩

lemma *foldl-strict* [*simp*]:
foldl·*f*·*d*·⊥ = ⊥
foldl·*f*·⊥·[] = ⊥
⟨*proof*⟩

lemma *foldr1-strict* [*simp*]:
foldr1·*f*·⊥= ⊥
foldr1·*f*·(*x*:⊥)= ⊥
⟨*proof*⟩

lemma *foldl1-strict* [*simp*]:
foldl1·*f*·⊥= ⊥
⟨*proof*⟩

lemma *foldl-spine-strict*:
¬ *finite-list xs* ⟹ *foldl*·*f*·*x*·*xs* = ⊥
⟨*proof*⟩

lemma *foldl-assoc-foldr*:
assumes *finite-list xs*
and *assoc*: ⋀*x y z*. *f*·(*f*·*x*·*y*)·*z* = *f*·*x*·(*f*·*y*·*z*)
and *neutr1*: ⋀*x*. *f*·*z*·*x* = *x*
and *neutr2*: ⋀*x*. *f*·*x*·*z* = *x*
shows *foldl*·*f*·*z*·*xs* = *foldr*·*f*·*z*·*xs*
⟨*proof*⟩

lemma *elem-strict* [*simp*]:
elem·*x*·⊥ = ⊥
⟨*proof*⟩

lemma *notElem-strict* [*simp*]:
notElem·*x*·⊥ = ⊥
⟨*proof*⟩

lemma *list-eq-nil*[*simp*]:
eq·*l*·[] = *TT* ⟷ *l* = []
eq·[]·*l* = *TT* ⟷ *l* = []
⟨*proof*⟩

lemma *take-Nil* [*simp*]:
n ≠ ⊥ ⟹ *take*·*n*·[] = []
⟨*proof*⟩

lemma *take-0* [*simp*]:
take·*0*·*xs* = []

take·(MkI·0)·xs = []
⟨*proof*⟩

lemma *take-Cons* [*simp*]:
le·1·i = TT ⟹ take·i·(x:xs) = x : take·(i − 1)·xs
⟨*proof*⟩

lemma *take-MkI-Cons* [*simp*]:
0 < n ⟹ take·(MkI·n)·(x : xs) = x : take·(MkI·(n − 1))·xs
⟨*proof*⟩

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
⟨*proof*⟩

lemma *drop-0* [*simp*]:
drop·0·xs = xs
drop·(MkI·0)·xs = xs
⟨*proof*⟩

lemma *drop-pos* [*simp*]:
le·n·0 = FF ⟹ drop·n·xs = (case xs of [] ⇒ [] | y : ys ⇒ drop·(n − 1)·ys)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *take-drop-append*:
take·(MkI·i)·xs ++ drop·(MkI·i)·xs = xs
⟨*proof*⟩

lemma *take-intsFrom-enumFrom* [*simp*]:
take·(MkI·n)·[MkI·i..] = [MkI·i..MkI·(n+i) − 1]
⟨*proof*⟩

lemma *drop-intsFrom-enumFrom* [*simp*]:
assumes *n ≥ 0*
shows *drop·(MkI·n)·[MkI·i..] = [MkI·(n+i)..]*
⟨*proof*⟩

lemma *last-append-singleton*:
finite-list xs ⟹ last·(xs ++ [x]) = x
⟨*proof*⟩

lemma *init-append-singleton*:
finite-list xs $\Longrightarrow$ *init*·(*xs* ++ [*x*]) = *xs*
⟨*proof*⟩

lemma *append-Nil2* [*simp*]:
xs ++ [] = *xs*
⟨*proof*⟩

lemma *append-assoc* [*simp*]:
(*xs* ++ *ys*) ++ *zs* = *xs* ++ *ys* ++ *zs*
⟨*proof*⟩

lemma *concat-simps* [*simp*]:
concat·[] = []
concat·(*xs* : *xss*) = *xs* ++ *concat*·*xss*
concat·⊥ = ⊥
⟨*proof*⟩

lemma *concatMap-simps* [*simp*]:
concatMap·*f*·[] = []
concatMap·*f*·(*x* : *xs*) = *f*·*x* ++ *concatMap*·*f*·*xs*
concatMap·*f*·⊥ = ⊥
⟨*proof*⟩

lemma *filter-append* [*simp*]:
filter·*P*·(*xs* ++ *ys*) = *filter*·*P*·*xs* ++ *filter*·*P*·*ys*
⟨*proof*⟩

lemma *elem-append* [*simp*]:
elem·*x*·(*xs* ++ *ys*) = (*elem*·*x*·*xs orelse elem*·*x*·*ys*)
⟨*proof*⟩

lemma *filter-filter* [*simp*]:
filter·*P*·(*filter*·*Q*·*xs*) = *filter*·(Λ *x*. *Q*·*x andalso P*·*x*)·*xs*
⟨*proof*⟩

lemma *filter-const-TT* [*simp*]:
filter·(Λ -. *TT*)·*xs* = *xs*
⟨*proof*⟩

lemma *tails-strict* [*simp*]:
tails·⊥ = ⊥
⟨*proof*⟩

lemma *inits-strict* [*simp*]:
inits·⊥ = ⊥
⟨*proof*⟩

lemma *the-and-strict* [*simp*]:

the-and·⊥ = ⊥
⟨*proof*⟩

lemma *the-or-strict* [*simp*]:
the-or·⊥ = ⊥
⟨*proof*⟩

lemma *all-strict* [*simp*]:
all·*P*·⊥ = ⊥
⟨*proof*⟩

lemma *any-strict* [*simp*]:
any·*P*·⊥ = ⊥
⟨*proof*⟩

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

lemma *inits-neq-Nil* [*simp*]:
inits·*xs* ≠ []
⟨*proof*⟩

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

lemma *Nil-neq-inits* [*simp*]:
[] ≠ *inits*·*xs*
⟨*proof*⟩

lemma *finite-list-not-bottom* [*simp*]:
assumes *finite-list xs* **shows** *xs* ≠ ⊥
⟨*proof*⟩

lemma *head-append* [*simp*]:
head·(*xs* ++ *ys*) = *If null*·*xs then head*·*ys else head*·*xs*
⟨*proof*⟩

lemma *filter-cong*:
∀ *x*∈*set xs*. *p*·*x* = *q*·*x* ⟹ *filter*·*p*·*xs* = *filter*·*q*·*xs*
⟨*proof*⟩

lemma *filter-TT* [*simp*]:
assumes ∀ *x*∈*set xs*. *P*·*x* = *TT*
shows *filter*·*P*·*xs* = *xs*
⟨*proof*⟩

lemma *filter-FF* [*simp*]:

assumes *finite-list xs*
and $\forall x{\in}set\ xs.\ P{\cdot}x = FF$
shows $filter{\cdot}P{\cdot}xs = []$
⟨*proof*⟩

lemma *map-cong*:
$\forall x{\in}set\ xs.\ p{\cdot}x = q{\cdot}x \Longrightarrow map{\cdot}p{\cdot}xs = map{\cdot}q{\cdot}xs$
⟨*proof*⟩

