Isabelle/HOLCF-Prelude

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

April 19, 2020

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

$-filter :: pttrn \Rightarrow 'a\ List.list \Rightarrow bool \Rightarrow 'a\ List.list\ ((1[-\leftarrow-\ ./\ -]))$

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*]:
$thenOrdering{\cdot}x{\cdot}y = LT \longleftrightarrow x = LT \lor x = EQ \land y = LT$
⟨*proof*⟩

lemma *thenOrdering-EQ-iff* [*simp*]:
$thenOrdering{\cdot}x{\cdot}y = EQ \longleftrightarrow x = EQ \land y = EQ$
⟨*proof*⟩

lemma *thenOrdering-GT-iff* [*simp*]:
$thenOrdering{\cdot}x{\cdot}y = GT \longleftrightarrow x = GT \lor x = EQ \land y = GT$
⟨*proof*⟩

lemma *thenOrdering-below-EQ-iff* [*simp*]:
$thenOrdering{\cdot}x{\cdot}y \sqsubseteq EQ \longleftrightarrow x \sqsubseteq EQ \land (x = \bot \lor y \sqsubseteq EQ)$
⟨*proof*⟩

lemma *is-EQ-thenOrdering* [*simp*]:
$is\text{-}EQ{\cdot}(thenOrdering{\cdot}x{\cdot}y) = (is\text{-}EQ{\cdot}x\ andalso\ is\text{-}EQ{\cdot}y)$
⟨*proof*⟩

lemma *oppOrdering-thenOrdering*:
$oppOrdering{\cdot}(thenOrdering{\cdot}x{\cdot}y) =$
$thenOrdering{\cdot}(oppOrdering{\cdot}x){\cdot}(oppOrdering{\cdot}y)$
⟨*proof*⟩

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

definition
$compare \equiv (\Lambda\ (Def\ x)\ (Def\ y).$
$if\ x < y\ then\ LT\ else\ if\ x > y\ then\ GT\ else\ EQ)$

instance ⟨*proof*⟩

end

lemma *lt-le*:
$lt{\cdot}(x::'a::Ord\text{-}linear){\cdot}y = (le{\cdot}x{\cdot}y\ andalso\ neq{\cdot}x{\cdot}y)$
⟨*proof*⟩

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

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

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

end

class *minus-cpo* = *minus* + *cpo* +
  assumes *cont-minus1*: *cont* (λx::$'a$::{*minus*,*cpo*}. $x - y$)
  assumes *cont-minus2*: *cont* (λy::$'a$::{*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* :: $'a \Rightarrow 'a \to 'a$ ('($-$-')) **where**
  $(-y) \equiv \Lambda\ x.\ x - y$

end

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

⟨*proof*⟩

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

instantiation *u* :: ({*zero*,*cpo*}) *zero*
begin
definition *zero-u* = $up\cdot(0::'a)$
instance ⟨*proof*⟩
end

instantiation *u* :: ({*one*,*cpo*}) *one*
begin
definition *one-u* = $up\cdot(1::'a)$
instance ⟨*proof*⟩
end

instantiation *u* :: (*plus-cpo*) *plus*
begin
definition *plus-u x y* = $(\Lambda(up\cdot a)\ (up\cdot b).\ up\cdot(a + b))\cdot x\cdot y$ **for** *x y* :: $'a_\bot$
instance ⟨*proof*⟩
end

instantiation *u* :: (*minus-cpo*) *minus*
begin
definition *minus-u x y* = $(\Lambda(up\cdot a)\ (up\cdot b).\ up\cdot(a - b))\cdot x\cdot y$ **for** *x y* :: $'a_\bot$
instance ⟨*proof*⟩
end

instantiation *u* :: (*times-cpo*) *times*
begin
definition *times-u x y* = $(\Lambda(up\cdot a)\ (up\cdot b).\ up\cdot(a * b))\cdot x\cdot y$ **for** *x y* :: $'a_\bot$
instance ⟨*proof*⟩
end

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

lemma *minus-u-strict* [*simp*]:
fixes *x* :: - *u* **shows** $x - \bot = \bot$ **and** $\bot - x = \bot$
⟨*proof*⟩

lemma *times-u-strict* [*simp*]:
fixes *x* :: - *u* **shows** $x * \bot = \bot$ **and** $\bot * x = \bot$
⟨*proof*⟩

lemma *plus-up-up* [*simp*]: $up{\cdot}x + up{\cdot}y = up{\cdot}(x + y)$
⟨*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* :: $('a \rightarrow 'b \rightarrow 'c) \rightarrow 'b \rightarrow 'a \rightarrow 'c$ **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* :: $('a \text{ −> } 'b) \text{ −> } 'a \text{ −> } 'b$ **where**
  $dollar \cdot f \cdot x = f \cdot x$

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

fixrec *on* :: $('b \text{ −> } 'b \text{ −> } 'c) \text{ −> } ('a \text{ −> } 'b) \text{ −> } 'a \text{ −> } 'a \text{ −> } 'c$ **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*]:
  $(x = \bot \Longrightarrow y = \bot) \Longrightarrow (y = \bot \Longrightarrow x = \bot) \Longrightarrow (x = TT \Longrightarrow y = TT) \Longrightarrow (y = TT \Longrightarrow x = TT) \Longrightarrow x = y$
⟨*proof*⟩

lemma *eq-tr-simps* [*simp*]:
  shows $eq \cdot TT \cdot TT = TT$ **and** $eq \cdot TT \cdot FF = FF$
    and $eq \cdot FF \cdot TT = FF$ **and** $eq \cdot FF \cdot FF = TT$
  ⟨*proof*⟩

Ord

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

5.2 Lemmas

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

lemma *andalso-eq-FF-iff* [*simp*]:

$(x\ andalso\ y) = FF \longleftrightarrow x = FF \lor (x = TT \land y = FF)$
⟨proof⟩

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

lemma *orelse-eq-FF-iff* [*simp*]:
$(x\ orelse\ y) = FF \longleftrightarrow x = FF \land y = FF$
⟨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* :: $('a \to 'b \to 'c) \to \langle 'a,\ 'b\rangle \to 'c$
  where $uncurry \cdot f \cdot p = f \cdot (fst \cdot p) \cdot (snd \cdot p)$

fixrec *curry* :: $(\langle 'a,\ 'b\rangle\ \to 'c) \to 'a \to 'b \to 'c$
  where $curry \cdot f \cdot a \cdot b = f \cdot \langle a,\ b\rangle$

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
  $eq = (\Lambda\ \langle\rangle\ \langle\rangle.\ TT)$

definition
  $compare = (\Lambda\ \langle\rangle\ \langle\rangle.\ EQ)$

instance
  ⟨*proof*⟩

end

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

definition
  $eq = (\Lambda\ \langle x1,\ y1\rangle\ \langle x2,\ y2\rangle.\ eq{\cdot}x1{\cdot}x2\ andalso\ eq{\cdot}y1{\cdot}y2)$

instance ⟨*proof*⟩

end

lemma *eq-Tuple2-simps* [*simp*]:
  $eq{\cdot}\langle x1,\ y1\rangle{\cdot}\langle x2,\ y2\rangle = (eq{\cdot}x1{\cdot}x2\ andalso\ eq{\cdot}y1{\cdot}y2)$
  ⟨*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
  $compare = (\Lambda\ \langle x1,\ y1\rangle\ \langle x2,\ y2\rangle.$
    $thenOrdering{\cdot}(compare{\cdot}x1{\cdot}x2){\cdot}(compare{\cdot}y1{\cdot}y2))$

instance
  ⟨*proof*⟩

end

lemma *compare-Tuple2-simps* [*simp*]:
  $compare{\cdot}\langle x1,\ y1\rangle{\cdot}\langle x2,\ y2\rangle = thenOrdering{\cdot}(compare{\cdot}x1{\cdot}x2){\cdot}(compare{\cdot}y1{\cdot}y2)$
  ⟨*proof*⟩

