

# Isabelle/HOLCF-Prelude

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

May 26, 2024

## 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

<b>1</b>	<b>Initial Setup for HOLCF-Prelude</b>	<b>2</b>
<b>2</b>	<b>Type Classes</b>	<b>4</b>
2.1	Eq class . . . . .	4
2.1.1	Class instances . . . . .	5
2.2	Ord class . . . . .	5
<b>3</b>	<b>Cpo for Numerals</b>	<b>8</b>
<b>4</b>	<b>Data: Functions</b>	<b>11</b>
<b>5</b>	<b>Data: Bool</b>	<b>12</b>
5.1	Class instances . . . . .	12
5.2	Lemmas . . . . .	12
<b>6</b>	<b>Data: Tuple</b>	<b>14</b>
6.1	Datatype definitions . . . . .	14
6.2	Type class instances . . . . .	14

---

\*Supported by the Deutsche Telekom Stiftung.

†Supported by the Austrian Science Fund (FWF): J3202.

<b>7</b>	<b>Data: Integers</b>	<b>16</b>
7.1	Induction rules that do not break the abstraction . . . . .	21
<b>8</b>	<b>Data: List</b>	<b>22</b>
8.1	Datatype definition . . . . .	22
8.1.1	Section syntax for <i>Cons</i> . . . . .	22
8.2	Haskell function definitions . . . . .	22
8.2.1	Arithmetic Sequences . . . . .	26
8.3	Logical predicates on lists . . . . .	28
8.4	Properties . . . . .	29
8.5	<i>reverse</i> and <i>reverse</i> induction . . . . .	42
<b>9</b>	<b>Data: Maybe</b>	<b>43</b>
<b>10</b>	<b>Definedness</b>	<b>45</b>
<b>11</b>	<b>List Comprehension</b>	<b>48</b>
<b>12</b>	<b>The Num Class</b>	<b>49</b>
12.1	Num class . . . . .	49
12.2	Instances for Integer . . . . .	50
<b>13</b>	<b>Fibonacci sequence</b>	<b>52</b>
<b>14</b>	<b>The Sieve of Eratosthenes</b>	<b>52</b>
<b>15</b>	<b>GHC's "fold/build" Rule</b>	<b>54</b>
15.1	Approximating the Rewrite Rule . . . . .	54
15.2	Lemmas . . . . .	55
15.3	Examples . . . . .	57
<b>16</b>	<b>HLint</b>	<b>58</b>
16.1	Ord . . . . .	58
16.2	List . . . . .	58
16.3	Folds . . . . .	61
16.4	Function . . . . .	62
16.5	Bool . . . . .	63
16.6	Arrow . . . . .	64
16.7	Seq . . . . .	64
16.8	Evaluate . . . . .	64
16.9	Complex hints . . . . .	66

# 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 :: pttm ⇒ 'a List.list ⇒ bool ⇒ 'a List.list ((1[-<--./ -]))

```

```

no-syntax

```

```
-filter :: pttm => 'a List.list => bool => 'a List.list ((1[←←- ./ -]))
```

Declarations that belong in HOLCF/Tr.thy:

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

end
```

## 2 Type Classes

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

### 2.1 Eq class

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

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

```
fixrec neq :: 'a::Eq → 'a → tr where
  neq.x.y = neq.(eq.x.y)
```

```
class Eq-strict = Eq +
  assumes eq-strict [simp]:
    eq.x.⊥ = ⊥
    eq.⊥.y = ⊥
```

```
class Eq-sym = Eq-strict +
  assumes eq-sym: eq.x.y = eq.y.x
```

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

```
lemma eq-refl: eq.x.x ≠ ⊥ ⇒ eq.x.x = TT
  ⟨proof⟩
```

```
end
```

```
class Eq-eq = Eq-sym +
  assumes eq-self-neq-FF': eq.x.x ≠ FF
  and eq-TT-dest: eq.x.y = TT ⇒ x = y
begin
```