lemma *finite-list-upto*:
finite-list $(upto{\cdot}(MkI{\cdot}m){\cdot}(MkI{\cdot}n))$ (**is** *?P m n*)
⟨*proof*⟩

lemma *filter-commute*:
assumes $\forall x{\in}set\ xs.\ (Q{\cdot}x\ andalso\ P{\cdot}x) = (P{\cdot}x\ andalso\ Q{\cdot}x)$
shows $filter{\cdot}P{\cdot}(filter{\cdot}Q{\cdot}xs) = filter{\cdot}Q{\cdot}(filter{\cdot}P{\cdot}xs)$
⟨*proof*⟩

lemma *upto-append-intsFrom* [*simp*]:
assumes $m \leq n$
shows $upto{\cdot}(MkI{\cdot}m){\cdot}(MkI{\cdot}n) +\!\!+ intsFrom{\cdot}(MkI{\cdot}n{+}1) = intsFrom{\cdot}(MkI{\cdot}m)$
(**is** *?u m n ++ - = ?i m*)
⟨*proof*⟩

lemma *set-upto* [*simp*]:
$set\ (upto{\cdot}(MkI{\cdot}m){\cdot}(MkI{\cdot}n)) = \{MkI{\cdot}i \mid i.\ m \leq i \wedge i \leq n\}$
(**is** *set (?u m n) = ?R m n*)
⟨*proof*⟩

lemma *Nil-append-iff* [*iff*]:
$xs +\!\!+ ys = [] \longleftrightarrow xs = [] \wedge ys = []$
⟨*proof*⟩

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

lemma *bottom-neq-Nil* [*simp*]: $\bot \neq []$
⟨*proof*⟩

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

⟨*ML*⟩

lemma *set-append* [*simp*]:
assumes *finite-list xs*
shows $set\ (xs +\!\!+ ys) = set\ xs \cup set\ ys$
⟨*proof*⟩

lemma *distinct-Cons* [*simp*]:
$distinct\ (x : xs) \longleftrightarrow distinct\ xs \wedge x \notin set\ xs$

(**is** *?l* = *?r*)
⟨*proof*⟩

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

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*)
⟨*proof*⟩

lemma *finite-set* [*simp*]:
assumes *distinct xs*
shows *finite* (*set xs*)
⟨*proof*⟩

lemma *distinct-card*:
assumes *distinct xs*
shows *MkI*·(*int* (*card* (*set xs*))) = *length*·*xs*
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *distinct-delete* [*simp*]:
fixes *xs* :: ['*a*::*Eq-eq*]
assumes *distinct xs*
and ∀ *x*∈*set xs*. *eq*·*a*·*x* ≠ ⊥
shows *distinct* (*delete*·*a*·*xs*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *distinct-upto* [*intro*, *simp*]:
distinct [*MkI·m*..*MkI·n*]
⟨*proof*⟩

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

lemma *If-eq-bottom-iff* [*simp*]:
(*If b then x else y* = ⊥) ⟷ *b* = ⊥ ∨ *b* = *TT* ∧ *x* = ⊥ ∨ *b* = *FF* ∧ *y* = ⊥
⟨*proof*⟩

lemma *upto-eq-bottom-iff* [*simp*]:
upto·m·n = ⊥ ⟷ *m* = ⊥ ∨ *n* = ⊥
⟨*proof*⟩

lemma *seq-eq-bottom-iff* [*simp*]:
seq·x·y = ⊥ ⟷ *x* = ⊥ ∨ *y* = ⊥
⟨*proof*⟩

lemma *intsFrom-eq-bottom-iff* [*simp*]:
intsFrom·m = ⊥ ⟷ *m* = ⊥
⟨*proof*⟩

lemma *intsFrom-split*:
assumes *m* ≥ *n*
shows [*MkI·n*..] = [*MkI·n* .. *MkI·*(*m* − *1*)] ++ [*MkI·m*..]
⟨*proof*⟩

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*⟩

lemma *null-eq-TT-iff* [*simp*]:

$null{\cdot}xs = TT \longleftrightarrow xs = []$
⟨*proof*⟩

lemma *null-set-empty-conv*:
$xs \neq \bot \Longrightarrow null{\cdot}xs = TT \longleftrightarrow set\ xs = \{\}$
⟨*proof*⟩

lemma *elem-TT* [*simp*]:
fixes $x :: 'a$::*Eq-eq* **shows** $elem{\cdot}x{\cdot}xs = TT \Longrightarrow x \in set\ xs$
⟨*proof*⟩

lemma *elem-FF* [*simp*]:
fixes $x :: 'a$::*Eq-equiv* **shows** $elem{\cdot}x{\cdot}xs = FF \Longrightarrow x \notin set\ xs$
⟨*proof*⟩

lemma *length-strict* [*simp*]:
$length{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *repeat-neq-bottom* [*simp*]:
$repeat{\cdot}x \neq \bot$
⟨*proof*⟩

lemma *list-case-repeat* [*simp*]:
$list\text{-}case{\cdot}a{\cdot}f{\cdot}(repeat{\cdot}x) = f{\cdot}x{\cdot}(repeat{\cdot}x)$
⟨*proof*⟩

lemma *length-append* [*simp*]:
$length{\cdot}(xs +\!\!+ ys) = length{\cdot}xs + length{\cdot}ys$
⟨*proof*⟩

lemma *replicate-strict* [*simp*]:
$replicate{\cdot}\bot{\cdot}x = \bot$
⟨*proof*⟩

lemma *replicate-0* [*simp*]:
$replicate{\cdot}0{\cdot}x = []$
$replicate{\cdot}(MkI{\cdot}0){\cdot}xs = []$
⟨*proof*⟩

lemma *Integer-add-0* [*simp*]: $MkI{\cdot}0 + n = n$
⟨*proof*⟩

lemma *replicate-MkI-plus-1* [*simp*]:
$0 \leq n \Longrightarrow replicate{\cdot}(MkI{\cdot}(n{+}1)){\cdot}x = x : replicate{\cdot}(MkI{\cdot}n){\cdot}x$
$0 \leq n \Longrightarrow replicate{\cdot}(MkI{\cdot}(1{+}n)){\cdot}x = x : replicate{\cdot}(MkI{\cdot}n){\cdot}x$
⟨*proof*⟩

lemma *replicate-append-plus-conv*:

assumes $0 \leq m$ **and** $0 \leq n$
shows $replicate{\cdot}(MkI{\cdot}m){\cdot}x +\!\!+ replicate{\cdot}(MkI{\cdot}n){\cdot}x$
$= replicate{\cdot}(MkI{\cdot}m + MkI{\cdot}n){\cdot}x$
⟨*proof*⟩

lemma *replicate-MkI-1* [*simp*]:
$replicate{\cdot}(MkI{\cdot}1){\cdot}x = x : []$
⟨*proof*⟩