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

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

definition
  $eq = (\Lambda\ \langle x1,\ y1,\ z1\rangle\ \langle x2,\ y2,\ z2\rangle.$
    $eq\cdot x1\cdot x2\ andalso\ eq\cdot y1\cdot y2\ andalso\ eq\cdot z1\cdot z2)$

instance $\langle proof\rangle$

end

lemma *eq-Tuple3-simps* [*simp*]:
  $eq\cdot\langle x1,\ y1,\ z1\rangle\cdot\langle x2,\ y2,\ z2\rangle = (eq\cdot x1\cdot x2\ andalso\ eq\cdot y1\cdot y2\ andalso\ eq\cdot z1\cdot z2)$
  $\langle proof\rangle$

instance *Tuple3* :: (*Eq-sym*, *Eq-sym*, *Eq-sym*) *Eq-sym*
$\langle proof\rangle$

instance *Tuple3* :: (*Eq-equiv*, *Eq-equiv*, *Eq-equiv*) *Eq-equiv*
$\langle proof\rangle$

instance *Tuple3* :: (*Eq-eq*, *Eq-eq*, *Eq-eq*) *Eq-eq*
$\langle proof\rangle$

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

definition
  $compare = (\Lambda\ \langle x1,\ y1,\ z1\rangle\ \langle x2,\ y2,\ z2\rangle.$
   $thenOrdering\cdot(compare\cdot x1\cdot x2)\cdot(thenOrdering\cdot(compare\cdot y1\cdot y2)\cdot(compare\cdot z1\cdot z2)))$

instance
  $\langle proof\rangle$

end

lemma *compare-Tuple3-simps* [*simp*]:
  $compare\cdot\langle x1,\ y1,\ z1\rangle\cdot\langle x2,\ y2,\ z2\rangle =$
    $thenOrdering\cdot(compare\cdot x1\cdot x2)\cdot$
      $(thenOrdering\cdot(compare\cdot y1\cdot y2)\cdot(compare\cdot z1\cdot z2))$
  $\langle proof\rangle$

instance *Tuple3* :: (*Ord-linear*, *Ord-linear*, *Ord-linear*) *Ord-linear*
$\langle proof\rangle$

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 $0 = MkI \cdot 0$
definition $1 = MkI \cdot 1$
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** $x + y = \bot \longleftrightarrow x = \bot \lor y = \bot$
  ⟨*proof*⟩

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

lemma *mult-Integer-eq-bottom-iff* [*simp*]:
  fixes $x\ y$ :: *Integer* **shows** $x * y = \bot \longleftrightarrow x = \bot \lor y = \bot$
  ⟨*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*]: $(\bot{::}Integer) \neq$ *numeral* k
  ⟨*proof*⟩

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

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

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

instance ⟨*proof*⟩

end

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

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

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

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

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

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

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

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

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

$\langle proof \rangle$

lemma *eq-refl-Integer* [*simp*]:
$(x{::}Integer) \neq \bot \Longrightarrow eq{\cdot}x{\cdot}x = TT$
$\langle proof \rangle$

lemma *lt-refl-Integer* [*simp*]:
$(x{::}Integer) \neq \bot \Longrightarrow lt{\cdot}x{\cdot}x = FF$
$\langle proof \rangle$

lemma *le-refl-Integer* [*simp*]:
$(x{::}Integer) \neq \bot \Longrightarrow le{\cdot}x{\cdot}x = TT$
$\langle proof \rangle$

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

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

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

$\langle proof \rangle$

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

lemma *MkI-eq-0-iff* [*simp*]: $MkI \cdot n = 0 \longleftrightarrow n = 0$
$\langle proof \rangle$

lemma *MkI-eq-1-iff* [*simp*]: $MkI \cdot n = 1 \longleftrightarrow n = 1$
$\langle proof \rangle$

lemma *MkI-eq-numeral-iff* [*simp*]: $MkI \cdot n = numeral\ k \longleftrightarrow n = numeral\ k$
$\langle proof \rangle$

lemma *MkI-0*: $MkI \cdot 0 = 0$
$\langle proof \rangle$

lemma *MkI-1*: $MkI \cdot 1 = 1$
$\langle proof \rangle$

lemma *le-plus-1*:
fixes $m :: Integer$
assumes $le \cdot m \cdot n = TT$
shows $le \cdot m \cdot (n + 1) = TT$
$\langle proof \rangle$

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*: $le \cdot 0 \cdot i = TT$
and *zero*: $P\ 0$
and *step*: $\bigwedge i.\ le \cdot 1 \cdot i = TT \Longrightarrow P\ (i - 1) \Longrightarrow P\ i$
shows $P\ i$
$\langle proof \rangle$

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* → [*'a*] → [*'a*] ('(:'))
  -Cons-section-left :: *'a* ⇒ [*'a*] → [*'a*] ('(-:'))
translations
  (*x*:) == (*CONST Rep-cfun*) (*CONST Cons*) *x*

abbreviation *Cons-section-right* :: [*'a*] ⇒ *'a* → [*'a*] ('(:-')) **where**
  (:*xs*) ≡ Λ *x*. *x*:*xs*

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

abbreviation *null* :: [*'a*] → *tr* **where** *null* ≡ *is-Nil*

8.2 Haskell function definitions

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

fixrec *eq-list* :: [*'a*] → [*'a*] → *tr* **where**
  eq-list·[]·[] = *TT* |
  eq-list·(*x* : *xs*)·[] = *FF* |
  eq-list·[]·(*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*⟩

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 $compare\text{-}list :: ['a] \to ['a] \to Ordering$ **where**
$compare\text{-}list \cdot [] \cdot [] = EQ \mid$
$compare\text{-}list \cdot (x : xs) \cdot [] = GT \mid$
$compare\text{-}list \cdot [] \cdot (y : ys) = LT \mid$
$compare\text{-}list \cdot (x : xs) \cdot (y : ys) =$
$thenOrdering \cdot (compare \cdot x \cdot y) \cdot (compare\text{-}list \cdot xs \cdot ys)$

instance
⟨*proof*⟩

end

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

fixrec $zipWith :: ('a \to 'b \to 'c) \to ['a] \to ['b] \to ['c]$ **where**
$zipWith \cdot f \cdot (x : xs) \cdot (y : ys) = f \cdot x \cdot y : zipWith \cdot f \cdot xs \cdot ys \mid$
$zipWith \cdot f \cdot (x : xs) \cdot [] = [] \mid$
$zipWith \cdot f \cdot [] \cdot ys = []$

definition $zip :: ['a] \to ['b] \to [\langle 'a, 'b \rangle]$ **where**
$zip = zipWith \cdot \langle , \rangle$

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

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

fixrec $map :: ('a \to 'b) \to ['a] \to ['b]$ **where**

$map \cdot f \cdot [] = [] \mid$
$map \cdot f \cdot (x : xs) = f \cdot x : map \cdot f \cdot xs$

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

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

abbreviation $nth\text{-}syn :: ['a] \Rightarrow Integer \Rightarrow 'a$ (**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 $foldr$ :: $('a \rightarrow 'b \rightarrow 'b) \rightarrow 'b \rightarrow ['a] \rightarrow 'b$ **where**
$foldr{\cdot}f{\cdot}d{\cdot}[] = d$ |
$foldr{\cdot}f{\cdot}d{\cdot}(x : xs) = f{\cdot}x{\cdot}(foldr{\cdot}f{\cdot}d{\cdot}xs)$

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

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

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

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

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

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

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

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

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

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

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

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

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

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*]: $intsFrom \cdot x = seq \cdot x \cdot (x : intsFrom \cdot (x+1))$

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 ⟨*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**
  $intersperse{\cdot}sep{\cdot}[] = []$ |
  $intersperse{\cdot}sep{\cdot}[x] = [x]$ |
  $intersperse{\cdot}sep{\cdot}(x{:}y{:}xs) = x{:}sep{:}intersperse{\cdot}sep{\cdot}(y{:}xs)$

fixrec *intercalate* :: $['a] \to [['a]] \to ['a]$ **where**
  $intercalate{\cdot}xs{\cdot}xss = concat{\cdot}(intersperse{\cdot}xs{\cdot}xss)$

definition *replicate* :: $Integer \to 'a \to ['a]$ **where**
  $replicate = (\Lambda\ n\ x.\ take{\cdot}n{\cdot}(repeat{\cdot}x))$

definition *findIndices* :: $('a \to tr) \to ['a] \to [Integer]$ **where**
  $findIndices = (\Lambda\ P\ xs.$
    $map{\cdot}snd{\cdot}(filter{\cdot}(\Lambda\ \langle x,\ i\rangle.\ P{\cdot}x){\cdot}(zip{\cdot}xs{\cdot}[0..])))$

fixrec *length* :: $['a] \to Integer$ **where**
  $length{\cdot}[] = 0$ |
  $length{\cdot}(x : xs) = length{\cdot}xs + 1$

fixrec *delete* :: $'a{::}Eq \to ['a] \to ['a]$ **where**
  $delete{\cdot}x{\cdot}[] = []$ |
  $delete{\cdot}x{\cdot}(y : ys) = If\ eq{\cdot}x{\cdot}y\ then\ ys\ else\ y : delete{\cdot}x{\cdot}ys$

fixrec *diff* :: $['a{::}Eq] \to ['a] \to ['a]$ **where**
  $diff{\cdot}xs{\cdot}[] = xs$ |
  $diff{\cdot}xs{\cdot}(y : ys) = diff{\cdot}(delete{\cdot}y{\cdot}xs){\cdot}ys$

abbreviation *diff-syn* :: $['a{::}Eq] \Rightarrow ['a] \Rightarrow ['a]$ (**infixl** \\\\ *70*) **where**
  $xs \setminus\setminus ys \equiv diff{\cdot}xs{\cdot}ys$