**subclass** *Eq-equiv*  
  ⟨*proof*⟩

**lemma** *eqD* [*dest*]:  
   $eq \cdot x \cdot y = TT \implies x = y$   
   $eq \cdot x \cdot y = FF \implies x \neq y$   
  ⟨*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* | *EQ* | *GT*

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

**lemma** *oppOrdering-simps* [*simp*]:  
   $oppOrdering \cdot LT = GT$   
   $oppOrdering \cdot EQ = EQ$   
   $oppOrdering \cdot GT = LT$   
   $oppOrdering \cdot \perp = \perp$   
  ⟨*proof*⟩

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

**definition** *lt* :: 'a  $\rightarrow$  'a  $\rightarrow$  *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  $\rightarrow$  'a  $\rightarrow$  *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*:  $lt \cdot x \cdot y = TT \iff compare \cdot x \cdot y = LT$

```

    <proof>

end

class Ord-strict = Ord +
  assumes compare-strict [simp]:
    compare. $\perp$ .y =  $\perp$ 
    compare.x. $\perp$  =  $\perp$ 
begin

lemma lt-strict [simp]:
  shows lt. $\perp$ .x =  $\perp$ 
    and lt.x. $\perp$  =  $\perp$ 
  <proof>

lemma le-strict [simp]:
  shows le. $\perp$ .x =  $\perp$ 
    and le.x. $\perp$  =  $\perp$ 
  <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.x.y = is-EQ.(compare.x.y)
    and oppOrdering-compare [simp]:
      oppOrdering.(compare.x.y) = compare.y.x
    and compare-EQ-dest: compare.x.y = EQ  $\implies$  x = y
    and compare-self-below-EQ: compare.x.x  $\sqsubseteq$  EQ
    and compare-LT-trans:
      compare.x.y = LT  $\implies$  compare.y.z = LT  $\implies$  compare.x.z = LT

begin

lemma eq-TT-dest: eq.x.y = TT  $\implies$  x = y
  <proof>

lemma le-iff-lt-or-eq:
  le.x.y = TT  $\iff$  lt.x.y = TT  $\vee$  eq.x.y = TT
  <proof>

lemma compare-sym:
  compare.x.y = (case compare.y.x of LT  $\Rightarrow$  GT | EQ  $\Rightarrow$  EQ | GT  $\Rightarrow$  LT)
  <proof>

lemma compare-self-neq-LT [simp]: compare.x.x  $\neq$  LT
  <proof>

```

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

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

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

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

**lemma** *compare-GT-trans*:  
 $\text{compare}\cdot x\cdot y = GT \implies \text{compare}\cdot y\cdot z = GT \implies \text{compare}\cdot x\cdot z = GT$   
{*proof*}

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

**lemma** *compare-EQ-trans*:  
 $\text{compare}\cdot x\cdot y = EQ \implies \text{compare}\cdot y\cdot z = EQ \implies \text{compare}\cdot x\cdot z = EQ$   
{*proof*}

**lemma** *le-trans*:  
 $le\cdot x\cdot y = TT \implies le\cdot y\cdot z = TT \implies le\cdot x\cdot z = TT$   
{*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* ::  $Ordering \rightarrow Ordering \rightarrow Ordering$  **where**  
 $\text{thenOrdering} = (\lambda x\ y. \text{case } x \text{ of } LT \Rightarrow LT \mid EQ \Rightarrow y \mid GT \Rightarrow GT)$

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

**lemma** *thenOrdering-LT-iff* [simp]:  
 $thenOrdering.x.y = LT \longleftrightarrow x = LT \vee x = EQ \wedge y = LT$   
 ⟨proof⟩