lemma *length-replicate* [*simp*]:
assumes $0 \leq n$
shows $length{\cdot}(replicate{\cdot}(MkI{\cdot}n){\cdot}x) = MkI{\cdot}n$
⟨*proof*⟩

lemma *map-oo* [*simp*]:
$map{\cdot}f{\cdot}(map{\cdot}g{\cdot}xs) = map{\cdot}(f\ oo\ g){\cdot}xs$
⟨*proof*⟩

lemma *nth-Cons-MkI* [*simp*]:
$0 < i \Longrightarrow (a : xs)\ !!\ (MkI{\cdot}i) = xs\ !!\ (MkI{\cdot}(i - 1))$
⟨*proof*⟩

lemma *map-plus-intsFrom*:
$map{\cdot}(+\ MkI{\cdot}n){\cdot}(intsFrom{\cdot}(MkI{\cdot}m)) = intsFrom{\cdot}(MkI{\cdot}(m{+}n))$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *plus-eq-MkI-conv*:
$l + n = MkI{\cdot}m \longleftrightarrow (\exists\, l'\ n'.\ l = MkI{\cdot}l' \wedge n = MkI{\cdot}n' \wedge m = l' + n')$
⟨*proof*⟩

lemma *length-ge-0*:
$length{\cdot}xs = MkI{\cdot}n \Longrightarrow n \geq 0$
⟨*proof*⟩

lemma *length-0-conv* [*simp*]:
$length{\cdot}xs = MkI{\cdot}0 \longleftrightarrow xs = []$
⟨*proof*⟩

lemma *length-ge-1* [*simp*]:
$length{\cdot}xs = MkI{\cdot}(1\ +\ int\ n)$
$\longleftrightarrow (\exists\, u\ us.\ xs = u : us \wedge length{\cdot}us = MkI{\cdot}(int\ n))$
(**is** *?l* = *?r*)
⟨*proof*⟩

lemma *finite-list-length-conv*:
$finite\text{-}list\ xs \longleftrightarrow (\exists\, n.\ length{\cdot}xs = MkI{\cdot}(int\ n))$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *nth-append*:

assumes *length·xs* = *MkI·n* **and** $n \leq m$
shows (*xs* ++ *ys*) !! *MkI·m* = *ys* !! *MkI·*($m - n$)
⟨*proof*⟩

lemma *replicate-nth* [*simp*]:
assumes $0 \leq n$
shows (*replicate·*(*MkI·n*)·*x* ++ *xs*) !! *MkI·n* = *xs* !! *MkI·0*
⟨*proof*⟩

lemma *map2-zip*:
map·(Λ⟨*x*, *y*⟩. ⟨*x*, *f·y*⟩)·(*zip·xs·ys*) = *zip·xs·*(*map·f·ys*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *map-map-snd*:
f·⊥ = ⊥ ⟹ *map·f·*(*map·snd·xs*)
= *map·snd·*(*map·*(Λ⟨*x*, *y*⟩. ⟨*x*, *f·y*⟩)·*xs*)
⟨*proof*⟩

lemma *findIndices-Cons* [*simp*]:
findIndices·P·(*a* : *xs*) =
If P·a then 0 : *map·*(+*1*)·(*findIndices·P·xs*)
else map·(+*1*)·(*findIndices·P·xs*)
⟨*proof*⟩

lemma *filter-alt-def*:
fixes *xs* :: ['*a*]
shows *filter·P·xs* = *map·*(*nth·xs*)·(*findIndices·P·xs*)
⟨*proof*⟩

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*⟩

8.5 *reverse* and *reverse* induction

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

lemma *reverse-Nil* [*simp*]:
reverse·[] = []
⟨*proof*⟩

lemma *reverse-singleton* [*simp*]:
reverse·[*x*] = [*x*]
⟨*proof*⟩

lemma *reverse-strict* [*simp*]:
reverse·⊥ = ⊥
⟨*proof*⟩

lemma *foldl-flip-Cons-append*:
foldl·(*flip*·(:))·*ys*·*xs* = *foldl*·(*flip*·(:))·[]·*xs* ++ *ys*
⟨*proof*⟩

lemma *reverse-Cons* [*simp*]:
reverse·(*x*:*xs*) = *reverse*·*xs* ++ [*x*]
⟨*proof*⟩

lemma *reverse-append-below*:
reverse·(*xs* ++ *ys*) ⊑ *reverse*·*ys* ++ *reverse*·*xs*
⟨*proof*⟩

lemma *reverse-reverse-below*:
reverse·(*reverse*·*xs*) ⊑ *xs*
⟨*proof*⟩

lemma *reverse-append* [*simp*]:
assumes *finite-list xs*
shows *reverse*·(*xs* ++ *ys*) = *reverse*·*ys* ++ *reverse*·*xs*
⟨*proof*⟩

lemma *reverse-spine-strict*:
¬ *finite-list xs* ⟹ *reverse*·*xs* = ⊥
⟨*proof*⟩

lemma *reverse-finite* [*simp*]:
assumes *finite-list xs* **shows** *finite-list* (*reverse*·*xs*)
⟨*proof*⟩

lemma *reverse-reverse* [*simp*]:
assumes *finite-list xs* **shows** *reverse*·(*reverse*·*xs*) = *xs*
⟨*proof*⟩

lemma *reverse-induct* [*consumes 1*, *case-names Nil snoc*]:
⟦*finite-list xs*; *P* []; ⋀*x xs* . *finite-list xs* ⟹ *P xs* ⟹ *P* (*xs* ++ [*x*])⟧ ⟹ *P xs*
⟨*proof*⟩

lemma *length-plus-not-0*:
le·*1*·*n* = *TT* ⟹ *le*·(*length*·*xs* + *n*)·*0* = *TT* ⟹ *False*
⟨*proof*⟩

lemma *take-length-plus-1*:
length·xs $\neq \bot \Longrightarrow$ *take·*(*length·xs* + *1*)*·*(*y*:*ys*) = *y* : *take·*(*length·xs*)*·ys*
⟨*proof*⟩

lemma *le-length-plus*:
length·xs $\neq \bot \Longrightarrow$ *n* $\neq \bot \Longrightarrow$ *le·n·*(*length·xs* + *n*) = *TT*
⟨*proof*⟩

lemma *eq-take-length-isPrefixOf*:
eq·xs·(*take·*(*length·xs*)*·ys*) $\sqsubseteq$ *isPrefixOf·xs·ys*
⟨*proof*⟩

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⟩

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
  ⟨proof⟩

instance Maybe :: (Eq-sym) Eq-sym
⟨proof⟩

instance Maybe :: (Eq-equiv) Eq-equiv
⟨proof⟩

instance Maybe :: (Eq-eq) Eq-eq
⟨proof⟩

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⟩

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*
⟨*proof*⟩

instance *Maybe* :: (*Ord-linear*) *Ord-linear*
⟨*proof*⟩

lemma *isJust-strict* [*simp*]: *isJust*·⊥ = ⊥ ⟨*proof*⟩
lemma *fromMaybe-strict* [*simp*]: *fromMaybe*·*x*·⊥ = ⊥ ⟨*proof*⟩
lemma *maybeToList-strict* [*simp*]: *maybeToList*·⊥ = ⊥ ⟨*proof*⟩

end

10 Definedness

theory *Definedness*
imports
Data-List
HOL−Library.Adhoc-Overloading
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* ⊥
⟨*proof*⟩

lemma *defined-seq* [*simp*]: *defined x* ⟹ *seq*·*x*·*y* = *y*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *val-Bool-simp1* [*simp*]:

$[\![Def\ i]\!] = i$
⟨*proof*⟩

lemma *val-Bool-simp2* [*simp*]:
$[\![TT]\!] = True$
$[\![FF]\!] = False$
⟨*proof*⟩

lemma *IF-simps* [*simp*]:
$defined\ b \Longrightarrow [\![\ b\ ]\!] \Longrightarrow (If\ b\ then\ x\ else\ y) = x$
$defined\ b \Longrightarrow [\![\ b\ ]\!] = False \Longrightarrow (If\ b\ then\ x\ else\ y) = y$
⟨*proof*⟩

lemma *defined-neg* [*simp*]: $defined\ (neg{\cdot}b) \longleftrightarrow defined\ b$
⟨*proof*⟩

lemma *val-Bool-neg* [*simp*]: $defined\ b \Longrightarrow [\![\ neg\ \cdot\ b\ ]\!] = (\neg\ [\![\ b\ ]\!])$
⟨*proof*⟩

val for integers

definition *val-Integer* :: *Integer* ⇒ *int* **where**
$val\text{-}Integer\ i = (THE\ j.\ i = MkI{\cdot}j)$

adhoc-overloading
val val-Integer

lemma *defined-Integer-simps* [*simp*]:
$defined\ (MkI{\cdot}i)$
$defined\ (0{::}Integer)$
$defined\ (1{::}Integer)$
⟨*proof*⟩

lemma *defined-numeral* [*simp*]: $defined\ (numeral\ x :: Integer)$
⟨*proof*⟩

lemma *val-Integer-simps* [*simp*]:
$[\![MkI{\cdot}i]\!] = i$
$[\![0]\!] = 0$
$[\![1]\!] = 1$
⟨*proof*⟩

lemma *val-Integer-numeral* [*simp*]: $[\![\ numeral\ x :: Integer\ ]\!] = numeral\ x$
⟨*proof*⟩

lemma *val-Integer-to-MkI*:
$defined\ i \Longrightarrow i = (MkI\ \cdot\ [\![\ i\ ]\!])$
⟨*proof*⟩

lemma *defined-Integer-minus* [*simp*]: *defined i* $\Longrightarrow$ *defined j* $\Longrightarrow$ *defined* (*i* − (*j*::*Integer*))
⟨*proof*⟩

lemma *val-Integer-minus* [*simp*]: *defined i* $\Longrightarrow$ *defined j* $\Longrightarrow$ ⟦ *i* − *j* ⟧ = ⟦ *i* ⟧ − ⟦ *j* ⟧
⟨*proof*⟩

lemma *defined-Integer-plus* [*simp*]: *defined i* $\Longrightarrow$ *defined j* $\Longrightarrow$ *defined* (*i* + (*j*::*Integer*))
⟨*proof*⟩

lemma *val-Integer-plus* [*simp*]: *defined i* $\Longrightarrow$ *defined j* $\Longrightarrow$ ⟦ *i* + *j* ⟧ = ⟦ *i* ⟧ + ⟦ *j* ⟧
⟨*proof*⟩

lemma *defined-Integer-eq* [*simp*]: *defined* (*eq*·*a*·*b*) $\longleftrightarrow$ *defined a* ∧ *defined* (*b*::*Integer*)
⟨*proof*⟩

lemma *val-Integer-eq* [*simp*]: *defined a* $\Longrightarrow$ *defined b* $\Longrightarrow$ ⟦ *eq*·*a*·*b* ⟧ = (⟦ *a* ⟧ = (⟦ *b* ⟧ :: *int*))
⟨*proof*⟩

Full induction for non-negative integers

lemma *nonneg-full-Int-induct* [*consumes 1*, *case-names neg Suc*]:
assumes *defined*: *defined i*
assumes *neg*: $\bigwedge$ *i*. *defined i* $\Longrightarrow$ ⟦*i*⟧ < *0* $\Longrightarrow$ *P i*
assumes *step*: $\bigwedge$ *i*. *defined i* $\Longrightarrow$ *0* ≤ ⟦*i*⟧ $\Longrightarrow$ ($\bigwedge$ *j*. *defined j* $\Longrightarrow$ ⟦ *j* ⟧ < ⟦ *i* ⟧ $\Longrightarrow$ *P j*) $\Longrightarrow$ *P i*
shows *P* (*i*::*Integer*)
⟨*proof*⟩

Some list lemmas re-done with the new setup.

lemma *nth-tail*:
defined n $\Longrightarrow$ ⟦ *n* ⟧ ≥ *0* $\Longrightarrow$ *tail*·*xs* !! *n* = *xs* !! (*1* + *n*)
⟨*proof*⟩

lemma *nth-zipWith*:
assumes *f1* [*simp*]: $\bigwedge$*y*. *f*·⊥·*y* = ⊥
assumes *f2* [*simp*]: $\bigwedge$*x*. *f*·*x*·⊥ = ⊥
shows *zipWith*·*f*·*xs*·*ys* !! *n* = *f*·(*xs* !! *n*)·(*ys* !! *n*)
⟨*proof*⟩

lemma *nth-neg* [*simp*]: *defined n* $\Longrightarrow$ ⟦ *n* ⟧ < *0* $\Longrightarrow$ *nth*·*xs*·*n* = ⊥
⟨*proof*⟩

lemma *nth-Cons-simp* [*simp*]:
defined n $\Longrightarrow$ ⟦ *n* ⟧ = *0* $\Longrightarrow$ *nth*·(*x* : *xs*)·*n* = *x*
defined n $\Longrightarrow$ ⟦ *n* ⟧ > *0* $\Longrightarrow$ *nth*·(*x* : *xs*)·*n* = *nth*·*xs*·(*n* − *1*)

⟨*proof*⟩

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

⟨*ML*⟩

lemma *concatMap-singleton* [*simp*]:
concatMap·(Λ *x*. [*f*·*x*])·*xs* = *map*·*f*·*xs*
⟨*proof*⟩

lemma *listcompr-filter* [*simp*]:
[*x* | *x* <− *xs*, *P*·*x*] = *filter*·*P*·*xs*
⟨*proof*⟩

lemma [*y* | *let y* = *x*∗*2*; *z* = *y*, *x* <− *xs*] = *A*
⟨*proof*⟩

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⟨*proof*⟩
end

instance *Integer* :: *Num*
⟨*proof*⟩

instance *Integer* :: *Num-faithful*
⟨*proof*⟩

instance *Integer* :: *Num-strict*
⟨*proof*⟩

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 ⟨*proof*⟩
end

instance *Integer* :: *Integral-strict*
⟨*proof*⟩

instance *Integer* :: *Integral-faithful*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *fibs-1* [*simp*]:
  fibs !! *1* = *1*
  ⟨*proof*⟩

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

lemma [*simp*]:−*1* + ⟦*i*⟧ = ⟦ *i* ⟧ − *1* ⟨*proof*⟩
lemma [*simp*]:−*2* + ⟦*i*⟧ = ⟦ *i* ⟧ − *2* ⟨*proof*⟩

lemma *nth-fibs*:
  assumes *defined i* **and** ⟦ *i* ⟧ ≥ *0* **shows** *defined* (*fibs* !! *i*) **and** ⟦ *fibs* !! *i* ⟧ = *fib* ⟦ *i* ⟧
  ⟨*proof*⟩

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

begin

14 The Sieve of Eratosthenes

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-Main*.