8.3 Logical predicates on lists

inductive *finite-list* :: $['a] \Rightarrow bool$ **where**
  Nil [*intro*!, *simp*]: *finite-list* [] |
  Cons [*intro*!, *simp*]: $\bigwedge x\ xs.\ finite\text{-}list\ xs \Longrightarrow finite\text{-}list\ (x : xs)$

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

lemma *finite-list-upwards*:
  assumes *finite-list xs* **and** $xs \sqsubseteq ys$
  shows *finite-list ys*
⟨*proof*⟩

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

lemma *bot-not-finite-list* [*simp*]:
  $finite\text{-}list\ \bot = False$

$\langle proof \rangle$

inductive *listmem* :: $'a \Rightarrow ['a] \Rightarrow bool$ **where**
listmem x $(x : xs)$ |
listmem x $xs \Longrightarrow$ *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
$\langle proof \rangle$

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)
$\langle proof \rangle$

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*]:
$map{\cdot}P{\cdot}\bot = \bot$
$\langle proof \rangle$

lemma *map-ID* [*simp*]:
$map{\cdot}ID{\cdot}xs = xs$
$\langle proof \rangle$

lemma *enumFrom-intsFrom-conv* [*simp*]:
enumFrom = *intsFrom*
$\langle proof \rangle$

lemma *enumFromTo-upto-conv* [*simp*]:
enumFromTo = *upto*
$\langle proof \rangle$

lemma *zipWith-strict* [*simp*]:
$zipWith{\cdot}f{\cdot}\bot{\cdot}ys = \bot$
$zipWith{\cdot}f{\cdot}(x : xs){\cdot}\bot = \bot$
$\langle proof \rangle$

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

$zip\cdot(x : xs)\cdot\bot = \bot$
$zip\cdot[]\cdot ys = []$
$zip\cdot\bot\cdot ys = \bot$
$\langle proof \rangle$

lemma *zip-Nil2* [*simp*]:
$xs \neq \bot \Longrightarrow zip\cdot xs\cdot[] = []$
$\langle proof \rangle$

lemma *nth-strict* [*simp*]:
$nth\cdot\bot\cdot n = \bot$
$nth\cdot xs\cdot\bot = \bot$
$\langle proof \rangle$

lemma *upto-strict* [*simp*]:
$upto\cdot\bot\cdot y = \bot$
$upto\cdot x\cdot\bot = \bot$
$\langle proof \rangle$

lemma *upto-simps* [*simp*]:
$n < m \Longrightarrow upto\cdot(MkI\cdot m)\cdot(MkI\cdot n) = []$
$m \leq n \Longrightarrow upto\cdot(MkI\cdot m)\cdot(MkI\cdot n) = MkI\cdot m : [MkI\cdot m+1..MkI\cdot n]$
$\langle proof \rangle$

lemma *filter-strict* [*simp*]:
$filter\cdot P\cdot\bot = \bot$
$\langle proof \rangle$

lemma *nth-Cons-simp* [*simp*]:
$eq\cdot n\cdot 0 = TT \Longrightarrow nth\cdot(x : xs)\cdot n = x$
$eq\cdot n\cdot 0 = FF \Longrightarrow nth\cdot(x : xs)\cdot n = nth\cdot xs\cdot(n - 1)$
$\langle proof \rangle$

lemma *nth-Cons-split*:

$$\begin{aligned} P\ (nth\cdot(x : xs)\cdot n) = (&(eq\cdot n\cdot 0 = FF \longrightarrow P\ (nth\cdot(x : xs)\cdot n)) \land \\ &(eq\cdot n\cdot 0 = TT \longrightarrow P\ (nth\cdot(x : xs)\cdot n)) \land \\ &(n = \bot \longrightarrow P\ (nth\cdot(x : xs)\cdot n))) \end{aligned}$$

$\langle proof \rangle$

lemma *nth-Cons-numeral* [*simp*]:
$(x : xs)\ !!\ 0 = x$
$(x : xs)\ !!\ 1 = xs\ !!\ 0$
$(x : xs)\ !!\ numeral\ (Num.Bit0\ k) = xs\ !!\ numeral\ (Num.BitM\ k)$
$(x : xs)\ !!\ numeral\ (Num.Bit1\ k) = xs\ !!\ numeral\ (Num.Bit0\ k)$
$\langle proof \rangle$

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

lemma *take-strict-2* [*simp*]:
$le\cdot 1\cdot i = TT \Longrightarrow take\cdot i\cdot\bot = \bot$
⟨*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*]:
$\perp ++\ ys = \perp$
⟨*proof*⟩

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

lemma *foldl-strict* [*simp*]:
$foldl \cdot f \cdot d \cdot \perp = \perp$
$foldl \cdot f \cdot \perp \cdot [] = \perp$
⟨*proof*⟩

lemma *foldr1-strict* [*simp*]:
$foldr1 \cdot f \cdot \perp = \perp$
$foldr1 \cdot f \cdot (x:\perp) = \perp$
⟨*proof*⟩

lemma *foldl1-strict* [*simp*]:
$foldl1 \cdot f \cdot \perp = \perp$
⟨*proof*⟩

lemma *foldl-spine-strict*:
$\neg$ *finite-list xs* $\Longrightarrow foldl \cdot f \cdot x \cdot xs = \perp$
⟨*proof*⟩

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 $foldl \cdot f \cdot z \cdot xs = foldr \cdot f \cdot z \cdot xs$
⟨*proof*⟩

lemma *elem-strict* [*simp*]:
$elem \cdot x \cdot \perp = \perp$
⟨*proof*⟩

lemma *notElem-strict* [*simp*]:
$notElem \cdot x \cdot \perp = \perp$
⟨*proof*⟩

lemma *list-eq-nil*[*simp*]:
$eq \cdot l \cdot [] = TT \longleftrightarrow l = []$
$eq \cdot [] \cdot l = TT \longleftrightarrow l = []$
⟨*proof*⟩

lemma *take-Nil* [*simp*]:
$n \neq \bot \Longrightarrow take{\cdot}n{\cdot}[] = []$
⟨*proof*⟩

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

lemma *take-Cons* [*simp*]:
$le{\cdot}1{\cdot}i = TT \Longrightarrow take{\cdot}i{\cdot}(x{:}xs) = x : take{\cdot}(i - 1){\cdot}xs$
⟨*proof*⟩

lemma *take-MkI-Cons* [*simp*]:
$0 < n \Longrightarrow take{\cdot}(MkI{\cdot}n){\cdot}(x : xs) = x : take{\cdot}(MkI{\cdot}(n - 1)){\cdot}xs$
⟨*proof*⟩

lemma *take-numeral-Cons* [*simp*]:
$take{\cdot}1{\cdot}(x : xs) = [x]$
$take{\cdot}(numeral\ (Num.Bit0\ k)){\cdot}(x : xs) = x : take{\cdot}(numeral\ (Num.BitM\ k)){\cdot}xs$
$take{\cdot}(numeral\ (Num.Bit1\ k)){\cdot}(x : xs) = x : take{\cdot}(numeral\ (Num.Bit0\ k)){\cdot}xs$
⟨*proof*⟩