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

Rings.modulo (**infixl** *mod 70*)

no-notation
Set.member (*op :*) **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*
⟨*proof*⟩

lemma *modI-numeral-numeral* [*simp*]:
mod·(*numeral i*)·(*numeral j*) = *MkI'* (*Rings.modulo* (*numeral i*) (*numeral j*))
⟨*proof*⟩

Some lemmas demonstrating evaluation of our list:

lemma *primes* !! *0* = *2*
⟨*proof*⟩

lemma *primes* !! *1* = *3*
⟨*proof*⟩

lemma *primes* !! *2* = *5*
⟨*proof*⟩

lemma *primes* !! *3* = *7*
⟨*proof*⟩

Auxiliary lemmas about prime numbers

lemma *find-next-prime-nat*:
fixes *n* :: *nat*
assumes *prime n*
shows ∃ *n'*. *n'* > *n* ∧ *prime n'* ∧ (∀ *k*. *n* < *k* ⟶ *k* < *n'* ⟶ ¬ *prime k*)
⟨*proof*⟩

Simplification for andalso:

lemma *andAlso-Def*[*simp*]: ((*Def x*) *andalso* (*Def y*)) = *Def* (*x* ∧ *y*)
⟨*proof*⟩

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

definition *prim-bisim*:
prim-bisim x1 x2 = (∃ *n* . *prime n* ∧
x1 = *sieve*·(*filter*·(Λ (*MkI*·*i*). *Def* ((∀ *d*. *d* > *1* ⟶ *d* < *n* ⟶ ¬ (*d dvd i*)))·[*MkI*·*n*..]) ∧
x2 = *filter*·(Λ (*MkI*·*i*). *Def* (*prime* (*nat* |*i*|)))·[*MkI*·*n*..])

lemma *prim-bisim-is-bisim*: *list-bisim prim-bisim*
⟨*proof*⟩

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*..]
⟨*proof*⟩

And finally show the correctness of primes.

theorem *primes*:
shows *primes* = *filter*·(Λ (*MkI*·*i*). *Def* (*prime* (*nat* |*i*|)))·[*MkI*·*2*..]
⟨*proof*⟩

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*}
⟨*proof*⟩

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*))
⟨*proof*⟩

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

lemma *build-Abs-listfun*:
assumes *abstract-list xs* = *f*
shows *build*·(*Abs-listfun* (*f*, *xs*)) = *xs*
⟨*proof*⟩

lemma *listfun-Abs-listfun* [*simp*]:
assumes *abstract-list xs* = *f*
shows *listfun*·(*Abs-listfun* (*f*, *xs*)) = *f*
⟨*proof*⟩

lemma *augment-Abs-listfun* [*simp*]:
assumes *abstract-list xs* = *f*
shows *augment*·(*Abs-listfun* (*f*, *xs*))·*ys* = *xs* ++ *ys*
⟨*proof*⟩

lemma *cont-augment-body* [*simp*]:
cont (λ*g*. *Abs-cfun* (*op* ++ (*Product-Type.snd* (*Rep-listfun g*))))
⟨*proof*⟩

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

lemma *foldr*/*augment*:
fixes *g* :: ('*a*, '*b*) *listfun*
shows *foldr*·*k*·*z*·(*augment*·*g*·*xs*) = *listfun*·*g*·*k*·(*foldr*·*k*·*z*·*xs*)
⟨*proof*⟩

lemma *foldr*/*id*:
foldr·(:)·[] = (Λ *x*. *x*)
⟨*proof*⟩

lemma *foldr*/*app*:
foldr·(:)·*ys* = (Λ *xs*. *xs* ++ *ys*)
⟨*proof*⟩

lemma *foldr*/*cons*: *foldr*·*k*·*z*·(*x*:*xs*) = *k*·*x*·(*foldr*·*k*·*z*·*xs*) ⟨*proof*⟩
lemma *foldr*/*single*: *foldr*·*k*·*z*·[*x*] = *k*·*x*·*z* ⟨*proof*⟩
lemma *foldr*/*nil*: *foldr*·*k*·*z*·[] = *z* ⟨*proof*⟩

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*⟩

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*⟩

lemma *cont-listfun-comp-body* [*simp*]:
cont (λ*g*. Λ *h*. *Abs-listfun* (Λ *c n*. *listfun*·*g*·*c*·(*listfun*·*h*·*c*·*n*), *build*·*g* ++ *build*·*h*))
⟨*proof*⟩

lemma *abstract-list-build-append*:
abstract-list (*build*·*g* ++ *build*·*h*) = (Λ *c n*. *listfun*·*g*·*c*·(*listfun*·*h*·*c*·*n*))
⟨*proof*⟩

lemma *augment*/*build*:
augment·*g*·(*build*·*h*) = *build*·(*g* ∘*lf* *h*)

⟨*proof*⟩

lemma *augment/nil*:
augment·*g*·[] = *build*·*g*
⟨*proof*⟩

lemma *build-listfun-comp* [*simp*]:
build·(*g* ∘*lf* *h*) = *build*·*g* ++ *build*·*h*
⟨*proof*⟩

lemma *augment-augment*:
augment·*g*·(*augment*·*h*·*xs*) = *augment*·(*g* ∘*lf* *h*)·*xs*
⟨*proof*⟩

lemma *abstract-list-map* [*simp*]:
abstract-list (*map*·*f*·*xs*) = (Λ *c* *n*. *foldr*·(*mapFB*·*c*·*f*)·*n*·*xs*)
⟨*proof*⟩

lemma *map*:
map·*f*·*xs* = *build*·(*Abs-listfun* (Λ *c* *n*. *foldr*·(*mapFB*·*c*·*f*)·*n*·*xs*, *map*·*f*·*xs*))
⟨*proof*⟩

lemma *mapList*:
foldr·(*mapFB*·(:)·*f*)·[] = *map*·*f*
⟨*proof*⟩

lemma *mapFB*:
mapFB·(*mapFB*·*c*·*f*)·*g* = *mapFB*·*c*·(*f* *oo* *g*)
⟨*proof*⟩

lemma ++:
xs ++ *ys* = *augment*·(*Abs-listfun* (*abstract-list* *xs*, *xs*))·*ys*
⟨*proof*⟩

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′*·⊥ = ⊥ ⟨*proof*⟩

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*⟩

lemma *cont-Abs-listfun-down'* [*simp*]:
cont (λ*v*. *Abs-listfun* (*down'*·*v*, *down'*·*v*·(:)·[]))
⟨*proof*⟩

lemma *sum-down*:
sum·((*down TYPE*(*Integer*))·*v*) = *down'*·*v*·(+)·*0*
⟨*proof*⟩

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*]
⟨*proof*⟩

```
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*]
⟨*proof*⟩

16.2 List

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

lemma *concat*·(*map*·*f*·*x*) = *concatMap*·*f*·*x*
⟨*proof*⟩

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

lemma *concat*·[*a*, *b*] = *a* ++ *b*
⟨*proof*⟩

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

lemma *map*·*f*·(*map*·*g*·*x*) = *map*·(*f oo g*)·*x*
⟨*proof*⟩

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

lemma *x* !! *0* = *head*·*x*
⟨*proof*⟩

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

lemma *take*·*n*·(*repeat*·*x*) = *replicate*·*n*·*x*
⟨*proof*⟩

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

lemma *head*·(*reverse*·*x*) = *last*·*x*
⟨*proof*⟩

```
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*⟩

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

lemma *reverse*·(*tail*·(*reverse*·*x*)) ⊑ *init*·*x*
⟨*proof*⟩

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

lemma
assumes *x* ≠ []
shows *take*·(*length*·*x* − *1*)·*x* ⊑ *init*·*x*
⟨*proof*⟩

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

lemma *foldr-append-concat*:*foldr*·*append*·[] = *concat*
⟨*proof*⟩

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

lemma *foldl*·*append*·[] ⊑ *concat*
⟨*proof*⟩

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

lemma *span*·(*neg oo p*) = *break*·*p*
⟨*proof*⟩

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

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

⟨*proof*⟩

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

lemma *the-or*·(*map*·*p*·*x*) = *any*·*p*·*x*
⟨*proof*⟩

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

lemma *the-and*·(*map*·*p*·*x*) = *all*·*p*·*x*
⟨*proof*⟩

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

lemma *zipWith*·⟨,⟩ = *zip*
⟨*proof*⟩

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

lemma *zipWith3*·⟨,,⟩ = *zip3*
⟨*proof*⟩

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

lemma *eq*·(*length*·*x*)·*0* ⊑ *null*·*x*
⟨*proof*⟩

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

lemma *neq*·(*length*·*x*)·*0* ⊑ *neg*·(*null*·*x*)
⟨*proof*⟩

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

lemma *map*·(*uncurry*·*f*)·(*zip*·*x*·*y*) = *zipWith*·*f*·*x*·*y*
⟨*proof*⟩

```
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*⟩

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

lemma *neg*·(*elem*·*x*·*y*) = *notElem*·*x*·*y*
⟨*proof*⟩

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

lemma *foldr*·*f*·*z*·(*map*·*g*·*x*) = *foldr*·(*f oo g*)·*z*·*x*
⟨*proof*⟩

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

lemma *null*·(*filter*·*f*·*x*) = *neg*·(*any*·*f*·*x*)
⟨*proof*⟩

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

lemma $eq{\cdot}(filter{\cdot}f{\cdot}x){\cdot}[] = neg{\cdot}(any{\cdot}f{\cdot}x)$
⟨*proof*⟩

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

lemma $neq{\cdot}(filter{\cdot}f{\cdot}x){\cdot}[] = any{\cdot}f{\cdot}x$
⟨*proof*⟩

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

lemma $any{\cdot}(\Lambda\ z.\ eq{\cdot}z{\cdot}a) = elem{\cdot}a$
⟨*proof*⟩

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

lemma $any{\cdot}(eq{\cdot}(a::'a::Eq\text{-}sym)) = elem{\cdot}a$
⟨*proof*⟩

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

lemma $any{\cdot}(\Lambda\ z.\ eq{\cdot}(a::'a::Eq\text{-}sym){\cdot}z) = elem{\cdot}a$
⟨*proof*⟩

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

lemma $all{\cdot}(\Lambda\ z.\ neq{\cdot}z{\cdot}(a::'a::Eq\text{-}sym)) = notElem{\cdot}a$
⟨*proof*⟩

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

lemma $all{\cdot}(\Lambda\ z.\ neq{\cdot}(a::'a::Eq\text{-}sym){\cdot}z) = notElem{\cdot}a$
⟨*proof*⟩

16.3 Folds

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

lemma $foldr{\cdot}trand{\cdot}TT = the\text{-}and$
⟨*proof*⟩

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

lemma $foldl\text{-}to\text{-}and{:}foldl{\cdot}trand{\cdot}TT \sqsubseteq the\text{-}and$
⟨*proof*⟩

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

lemma $foldr1{\cdot}trand \sqsubseteq the\text{-}and$
⟨*proof*⟩

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

lemma $foldl1{\cdot}trand \sqsubseteq the\text{-}and$
⟨*proof*⟩

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

lemma *foldr·tror·FF = the-or*
⟨*proof*⟩

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

lemma *foldl-to-or*: *foldl·tror·FF* ⊑ *the-or*
⟨*proof*⟩

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

lemma *foldr1·tror* ⊑ *the-or*
⟨*proof*⟩

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

lemma *foldl1·tror* ⊑ *the-or*
⟨*proof*⟩

16.4 Function

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

lemma (Λ *x*. *x*) = *ID*
⟨*proof*⟩

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

lemma (Λ *x y*. *x*) = *const*
⟨*proof*⟩

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

lemma (Λ ⟨*x*, *y*⟩. *x*) = *fst*
⟨*proof*⟩

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

lemma (Λ ⟨*x*, *y*⟩. *y*) = *snd*
⟨*proof*⟩

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

lemma (Λ *x y*. *f*·⟨*x*, *y*⟩) = *curry*·*f*
⟨*proof*⟩

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

lemma (Λ ⟨*x*, *y*⟩. *f*·*x*·*y*) ⊑ *uncurry*·*f*
⟨*proof*⟩

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

lemma (Λ *x*. *y*) = *const*·*y*
⟨*proof*⟩

lemma *flip*·*f*·*x*·*y* = *f*·*y*·*x* ⟨*proof*⟩

16.5 Bool

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

lemma *eq-true*:$eq \cdot x \cdot TT = x$
⟨*proof*⟩

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

lemma *eq-false*:$eq \cdot x \cdot FF = neg \cdot x$
⟨*proof*⟩

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

lemma *if-equal*:$(If\ a\ then\ x\ else\ x) \sqsubseteq x$
⟨*proof*⟩

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

lemma $(If\ a\ then\ TT\ else\ FF) = a$
⟨*proof*⟩

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

lemma $(If\ a\ then\ FF\ else\ TT) = neg \cdot a$
⟨*proof*⟩

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

lemma $(If\ a\ then\ t\ else\ (If\ b\ then\ t\ else\ f)) = (If\ a\ orelse\ b\ then\ t\ else\ f)$
⟨*proof*⟩

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

lemma $(If\ a\ then\ (If\ b\ then\ t\ else\ f)\ else\ f) = (If\ a\ andalso\ b\ then\ t\ else\ f)$
⟨*proof*⟩

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

lemma $(If\ x\ then\ TT\ else\ y) = (x\ orelse\ y)$
⟨*proof*⟩

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

lemma $(If\ x\ then\ y\ else\ FF) = (x\ andalso\ y)$
⟨*proof*⟩

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

lemma $(If\ c\ then\ \langle TT,\ x\rangle\ else\ \langle FF,\ x\rangle) \sqsubseteq \langle c,\ x\rangle$
⟨*proof*⟩

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

lemma (*If c then* $\langle FF,\ x\rangle$ *else* $\langle TT,\ x\rangle$) $\sqsubseteq \langle neg{\cdot}c,\ x\rangle$
⟨*proof*⟩

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

lemma *the-or*$\cdot[x,\ y] = (x$ *orelse* $y)$
⟨*proof*⟩

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

lemma *the-or*$\cdot[x,\ y,\ z] = (x$ *orelse* y *orelse* $z)$
⟨*proof*⟩

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

lemma *the-and*$\cdot[x,\ y] = (x$ *andalso* $y)$
⟨*proof*⟩

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

lemma *the-and*$\cdot[x,\ y,\ z] = (x$ *andalso* y *andalso* $z)$
⟨*proof*⟩

16.6 Arrow