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

lemma *drop-pos* [*simp*]:
$le{\cdot}n{\cdot}0 = FF \Longrightarrow drop{\cdot}n{\cdot}xs = (case\ xs\ of\ [] \Rightarrow [] \mid y : ys \Rightarrow drop{\cdot}(n - 1){\cdot}ys)$
⟨*proof*⟩

lemma *drop-numeral-Cons* [*simp*]:
$drop{\cdot}1{\cdot}(x : xs) = xs$
$drop{\cdot}(numeral\ (Num.Bit0\ k)){\cdot}(x : xs) = drop{\cdot}(numeral\ (Num.BitM\ k)){\cdot}xs$
$drop{\cdot}(numeral\ (Num.Bit1\ k)){\cdot}(x : xs) = drop{\cdot}(numeral\ (Num.Bit0\ k)){\cdot}xs$
⟨*proof*⟩

lemma *take-drop-append*:
$take{\cdot}(MkI{\cdot}i){\cdot}xs\ {+}{+}\ drop{\cdot}(MkI{\cdot}i){\cdot}xs = xs$
⟨*proof*⟩

lemma *take-intsFrom-enumFrom* [*simp*]:
$take{\cdot}(MkI{\cdot}n){\cdot}[MkI{\cdot}i..] = [MkI{\cdot}i..MkI{\cdot}(n{+}i) - 1]$
⟨*proof*⟩

lemma *drop-intsFrom-enumFrom* [*simp*]:
assumes $n \geq 0$

shows $drop\cdot(MkI\cdot n)\cdot[MkI\cdot i..] = [MkI\cdot(n+i)..]$
⟨proof⟩

lemma *last-append-singleton*:
$finite\text{-}list\ xs \Longrightarrow last\cdot(xs\ {+}{+}\ [x]) = x$
⟨proof⟩

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

lemma *append-Nil2* [*simp*]:
$xs\ {+}{+}\ [] = xs$
⟨proof⟩

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

lemma *concat-simps* [*simp*]:
$concat\cdot[] = []$
$concat\cdot(xs : xss) = xs\ {+}{+}\ concat\cdot xss$
$concat\cdot\bot = \bot$
⟨proof⟩

lemma *concatMap-simps* [*simp*]:
$concatMap\cdot f\cdot[] = []$
$concatMap\cdot f\cdot(x : xs) = f\cdot x\ {+}{+}\ concatMap\cdot f\cdot xs$
$concatMap\cdot f\cdot\bot = \bot$
⟨proof⟩

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

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

lemma *filter-filter* [*simp*]:
$filter\cdot P\cdot(filter\cdot Q\cdot xs) = filter\cdot(\Lambda\ x.\ Q\cdot x\ andalso\ P\cdot x)\cdot xs$
⟨proof⟩

lemma *filter-const-TT* [*simp*]:
$filter\cdot(\Lambda\ \text{-}.\ TT)\cdot xs = xs$
⟨proof⟩

lemma *tails-strict* [*simp*]:
$tails\cdot\bot = \bot$

⟨*proof*⟩

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

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

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

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

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

lemma *tails-neq-Nil* [*simp*]:
$tails \cdot xs \neq []$
⟨*proof*⟩

lemma *inits-neq-Nil* [*simp*]:
$inits \cdot xs \neq []$
⟨*proof*⟩

lemma *Nil-neq-tails* [*simp*]:
$[] \neq tails \cdot xs$
⟨*proof*⟩

lemma *Nil-neq-inits* [*simp*]:
$[] \neq inits \cdot xs$
⟨*proof*⟩

lemma *finite-list-not-bottom* [*simp*]:
assumes *finite-list xs* **shows** $xs \neq \bot$
⟨*proof*⟩

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

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

lemma *filter-TT* [*simp*]:
  assumes $\forall x{\in}set\ xs.\ P{\cdot}x = TT$
  shows $filter{\cdot}P{\cdot}xs = xs$
  ⟨*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 $m \le n$
  shows $upto{\cdot}(MkI{\cdot}m){\cdot}(MkI{\cdot}n) \mathbin{++} intsFrom{\cdot}(MkI{\cdot}n{+}1) = intsFrom{\cdot}(MkI{\cdot}m)$
    (**is** $?u\ m\ n \mathbin{++} \text{-} = ?i\ m$)
⟨*proof*⟩

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

lemma *Nil-append-iff* [*iff*]:
  $xs \mathbin{++} ys = [] \longleftrightarrow xs = [] \land ys = []$
  ⟨*proof*⟩

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

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

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

⟨*ML*⟩

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

lemma *distinct-Cons* [*simp*]:
  distinct (*x* : *xs*) ⟷ *distinct xs* ∧ *x* ∉ *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 $set\ (xs \setminus\setminus\ ys) = set\ xs - set\ ys$
⟨*proof*⟩

lemma *distinct-delete-filter*:
fixes $xs :: [{}'a{::}Eq\text{-}eq]$
assumes *distinct xs*
and $\forall\, x{\in}set\ xs.\ eq{\cdot}a{\cdot}x \neq \bot$
shows $delete{\cdot}a{\cdot}xs = filter{\cdot}(\Lambda\ x.\ neq{\cdot}a{\cdot}x){\cdot}xs$
⟨*proof*⟩

lemma *distinct-diff-filter*:
fixes $xs\ ys :: [{}'a{::}Eq\text{-}eq]$
assumes *finite-list ys*
and *distinct xs*
and $\forall\, a{\in}set\ ys.\ \forall\, x{\in}set\ xs.\ eq{\cdot}a{\cdot}x \neq \bot$
shows $xs \setminus\setminus\ ys = filter{\cdot}(\Lambda\ x.\ neg{\cdot}(elem{\cdot}x{\cdot}ys)){\cdot}xs$
⟨*proof*⟩

lemma *distinct-upto* [*intro*, *simp*]:
$distinct\ [MkI{\cdot}m..MkI{\cdot}n]$
⟨*proof*⟩

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

lemma *If-eq-bottom-iff* [*simp*]:
$(If\ b\ then\ x\ else\ y = \bot) \longleftrightarrow b = \bot \lor b = TT \land x = \bot \lor b = FF \land y = \bot$
⟨*proof*⟩

lemma *upto-eq-bottom-iff* [*simp*]:
$upto{\cdot}m{\cdot}n = \bot \longleftrightarrow m = \bot \lor n = \bot$
⟨*proof*⟩

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

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

lemma *intsFrom-split*:
assumes $m \geq n$
shows $[MkI{\cdot}n..] = [MkI{\cdot}n\ ..\ MkI{\cdot}(m - 1)] +\!+ [MkI{\cdot}m..]$
⟨*proof*⟩

lemma *filter-fast-forward*:

assumes $n+1 \leq n'$
and $\forall k . n < k \longrightarrow k < n' \longrightarrow \neg P k$
shows $filter\cdot(\Lambda (MkI\cdot i) . Def (P i))\cdot[MkI\cdot(n+1)..] = filter\cdot(\Lambda (MkI\cdot i) . Def (P i))\cdot[MkI\cdot n'..]$
$\langle proof \rangle$

lemma *null-eq-TT-iff* [*simp*]:
$null\cdot xs = TT \longleftrightarrow xs = []$
$\langle proof \rangle$

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

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

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

lemma *length-strict* [*simp*]:
$length\cdot\bot = \bot$
$\langle proof \rangle$

lemma *repeat-neq-bottom* [*simp*]:
$repeat\cdot x \neq \bot$
$\langle proof \rangle$

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

lemma *length-append* [*simp*]:
$length\cdot(xs ++ ys) = length\cdot xs + length\cdot ys$
$\langle proof \rangle$

lemma *replicate-strict* [*simp*]:
$replicate\cdot\bot\cdot x = \bot$
$\langle proof \rangle$

lemma *replicate-0* [*simp*]:
$replicate\cdot 0\cdot x = []$
$replicate\cdot(MkI\cdot 0)\cdot xs = []$
$\langle proof \rangle$

lemma *Integer-add-0* [*simp*]: $MkI\cdot 0 + n = n$
$\langle proof \rangle$

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

lemma *replicate-append-plus-conv*:
assumes $0 \leq m$ **and** $0 \leq n$
shows $replicate\cdot(MkI\cdot m)\cdot x \mathbin{++} replicate\cdot(MkI\cdot n)\cdot x$
$= replicate\cdot(MkI\cdot m + MkI\cdot n)\cdot x$
$\langle proof \rangle$

lemma *replicate-MkI-1* [*simp*]:
$replicate\cdot(MkI\cdot 1)\cdot x = x : []$
$\langle proof \rangle$

lemma *length-replicate* [*simp*]:
assumes $0 \leq n$
shows $length\cdot(replicate\cdot(MkI\cdot n)\cdot x) = MkI\cdot n$
$\langle proof \rangle$

lemma *map-oo* [*simp*]:
$map\cdot f\cdot(map\cdot g\cdot xs) = map\cdot(f \; oo \; g)\cdot xs$
$\langle proof \rangle$

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

lemma *map-plus-intsFrom*:
$map\cdot(+ \; MkI\cdot n)\cdot(intsFrom\cdot(MkI\cdot m)) = intsFrom\cdot(MkI\cdot(m+n))$ (**is** $?l = ?r$)
$\langle proof \rangle$

lemma *plus-eq-MkI-conv*:
$l + n = MkI\cdot m \longleftrightarrow (\exists\, l'\; n'.\; l = MkI\cdot l' \wedge n = MkI\cdot n' \wedge m = l' + n')$
$\langle proof \rangle$

lemma *length-ge-0*:
$length\cdot xs = MkI\cdot n \Longrightarrow n \geq 0$
$\langle proof \rangle$

lemma *length-0-conv* [*simp*]:
$length\cdot xs = MkI\cdot 0 \longleftrightarrow xs = []$
$\langle proof \rangle$

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** $?l = ?r$)

$\langle proof \rangle$

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

lemma *nth-append*:
  assumes $length{\cdot}xs = MkI{\cdot}n$ **and** $n \leq m$
  shows $(xs\ {+}{+}\ ys)\ !!\ MkI{\cdot}m = ys\ !!\ MkI{\cdot}(m - n)$
  $\langle proof \rangle$

lemma *replicate-nth* [*simp*]:
  assumes $0 \leq n$
  shows $(replicate{\cdot}(MkI{\cdot}n){\cdot}x\ {+}{+}\ xs)\ !!\ MkI{\cdot}n = xs\ !!\ MkI{\cdot}0$
  $\langle proof \rangle$

lemma *map2-zip*:
  $map{\cdot}(\Lambda\langle x,\ y\rangle.\ \langle x,\ f{\cdot}y\rangle){\cdot}(zip{\cdot}xs{\cdot}ys) = zip{\cdot}xs{\cdot}(map{\cdot}f{\cdot}ys)$
  $\langle proof \rangle$

lemma *map2-filter*:
  $map{\cdot}(\Lambda\langle x,\ y\rangle.\ \langle x,\ f{\cdot}y\rangle){\cdot}(filter{\cdot}(\Lambda\langle x,\ y\rangle.\ P{\cdot}x){\cdot}xs)$
    $= filter{\cdot}(\Lambda\langle x,\ y\rangle.\ P{\cdot}x){\cdot}(map{\cdot}(\Lambda\langle x,\ y\rangle.\ \langle x,\ f{\cdot}y\rangle){\cdot}xs)$
  $\langle proof \rangle$

lemma *map-map-snd*:
  $f{\cdot}\bot = \bot \Longrightarrow map{\cdot}f{\cdot}(map{\cdot}snd{\cdot}xs)$
    $= map{\cdot}snd{\cdot}(map{\cdot}(\Lambda\langle x,\ y\rangle.\ \langle x,\ f{\cdot}y\rangle){\cdot}xs)$
  $\langle proof \rangle$

lemma *findIndices-Cons* [*simp*]:
  $findIndices{\cdot}P{\cdot}(a : xs) =$
    $If\ P{\cdot}a\ then\ 0 : map{\cdot}(+1){\cdot}(findIndices{\cdot}P{\cdot}xs)$
    $else\ map{\cdot}(+1){\cdot}(findIndices{\cdot}P{\cdot}xs)$
  $\langle proof \rangle$

lemma *filter-alt-def*:
  fixes $xs :: [{'a}]$
  shows $filter{\cdot}P{\cdot}xs = map{\cdot}(nth{\cdot}xs){\cdot}(findIndices{\cdot}P{\cdot}xs)$
$\langle proof \rangle$

abbreviation *cfun-image* :: $({'a} \to {'b}) \Rightarrow {'a}\ set \Rightarrow {'b}\ set$ (**infixr** ‘$\cdot$ *90*) **where**
  f ‘$\cdot$ $A \equiv Rep\text{-}cfun\ f$ ‘ A

lemma *set-map*:
  $set\ (map{\cdot}f{\cdot}xs) = f$ ‘$\cdot$ $set\ xs$ (**is** $?l = ?r$)
$\langle proof \rangle$

8.5 *reverse* and *reverse* induction

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

lemma *reverse-Nil* [*simp*]:
  $reverse \cdot [] = []$
  ⟨*proof*⟩

lemma *reverse-singleton* [*simp*]:
  $reverse \cdot [x] = [x]$
  ⟨*proof*⟩

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

lemma *foldl-flip-Cons-append*:
  $foldl \cdot (flip \cdot (:)) \cdot ys \cdot xs = foldl \cdot (flip \cdot (:)) \cdot [] \cdot xs \mathbin{++} ys$
⟨*proof*⟩

lemma *reverse-Cons* [*simp*]:
  $reverse \cdot (x{:}xs) = reverse \cdot xs \mathbin{++} [x]$
  ⟨*proof*⟩

lemma *reverse-append-below*:
  $reverse \cdot (xs \mathbin{++} ys) \sqsubseteq reverse \cdot ys \mathbin{++} reverse \cdot xs$
  ⟨*proof*⟩

lemma *reverse-reverse-below*:
  $reverse \cdot (reverse \cdot xs) \sqsubseteq xs$
⟨*proof*⟩

lemma *reverse-append* [*simp*]:
  assumes *finite-list xs*
  shows $reverse \cdot (xs \mathbin{++} ys) = reverse \cdot ys \mathbin{++} reverse \cdot xs$
  ⟨*proof*⟩

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

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

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

lemma *reverse-induct* [*consumes 1*, *case-names Nil snoc*]:
$[\![$*finite-list xs*; P []; $\bigwedge x\ xs$. *finite-list xs* $\Longrightarrow P\ xs \Longrightarrow P\ (xs\ {+}{+}\ [x])]\!] \Longrightarrow P\ xs$
⟨*proof*⟩

lemma *length-plus-not-0*:
$le{\cdot}1{\cdot}n = TT \Longrightarrow le{\cdot}(length{\cdot}xs + n){\cdot}0 = TT \Longrightarrow False$
⟨*proof*⟩

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

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

lemma *eq-take-length-isPrefixOf*:
$eq{\cdot}xs{\cdot}(take{\cdot}(length{\cdot}xs){\cdot}ys) \sqsubseteq isPrefixOf{\cdot}xs{\cdot}ys$
⟨*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* :: $'b \to ('a \to 'b) \to 'a$ *Maybe* $\to 'b$ **where**
maybe ≡ *Maybe-case*

fixrec *isJust* :: $'a$ *Maybe* → *tr* **where**
$isJust{\cdot}(Just{\cdot}a) = TT$ |
$isJust{\cdot}Nothing = FF$

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

fixrec *fromJust* :: $'a$ *Maybe* → $'a$ **where**
$fromJust{\cdot}(Just{\cdot}a) = a$ |
$fromJust{\cdot}Nothing = \bot$

fixrec *fromMaybe* :: $'a \to 'a$ *Maybe* → $'a$ **where**

$fromMaybe \cdot d \cdot Nothing = d$ |
$fromMaybe \cdot d \cdot (Just \cdot a) = a$

fixrec $maybeToList :: 'a\ Maybe \to ['a]$ **where**
$maybeToList \cdot Nothing = []$ |
$maybeToList \cdot (Just \cdot a) = [a]$

fixrec $listToMaybe :: ['a] \to 'a\ Maybe$ **where**
$listToMaybe \cdot [] = Nothing$ |
$listToMaybe \cdot (a{:}\text{-}) = Just \cdot a$

fixrec $catMaybes :: ['a\ Maybe] \to ['a]$ **where**
$catMaybes = concatMap \cdot maybeToList$

fixrec $mapMaybe :: ('a \to 'b\ Maybe) \to ['a] \to ['b]$ **where**
$mapMaybe \cdot f = catMaybes\ oo\ map \cdot f$

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

definition
$eq = maybe \cdot (maybe \cdot TT \cdot (\Lambda\ y.\ FF)) \cdot (\Lambda\ x.\ maybe \cdot FF \cdot (\Lambda\ y.\ eq \cdot x \cdot y))$

instance ⟨*proof*⟩

end

lemma *eq-Maybe-simps* [*simp*]:
$eq \cdot Nothing \cdot Nothing = TT$
$eq \cdot Nothing \cdot (Just \cdot y) = FF$
$eq \cdot (Just \cdot x) \cdot Nothing = FF$
$eq \cdot (Just \cdot x) \cdot (Just \cdot y) = eq \cdot x \cdot y$
⟨*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
$compare = maybe \cdot (maybe \cdot EQ \cdot (\Lambda\ y.\ LT)) \cdot (\Lambda\ x.\ maybe \cdot GT \cdot (\Lambda\ y.\ compare \cdot x \cdot y))$

instance ⟨*proof*⟩

end

lemma *compare-Maybe-simps* [*simp*]:
  $compare \cdot Nothing \cdot Nothing = EQ$
  $compare \cdot Nothing \cdot (Just \cdot y) = LT$
  $compare \cdot (Just \cdot x) \cdot Nothing = GT$
  $compare \cdot (Just \cdot x) \cdot (Just \cdot y) = compare \cdot x \cdot y$
  ⟨*proof*⟩