```
(fst x, snd x) ==>  x
```

lemma $x \sqsubseteq \langle fst{\cdot}x,\ snd{\cdot}x\rangle$
⟨*proof*⟩

16.7 Seq

```
x `seq` x ==> x
```

lemma $seq{\cdot}x{\cdot}x = x$ ⟨*proof*⟩

16.8 Evaluate

```
True && x ==> x
```

lemma $(TT$ *andalso* $x) = x$ ⟨*proof*⟩

```
False && x ==> False
```

lemma $(FF$ *andalso* $x) = FF$ ⟨*proof*⟩

```
True || x ==> True
```

lemma $(TT$ *orelse* $x) = TT$ ⟨*proof*⟩

```
False || x ==> x
```

lemma $(FF$ *orelse* $x) = x$ ⟨*proof*⟩

```
not True ==> False
```

lemma $neg{\cdot}TT = FF$ $\langle proof \rangle$

```
not False ==> True
```

lemma $neg{\cdot}FF = TT$ $\langle proof \rangle$

```
fst (x,y) ==> x
```

lemma $fst{\cdot}\langle x, y\rangle = x$ $\langle proof \rangle$

```
snd (x,y) ==> y
```

lemma $snd{\cdot}\langle x, y\rangle = y$ $\langle proof \rangle$

```
f (fst p) (snd p) ==> uncurry f p
```

lemma $f{\cdot}(fst{\cdot}p){\cdot}(snd{\cdot}p) = uncurry{\cdot}f{\cdot}p$
$\langle proof \rangle$

```
init [x] ==> []
```

lemma $init{\cdot}[x] = []$ $\langle proof \rangle$

```
null [] ==> True
```

lemma $null{\cdot}[] = TT$ $\langle proof \rangle$

```
length [] ==> 0
```

lemma $length{\cdot}[] = 0$ $\langle proof \rangle$

```
foldl f z [] ==> z
```

lemma $foldl{\cdot}f{\cdot}z{\cdot}[] = z$ $\langle proof \rangle$

```
foldr f z [] ==> z
```

lemma $foldr{\cdot}f{\cdot}z{\cdot}[] = z$ $\langle proof \rangle$

```
foldr1 f [x] ==> x
```

lemma $foldr1{\cdot}f{\cdot}[x] = x$ $\langle proof \rangle$

```
scanr f z [] ==> [z]
```

lemma $scanr{\cdot}f{\cdot}z{\cdot}[] = [z]$ $\langle proof \rangle$

```
scanr1 f [] ==> []
```

lemma $scanr1{\cdot}f{\cdot}[] = []$ $\langle proof \rangle$

```
scanr1 f [x] ==> [x]
```

lemma $scanr1{\cdot}f{\cdot}[x] = [x]$ $\langle proof \rangle$

```
take n [] ==> []
```

lemma $take{\cdot}n{\cdot}[] \sqsubseteq []$ $\langle proof \rangle$

```
drop n [] ==> []
```

lemma *drop*·*n*·[] ⊑ []
⟨*proof*⟩

```
takeWhile p [] ==> []
```

lemma *takeWhile*·*p*·[] = [] ⟨*proof*⟩

```
dropWhile p [] ==> []
```

lemma *dropWhile*·*p*·[] = [] ⟨*proof*⟩

```
span p [] ==> ([],[])
```

lemma *span*·*p*·[] = ⟨[], []⟩ ⟨*proof*⟩

```
concat [a] ==> a
```

lemma *concat*·[*a*] = *a* ⟨*proof*⟩

```
concat [] ==> []
```

lemma *concat*·[] = [] ⟨*proof*⟩

```
zip [] [] ==> []
```

lemma *zip*·[]·[] = [] ⟨*proof*⟩

```
id x ==> x
```

lemma *ID*·*x* = *x* ⟨*proof*⟩

```
const x y ==> x
```

lemma *const*·*x*·*y* = *x* ⟨*proof*⟩

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*
⟨*proof*⟩

```
(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*))
⟨*proof*⟩

lemma
assumes *le*·*0*·*i* = *TT* **and** *le*·*i*·*0* = *FF*

and $s = \bot$ **and** $t = []$
shows $\neg\ ((le{\cdot}(length{\cdot}t){\cdot}i\ andalso\ isPrefixOf{\cdot}t{\cdot}s) \sqsubseteq eq{\cdot}(take{\cdot}i{\cdot}s){\cdot}t)$
$\langle proof \rangle$

lemma $neg{\cdot}(eq{\cdot}a{\cdot}b) = neq{\cdot}a{\cdot}b\ \langle proof \rangle$

```
not (a /= b) ==> a == b
```

lemma $neg{\cdot}(neq{\cdot}a{\cdot}b) = eq{\cdot}a{\cdot}b\ \langle proof \rangle$

```
map id ==> id
```

lemma $map\text{-}id{:}map{\cdot}ID = ID\ \langle proof \rangle$

```
x == [] ==> null x
```

lemma $eq{\cdot}x{\cdot}[] = null{\cdot}x\ \langle proof \rangle$

```
any id ==> or
```

lemma $any{\cdot}ID = the\text{-}or\ \langle proof \rangle$

```
all id ==> and
```

lemma $all{\cdot}ID = the\text{-}and\ \langle proof \rangle$

```
(if x then False else y) ==> (not x && y)
```

lemma $(If\ x\ then\ FF\ else\ y) = (neg{\cdot}x\ andalso\ y)\ \langle proof \rangle$

```
(if x then y else True) ==> (not x || y)
```