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

lemma *isJust-strict* [*simp*]: $isJust \cdot \bot = \bot$ ⟨*proof*⟩
lemma *fromMaybe-strict* [*simp*]: $fromMaybe \cdot x \cdot \bot = \bot$ ⟨*proof*⟩
lemma *maybeToList-strict* [*simp*]: $maybeToList \cdot \bot = \bot$ ⟨*proof*⟩

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 $x = (x \neq \bot)$

lemma *defined-bottom* [*simp*]: ¬ *defined* ⊥
  ⟨*proof*⟩

lemma *defined-seq* [*simp*]: *defined* $x \Longrightarrow seq \cdot x \cdot y = y$
  ⟨*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*]: $[\![\ numeral\ x :: Integer\ ]\!] = numeral\ x$

$\langle proof \rangle$

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

lemma *defined-Integer-minus* [*simp*]: $defined\ i \Longrightarrow defined\ j \Longrightarrow defined\ (i - (j{::}Integer))$
$\langle proof \rangle$

lemma *val-Integer-minus* [*simp*]: $defined\ i \Longrightarrow defined\ j \Longrightarrow [\![\ i - j\ ]\!] = [\![\ i\ ]\!] - [\![\ j\ ]\!]$
$\langle proof \rangle$

lemma *defined-Integer-plus* [*simp*]: $defined\ i \Longrightarrow defined\ j \Longrightarrow defined\ (i + (j{::}Integer))$
$\langle proof \rangle$

lemma *val-Integer-plus* [*simp*]: $defined\ i \Longrightarrow defined\ j \Longrightarrow [\![\ i + j\ ]\!] = [\![\ i\ ]\!] + [\![\ j\ ]\!]$
$\langle proof \rangle$

lemma *defined-Integer-eq* [*simp*]: $defined\ (eq{\cdot}a{\cdot}b) \longleftrightarrow defined\ a \wedge defined\ (b{::}Integer)$
$\langle proof \rangle$

lemma *val-Integer-eq* [*simp*]: $defined\ a \Longrightarrow defined\ b \Longrightarrow [\![\ eq{\cdot}a{\cdot}b\ ]\!] = ([\![\ a\ ]\!] = ([\![\ b\ ]\!] :: int))$
$\langle proof \rangle$

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*: $\bigwedge\ i.\ defined\ i \Longrightarrow 0 \leq [\![i]\!] \Longrightarrow (\bigwedge\ j.\ defined\ j \Longrightarrow [\![\ j\ ]\!] < [\![\ i\ ]\!] \Longrightarrow P\ j) \Longrightarrow P\ i$
shows $P\ (i{::}Integer)$
$\langle proof \rangle$

Some list lemmas re-done with the new setup.

lemma *nth-tail*:
$defined\ n \Longrightarrow [\![\ n\ ]\!] \geq 0 \Longrightarrow tail{\cdot}xs\ !!\ n = xs\ !!\ (1 + n)$
$\langle proof \rangle$

lemma *nth-zipWith*:
assumes *f1* [*simp*]: $\bigwedge y.\ f{\cdot}\bot{\cdot}y = \bot$
assumes *f2* [*simp*]: $\bigwedge x.\ f{\cdot}x{\cdot}\bot = \bot$
shows $zipWith{\cdot}f{\cdot}xs{\cdot}ys\ !!\ n = f{\cdot}(xs\ !!\ n){\cdot}(ys\ !!\ n)$
$\langle proof \rangle$

lemma *nth-neg* [*simp*]: *defined n* $\Longrightarrow$ $[\![\ n\ ]\!] < 0 \Longrightarrow nth{\cdot}xs{\cdot}n = \bot$
⟨*proof*⟩

lemma *nth-Cons-simp* [*simp*]:
  defined n $\Longrightarrow$ $[\![\ n\ ]\!] = 0 \Longrightarrow nth{\cdot}(x : xs){\cdot}n = x$
  defined n $\Longrightarrow$ $[\![\ n\ ]\!] > 0 \Longrightarrow nth{\cdot}(x : xs){\cdot}n = nth{\cdot}xs{\cdot}(n - 1)$
⟨*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*]:
  $concatMap{\cdot}(\Lambda\ x.\ [f{\cdot}x]){\cdot}xs = map{\cdot}f{\cdot}xs$
  ⟨*proof*⟩

lemma *listcompr-filter* [*simp*]:
  $[x \mid x <- xs,\ P{\cdot}x] = filter{\cdot}P{\cdot}xs$

⟨*proof*⟩

lemma $[y \mid let\ y = x{*}2;\ z = y,\ x <- xs] = A$
  ⟨*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* = (Λ (*MkI*·*x*) . *MkI*·(|*x*|))
definition *signum* = (Λ (*MkI*·*x*) . *MkI*·(*sgn x*))
definition *fromInteger* = (Λ *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*]: $fibs = 0 : 1 : zipWith{\cdot}(+){\cdot}fibs{\cdot}(tail{\cdot}fibs)$

fun *fib* :: $int \Rightarrow int$ **where**
 $fib\ n = (if\ n \leq 0\ then\ 0\ else\ if\ n = 1\ then\ 1\ else\ fib\ (n - 1) + fib\ (n - 2))$

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** $[\![\ i\ ]\!] \geq 0$ **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

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*
⟨*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 $\exists\ n'.\ n' > n \wedge prime\ n' \wedge (\forall k.\ n < k \longrightarrow k < n' \longrightarrow \neg\ prime\ k)$
  $\langle proof \rangle$

Simplification for andalso:

lemma *andAlso-Def* [*simp*]: $((Def\ x)\ andalso\ (Def\ y)) = Def\ (x \wedge y)$
  $\langle proof \rangle$

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

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

lemma *prim-bisim-is-bisim*: *list-bisim prim-bisim*
$\langle proof \rangle$

Now we apply coinduction:

lemma *sieve-produces-primes*:
  fixes $n :: nat$
  assumes $prime\ n$
  shows $sieve\cdot(filter\cdot(\Lambda\ (MkI\cdot i).\ Def\ ((\forall d{::}int.\ d > 1 \longrightarrow d < n \longrightarrow \neg\ (d\ dvd\ i))))\cdot[MkI\cdot n..])$
    $= filter\cdot(\Lambda\ (MkI\cdot i).\ Def\ (prime\ (nat\ |i|)))\cdot[MkI\cdot n..]$
  $\langle proof \rangle$

And finally show the correctness of primes.

theorem *primes*:
  shows $primes = filter\cdot(\Lambda\ (MkI\cdot i).\ Def\ (prime\ (nat\ |i|)))\cdot[MkI\cdot 2..]$
$\langle proof \rangle$

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 $\equiv$ (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* ((++) (*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*]:
$cont\ (\lambda g.\ \Lambda\ h.\ Abs\text{-}listfun\ (\Lambda\ c\ n.\ listfun\cdot g\cdot c\cdot(listfun\cdot h\cdot c\cdot n),\ build\cdot g\ {+}{+}\ build\cdot h))$
⟨*proof*⟩

lemma *abstract-list-build-append*:
$abstract\text{-}list\ (build\cdot g\ {+}{+}\ build\cdot h) = (\Lambda\ c\ n.\ listfun\cdot g\cdot c\cdot(listfun\cdot h\cdot c\cdot n))$
⟨*proof*⟩

lemma *augment/build*:
$augment\cdot g\cdot(build\cdot h) = build\cdot(g \circ lf\ h)$
⟨*proof*⟩

lemma *augment/nil*:
$augment\cdot g\cdot[] = build\cdot g$
⟨*proof*⟩

lemma *build-listfun-comp* [*simp*]:
$build\cdot(g \circ lf\ h) = build\cdot g\ {+}{+}\ build\cdot h$
⟨*proof*⟩

lemma *augment-augment*:
$augment\cdot g\cdot(augment\cdot h\cdot xs) = augment\cdot(g \circ lf\ h)\cdot xs$
⟨*proof*⟩

lemma *abstract-list-map* [*simp*]:
$abstract\text{-}list\ (map\cdot f\cdot xs) = (\Lambda\ c\ n.\ foldr\cdot(mapFB\cdot c\cdot f)\cdot n\cdot xs)$
⟨*proof*⟩

lemma *map*:
$map\cdot f\cdot xs = build\cdot(Abs\text{-}listfun\ (\Lambda\ c\ n.\ foldr\cdot(mapFB\cdot c\cdot f)\cdot n\cdot xs,\ map\cdot f\cdot xs))$
⟨*proof*⟩