lemma $(If\ x\ then\ y\ else\ TT) = (neg{\cdot}x\ orelse\ y)\ \langle proof \rangle$

```
not (not x) ==> x
```

lemma $neg{\cdot}(neg{\cdot}x) = x\ \langle proof \rangle$

```
(if c then f x else f y) ==> f (if c then x else y)
```

lemma $(If\ c\ then\ f{\cdot}x\ else\ f{\cdot}y) \sqsubseteq f{\cdot}(If\ c\ then\ x\ else\ y)\ \langle proof \rangle$

```
(\ x -> [x]) ==> (: [])
```

lemma $(\Lambda\ x.\ [x]) = (\Lambda\ z.\ z : [])\ \langle proof \rangle$

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

lemma $eq{\cdot}TT{\cdot}a = a\ \langle proof \rangle$

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

lemma $eq{\cdot}FF{\cdot}a = neg{\cdot}a\ \langle proof \rangle$

`a /= True ==> not a`

lemma $neq{\cdot}a{\cdot}TT = neg{\cdot}a\ \langle proof \rangle$

`a /= False ==> a`

lemma $neq{\cdot}a{\cdot}FF = a\ \langle proof \rangle$

`True /= a ==> not a`

lemma $neq{\cdot}TT{\cdot}a = neg{\cdot}a\ \langle proof \rangle$

`False /= a ==> a`

lemma $neq{\cdot}FF{\cdot}a = a\ \langle proof \rangle$

`not (isNothing x) ==> isJust x`

lemma $neg{\cdot}(isNothing{\cdot}x) = isJust{\cdot}x\ \langle proof \rangle$

`not (isJust x) ==> isNothing x`

lemma $neg{\cdot}(isJust{\cdot}x) = isNothing{\cdot}x\ \langle proof \rangle$

`x == Nothing ==> isNothing x`

lemma $eq{\cdot}x{\cdot}Nothing = isNothing{\cdot}x\ \langle proof \rangle$

`Nothing == x ==> isNothing x`

lemma $eq{\cdot}Nothing{\cdot}x = isNothing{\cdot}x\ \langle proof \rangle$

`x /= Nothing ==> Data.Maybe.isJust x`

lemma $neq{\cdot}x{\cdot}Nothing = isJust{\cdot}x\ \langle proof \rangle$

`Nothing /= x ==> Data.Maybe.isJust x`

lemma $neq{\cdot}Nothing{\cdot}x = isJust{\cdot}x\ \langle proof \rangle$

`(if isNothing x then y else fromJust x) ==> fromMaybe y x`

lemma $(If\ isNothing{\cdot}x\ then\ y\ else\ fromJust{\cdot}x) = fromMaybe{\cdot}y{\cdot}x\ \langle proof \rangle$

`(if isJust x then fromJust x else y) ==> fromMaybe y x`

lemma $(If\ isJust{\cdot}x\ then\ fromJust{\cdot}x\ else\ y) = fromMaybe{\cdot}y{\cdot}x\ \langle proof \rangle$

`(isJust x && (fromJust x == y)) ==> x == Just y`

lemma $(isJust{\cdot}x\ andalso\ (eq{\cdot}(fromJust{\cdot}x){\cdot}y)) = eq{\cdot}x{\cdot}(Just{\cdot}y)\ \langle proof \rangle$

`elem True ==> or`

lemma $elem{\cdot}TT = the\text{-}or$
$\langle proof \rangle$

`notElem False ==> and`

lemma $notElem{\cdot}FF = the\text{-}and$

⟨*proof*⟩

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

lemma *all*·(*neq*·(*a*::′*a*::*Eq-sym*)) = *notElem*·*a*
⟨*proof*⟩

```
maybe x id ==> Data.Maybe.fromMaybe x
```

lemma *maybe*·*x*·*ID* = *fromMaybe*·*x*
⟨*proof*⟩

```
maybe False (const True) ==> Data.Maybe.isJust
```

lemma *maybe*·*FF*·(*const*·*TT*) = *isJust*
⟨*proof*⟩

```
maybe True (const False) ==> Data.Maybe.isNothing
```

lemma *maybe*·*TT*·(*const*·*FF*) = *isNothing*
⟨*proof*⟩

```
maybe [] (: []) ==> maybeToList
```

lemma *maybe*·[]·(Λ *z*. *z* : []) = *maybeToList*
⟨*proof*⟩

```
catMaybes (map f x) ==> mapMaybe f x
```

lemma *catMaybes*·(*map*·*f*·*x*) = *mapMaybe*·*f*·*x* ⟨*proof*⟩

```
(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* ⟨*proof*⟩

```
(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* ⟨*proof*⟩

```
(map fromJust . filter isJust) ==> Data.Maybe.catMaybes
```

lemma (*map*·*fromJust oo filter*·*isJust*) = *catMaybes*
⟨*proof*⟩

```
concatMap (maybeToList . f) ==> Data.Maybe.mapMaybe f
```

lemma *concatMap*·(*maybeToList oo f*) = *mapMaybe*·*f*
⟨*proof*⟩

```
concatMap maybeToList ==> catMaybes
```

lemma *concatMap*·*maybeToList* = *catMaybes* ⟨*proof*⟩

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

lemma *mapMaybe*·*f*·(*map*·*g*·*x*) = *mapMaybe*·(*f oo g*)·*x* ⟨*proof*⟩

```
(($) . f) ==> f
```

lemma (*dollar oo f*) = *f* ⟨*proof*⟩

```
(f $) ==> f
```

lemma (Λ *z*. *dollar*·*f*·*z*) = *f* ⟨*proof*⟩

```
(\ a b -> g (f a) (f b)) ==> g `Data.Function.on` f
```

lemma (Λ *a b*. *g*·(*f*·*a*)·(*f*·*b*)) = *on*·*g*·*f* ⟨*proof*⟩

```
id $! x ==> x
```

lemma *dollarBang*·*ID*·*x* = *x* ⟨*proof*⟩

```
[x | x <- y] ==> y
```

lemma [*x* | *x* <− *y*] = *y* ⟨*proof*⟩

```
isPrefixOf (reverse x) (reverse y) ==> isSuffixOf x y
```

lemma *isPrefixOf*·(*reverse*·*x*)·(*reverse*·*y*) = *isSuffixOf*·*x*·*y* ⟨*proof*⟩

```
concat (intersperse x y) ==> intercalate x y
```

lemma *concat*·(*intersperse*·*x*·*y*) = *intercalate*·*x*·*y* ⟨*proof*⟩

```
x `seq` y ==> y
```

lemma
assumes *x* ≠ ⊥ **shows** *seq*·*x*·*y* = *y*
⟨*proof*⟩

```
f $! x ==> f x
```

lemma assumes *x* ≠ ⊥ **shows** *dollarBang*·*f*·*x* = *f*·*x*
⟨*proof*⟩

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

lemma *maybe*·(*f*·*x*)·(*f oo g*) ⊑ (*f oo maybe*·*x*·*g*)
⟨*proof*⟩

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.