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

lemma *mapFB*:
$mapFB\cdot(mapFB\cdot c\cdot f)\cdot g = mapFB\cdot c\cdot(f\ oo\ g)$
⟨*proof*⟩

lemma ++:
$xs\ {+}{+}\ ys = augment\cdot(Abs\text{-}listfun\ (abstract\text{-}list\ xs,\ xs))\cdot ys$
⟨*proof*⟩

15.3 Examples

fixrec $sum :: [Integer] \to Integer$ **where**
$sum\cdot xs = foldr\cdot(+)\cdot 0\cdot xs$

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 $concat{\cdot}(map{\cdot}f{\cdot}x) = concatMap{\cdot}f{\cdot}x$
⟨*proof*⟩

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

lemma $concat{\cdot}[a,\ b] = a\ {+}{+}\ b$
⟨*proof*⟩

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

lemma $map{\cdot}f{\cdot}(map{\cdot}g{\cdot}x) = map{\cdot}(f\ oo\ g){\cdot}x$
⟨*proof*⟩

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

lemma $x\ !!\ 0 = head{\cdot}x$
⟨*proof*⟩

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

lemma $take{\cdot}n{\cdot}(repeat{\cdot}x) = replicate{\cdot}n{\cdot}x$
⟨*proof*⟩

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

lemma $head{\cdot}(reverse{\cdot}x) = last{\cdot}x$
⟨*proof*⟩

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

lemma
assumes $le{\cdot}0{\cdot}n \neq FF$
shows $head{\cdot}(drop{\cdot}n{\cdot}x) = x\ !!\ n$
⟨*proof*⟩

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

lemma $reverse{\cdot}(tail{\cdot}(reverse{\cdot}x)) \sqsubseteq init{\cdot}x$
⟨*proof*⟩

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

lemma
assumes $x \neq []$
shows $take{\cdot}(length{\cdot}x\ -\ 1){\cdot}x \sqsubseteq init{\cdot}x$
⟨*proof*⟩

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

lemma *foldr-append-concat*: $foldr{\cdot}append{\cdot}[] = concat$
⟨*proof*⟩

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

lemma $foldl \cdot append \cdot [] \sqsubseteq concat$
⟨*proof*⟩

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

lemma $span \cdot (neg\ oo\ p) = break \cdot p$
⟨*proof*⟩

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

lemma $break \cdot (neg\ oo\ p) = span \cdot p$
⟨*proof*⟩

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

lemma $the\text{-}or \cdot (map \cdot p \cdot x) = any \cdot p \cdot x$
⟨*proof*⟩

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

lemma $the\text{-}and \cdot (map \cdot p \cdot x) = all \cdot p \cdot x$
⟨*proof*⟩

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

lemma $zipWith \cdot \langle , \rangle = zip$
⟨*proof*⟩

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

lemma $zipWith3 \cdot \langle ,, \rangle = zip3$
⟨*proof*⟩

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

lemma $eq \cdot (length \cdot x) \cdot 0 \sqsubseteq null \cdot x$
⟨*proof*⟩

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

lemma $neq \cdot (length \cdot x) \cdot 0 \sqsubseteq neg \cdot (null \cdot x)$
⟨*proof*⟩

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

lemma $map \cdot (uncurry \cdot f) \cdot (zip \cdot x \cdot y) = zipWith \cdot f \cdot x \cdot y$
⟨*proof*⟩

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

lemma $map \cdot f \cdot (zip \cdot x \cdot y) = zipWith \cdot (curry \cdot f) \cdot x \cdot y$
⟨*proof*⟩

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

lemma $neg \cdot (elem \cdot x \cdot y) = notElem \cdot x \cdot y$
⟨*proof*⟩

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

lemma $foldr \cdot f \cdot z \cdot (map \cdot g \cdot x) = foldr \cdot (f\ oo\ g) \cdot z \cdot x$
$\langle proof \rangle$

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

lemma $null \cdot (filter \cdot f \cdot x) = neg \cdot (any \cdot f \cdot x)$
$\langle proof \rangle$

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

lemma $eq \cdot (filter \cdot f \cdot x) \cdot [] = neg \cdot (any \cdot f \cdot x)$
$\langle proof \rangle$

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

lemma $neq \cdot (filter \cdot f \cdot x) \cdot [] = any \cdot f \cdot x$
$\langle proof \rangle$

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

lemma $any \cdot (\Lambda\ z.\ eq \cdot z \cdot a) = elem \cdot a$
$\langle proof \rangle$

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

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

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

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

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

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

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

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

16.3 Folds

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

lemma $foldr \cdot trand \cdot TT = the\text{-}and$
$\langle proof \rangle$

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

lemma $foldl\text{-}to\text{-}and{:}foldl \cdot trand \cdot TT \sqsubseteq the\text{-}and$
$\langle proof \rangle$

```
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 $foldr \cdot tror \cdot FF = the\text{-}or$
⟨*proof*⟩

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

lemma $foldl\text{-}to\text{-}or$: $foldl \cdot tror \cdot FF \sqsubseteq the\text{-}or$
⟨*proof*⟩

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

lemma $foldr1 \cdot tror \sqsubseteq the\text{-}or$
⟨*proof*⟩

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

lemma $foldl1 \cdot tror \sqsubseteq the\text{-}or$
⟨*proof*⟩

16.4 Function

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

lemma $(\Lambda\ x.\ x) = ID$
⟨*proof*⟩

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

lemma $(\Lambda\ x\ y.\ x) = const$
⟨*proof*⟩

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

lemma $(\Lambda\ \langle x,\ y\rangle.\ x) = fst$
⟨*proof*⟩

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

lemma $(\Lambda\ \langle x,\ y\rangle.\ y) = snd$
⟨*proof*⟩

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

lemma $(\Lambda\ x\ y.\ f \cdot \langle x,\ y\rangle) = curry \cdot f$
⟨*proof*⟩

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

lemma $(\Lambda\ \langle x,\ y\rangle.\ f\cdot x\cdot y) \sqsubseteq \mathit{uncurry}\cdot f$
$\langle \mathit{proof} \rangle$

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

lemma $(\Lambda\ x.\ y) = \mathit{const}\cdot y$
$\langle \mathit{proof} \rangle$

lemma $\mathit{flip}\cdot f\cdot x\cdot y = f\cdot y\cdot x$ $\langle \mathit{proof} \rangle$

16.5 Bool

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

lemma *eq-true*: $\mathit{eq}\cdot x\cdot TT = x$
$\langle \mathit{proof} \rangle$

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

lemma *eq-false*: $\mathit{eq}\cdot x\cdot FF = \mathit{neg}\cdot x$
$\langle \mathit{proof} \rangle$

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

lemma *if-equal*: $(\mathit{If}\ a\ \mathit{then}\ x\ \mathit{else}\ x) \sqsubseteq x$
$\langle \mathit{proof} \rangle$

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

lemma $(\mathit{If}\ a\ \mathit{then}\ TT\ \mathit{else}\ FF) = a$
$\langle \mathit{proof} \rangle$

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

lemma $(\mathit{If}\ a\ \mathit{then}\ FF\ \mathit{else}\ TT) = \mathit{neg}\cdot a$
$\langle \mathit{proof} \rangle$

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

lemma $(\mathit{If}\ a\ \mathit{then}\ t\ \mathit{else}\ (\mathit{If}\ b\ \mathit{then}\ t\ \mathit{else}\ f)) = (\mathit{If}\ a\ \mathit{orelse}\ b\ \mathit{then}\ t\ \mathit{else}\ f)$
$\langle \mathit{proof} \rangle$

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

lemma $(\mathit{If}\ a\ \mathit{then}\ (\mathit{If}\ b\ \mathit{then}\ t\ \mathit{else}\ f)\ \mathit{else}\ f) = (\mathit{If}\ a\ \mathit{andalso}\ b\ \mathit{then}\ t\ \mathit{else}\ f)$
$\langle \mathit{proof} \rangle$

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

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

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

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

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

lemma $(\mathit{If}\ c\ \mathit{then}\ \langle TT, x\rangle\ \mathit{else}\ \langle FF, x\rangle) \sqsubseteq \langle c, x\rangle$
$\langle \mathit{proof} \rangle$

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

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

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

lemma $\mathit{the\text{-}or}\cdot[x, y] = (x\ \mathit{orelse}\ y)$
$\langle \mathit{proof} \rangle$

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

lemma $\mathit{the\text{-}or}\cdot[x, y, z] = (x\ \mathit{orelse}\ y\ \mathit{orelse}\ z)$
$\langle \mathit{proof} \rangle$

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

lemma $\mathit{the\text{-}and}\cdot[x, y] = (x\ \mathit{andalso}\ y)$
$\langle \mathit{proof} \rangle$

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

lemma $\mathit{the\text{-}and}\cdot[x, y, z] = (x\ \mathit{andalso}\ y\ \mathit{andalso}\ z)$
$\langle \mathit{proof} \rangle$

16.6 Arrow

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

lemma $x \sqsubseteq \langle \mathit{fst}\cdot x, \mathit{snd}\cdot x\rangle$
$\langle \mathit{proof} \rangle$

16.7 Seq

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

lemma $\mathit{seq}\cdot x\cdot x = x$ $\langle \mathit{proof} \rangle$

16.8 Evaluate

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

lemma $(TT\ \mathit{andalso}\ x) = x$ $\langle \mathit{proof} \rangle$

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

lemma $(FF\ andalso\ x) = FF$ $\langle proof \rangle$

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

lemma $(TT\ orelse\ x) = TT$ $\langle proof \rangle$

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

lemma $(FF\ orelse\ x) = x$ $\langle proof \rangle$

```
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 $scanr1 \cdot f \cdot [] = []$ ⟨*proof*⟩

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

lemma $scanr1 \cdot f \cdot [x] = [x]$ ⟨*proof*⟩

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

lemma $take \cdot n \cdot [] \sqsubseteq []$ ⟨*proof*⟩

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

lemma $drop \cdot n \cdot [] \sqsubseteq []$
⟨*proof*⟩

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

lemma $takeWhile \cdot p \cdot [] = []$ ⟨*proof*⟩

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

lemma $dropWhile \cdot p \cdot [] = []$ ⟨*proof*⟩

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

lemma $span \cdot p \cdot [] = \langle [], [] \rangle$ ⟨*proof*⟩

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

lemma $concat \cdot [a] = a$ ⟨*proof*⟩

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

lemma $concat \cdot [] = []$ ⟨*proof*⟩

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

lemma $zip \cdot [] \cdot [] = []$ ⟨*proof*⟩

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

lemma $ID \cdot x = x$ ⟨*proof*⟩

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

lemma $const \cdot x \cdot y = x$ ⟨*proof*⟩

16.9 Complex hints

```
take (length t) s == t ==> t `Data.List.isPrefixOf` s
```

lemma
fixes $t :: [{'}a::Eq\text{-}sym]$
shows $eq \cdot (take \cdot (length \cdot t) \cdot s) \cdot t \sqsubseteq isPrefixOf \cdot t \cdot s$
⟨*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 $\neg\ (eq{\cdot}(take{\cdot}i{\cdot}s){\cdot}t \sqsubseteq (le{\cdot}(length{\cdot}t){\cdot}i\ andalso\ isPrefixOf{\cdot}t{\cdot}s))$
  $\langle proof \rangle$

lemma
  assumes $le{\cdot}0{\cdot}i = TT$ **and** $le{\cdot}i{\cdot}0 = FF$
    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 $(\mathit{If}\ c\ \mathit{then}\ f{\cdot}x\ \mathit{else}\ f{\cdot}y) \sqsubseteq f{\cdot}(\mathit{If}\ c\ \mathit{then}\ x\ \mathit{else}\ y)$ $\langle\mathit{proof}\rangle$

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

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

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

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

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

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

```
a /= True ==> not a
```

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

```
a /= False ==> a
```

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

```
True /= a ==> not a
```

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

```
False /= a ==> a
```

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

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

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

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

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

```
x == Nothing ==> isNothing x
```

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

```
Nothing == x ==> isNothing x
```

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

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

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

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

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

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

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

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

lemma $(\mathit{If}\ \mathit{isJust}\cdot x\ \mathit{then}\ \mathit{fromJust}\cdot x\ \mathit{else}\ y) = \mathit{fromMaybe}\cdot y\cdot x$ ⟨*proof*⟩

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

lemma $(\mathit{isJust}\cdot x\ \mathit{andalso}\ (\mathit{eq}\cdot(\mathit{fromJust}\cdot x)\cdot y)) = \mathit{eq}\cdot x\cdot(\mathit{Just}\cdot y)$ ⟨*proof*⟩

```
elem True ==> or
```

lemma $\mathit{elem}\cdot TT = \mathit{the\text{-}or}$
⟨*proof*⟩

```
notElem False ==> and
```

lemma $\mathit{notElem}\cdot FF = \mathit{the\text{-}and}$
⟨*proof*⟩

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

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

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

lemma $\mathit{maybe}\cdot x\cdot ID = \mathit{fromMaybe}\cdot x$
⟨*proof*⟩

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

lemma $\mathit{maybe}\cdot FF\cdot(\mathit{const}\cdot TT) = \mathit{isJust}$
⟨*proof*⟩

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

lemma $\mathit{maybe}\cdot TT\cdot(\mathit{const}\cdot FF) = \mathit{isNothing}$
⟨*proof*⟩

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

lemma $\mathit{maybe}\cdot[]\cdot(\Lambda\ z.\ z : []) = \mathit{maybeToList}$
⟨*proof*⟩

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

lemma $\mathit{catMaybes}\cdot(\mathit{map}\cdot f\cdot x) = \mathit{mapMaybe}\cdot f\cdot x$ ⟨*proof*⟩

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

lemma $(\mathit{If}\ \mathit{isNothing}\cdot x\ \mathit{then}\ y\ \mathit{else}\ f\cdot(\mathit{fromJust}\cdot x)) = \mathit{maybe}\cdot y\cdot f\cdot x$ ⟨*proof*⟩

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

lemma $(\mathit{If}\ \mathit{isJust}\cdot x\ \mathit{then}\ f\cdot(\mathit{fromJust}\cdot x)\ \mathit{else}\ y) = \mathit{maybe}\cdot y\cdot f\cdot x$ ⟨*proof*⟩

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

lemma $(\mathit{map}\cdot\mathit{fromJust}\ \mathit{oo}\ \mathit{filter}\cdot\mathit{isJust}) = \mathit{catMaybes}$
⟨*proof*⟩

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

lemma *concatMap*$\cdot$(*maybeToList oo f*) = *mapMaybe*$\cdot$*f*
⟨*proof*⟩

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

lemma *concatMap*$\cdot$*maybeToList* = *catMaybes* ⟨*proof*⟩

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

lemma *mapMaybe*$\cdot$*f*$\cdot$(*map*$\cdot$*g*$\cdot$*x*) = *mapMaybe*$\cdot$(*f oo g*)$\cdot$*x* ⟨*proof*⟩

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

lemma (*dollar oo f*) = *f* ⟨*proof*⟩

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

lemma (Λ *z*. *dollar*$\cdot$*f*$\cdot$*z*) = *f* ⟨*proof*⟩

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

lemma (Λ *a b*. *g*$\cdot$(*f*$\cdot$*a*)$\cdot$(*f*$\cdot$*b*)) = *on*$\cdot$*g*$\cdot$*f* ⟨*proof*⟩

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

lemma *dollarBang*$\cdot$*ID*$\cdot$*x* = *x* ⟨*proof*⟩

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

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

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

lemma *isPrefixOf*$\cdot$(*reverse*$\cdot$*x*)$\cdot$(*reverse*$\cdot$*y*) = *isSuffixOf*$\cdot$*x*$\cdot$*y* ⟨*proof*⟩

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

lemma *concat*$\cdot$(*intersperse*$\cdot$*x*$\cdot$*y*) = *intercalate*$\cdot$*x*$\cdot$*y* ⟨*proof*⟩

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

lemma
  assumes $x \neq \bot$ **shows** *seq*$\cdot$*x*$\cdot$*y* = *y*
  ⟨*proof*⟩

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

lemma assumes $x \neq \bot$ **shows** *dollarBang*$\cdot$*f*$\cdot$*x* = *f*$\cdot$*x*
  ⟨*proof*⟩

```
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*⟩

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.