**lemma** *thenOrdering-EQ-iff* [simp]:  
 $thenOrdering.x.y = EQ \longleftrightarrow x = EQ \wedge y = EQ$   
 ⟨proof⟩

**lemma** *thenOrdering-GT-iff* [simp]:  
 $thenOrdering.x.y = GT \longleftrightarrow x = GT \vee x = EQ \wedge y = GT$   
 ⟨proof⟩

**lemma** *thenOrdering-below-EQ-iff* [simp]:  
 $thenOrdering.x.y \sqsubseteq EQ \longleftrightarrow x \sqsubseteq EQ \wedge (x = \perp \vee y \sqsubseteq EQ)$   
 ⟨proof⟩

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

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

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

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

**instance** ⟨proof⟩

**end**

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

**end**

### 3 Cpo for Numerals

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



```

class plus-cpo = plus + cpo +
  assumes cont-plus1: cont ( $\lambda x::'a::\{plus,cpo\}. x + y$ )
  assumes cont-plus2: cont ( $\lambda y::'a::\{plus,cpo\}. x + y$ )
begin

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

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

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

end

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

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

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

abbreviation minus-section-right :: 'a  $\Rightarrow$  'a  $\rightarrow$  'a ('(--')) where
  (-y)  $\equiv$   $\Lambda x. x - y$ 

end

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

end

lemma cont2cont-plus [simp, cont2cont]:
  assumes cont ( $\lambda x. f x$ ) and cont ( $\lambda x. g x$ )
  shows cont ( $\lambda x. f x + g x :: 'a::plus-cpo$ )
  <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.(0::'a)
  instance <proof>
end

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

instantiation u :: (plus-cpo) plus
begin
  definition plus-u x y = ( $\Lambda$ (up.a) (up.b). up.(a + b)).x.y for x y :: 'a⊥
  instance <proof>
end

instantiation u :: (minus-cpo) minus
begin
  definition minus-u x y = ( $\Lambda$ (up.a) (up.b). up.(a - b)).x.y for x y :: 'a⊥
  instance <proof>
end

instantiation u :: (times-cpo) times
begin
  definition times-u x y = ( $\Lambda$ (up.a) (up.b). up.(a * b)).x.y for x y :: 'a⊥
  instance <proof>
end

lemma plus-u-strict [simp]:
  fixes x :: - u shows x + ⊥ = ⊥ and ⊥ + x = ⊥
  <proof>

lemma minus-u-strict [simp]:
  fixes x :: - u shows x - ⊥ = ⊥ and ⊥ - x = ⊥
  <proof>

lemma times-u-strict [simp]:
  fixes x :: - u shows x * ⊥ = ⊥ and ⊥ * x = ⊥
  <proof>

```

```

lemma plus-up-up [simp]:  $up \cdot x + up \cdot y = up \cdot (x + y)$ 
  <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 -> 'b -> 'c) -> 'b -> 'a -> 'c where
  flip . f . x . y = f . y . x

```

```

fixrec const :: 'a → 'b → 'a where
  const·x·- = x

fixrec dollar :: ('a -> 'b) -> 'a -> 'b where
  dollar·f·x = f·x

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

fixrec on :: ('b -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'a -> 'c where
  on·g·f·x·y = g·(f·x)·(f·y)

end

```

## 5 Data: Bool

```

theory Data-Bool
  imports Type-Classes
begin

```

### 5.1 Class instances

Eq

```

lemma eq-eqI [case-names bottomLTR bottomRTL LTR RTL]:
  (x = ⊥ ⇒ y = ⊥) ⇒ (y = ⊥ ⇒ x = ⊥) ⇒ (x = TT ⇒ y = TT) ⇒ (y
= TT ⇒ x = TT) ⇒ x = y
⟨proof⟩

```

```

lemma eq-tr-simps [simp]:
  shows eq·TT·TT = TT and eq·TT·FF = FF
  and eq·FF·TT = FF and eq·FF·FF = TT
⟨proof⟩

```

Ord

```

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

```

### 5.2 Lemmas

```

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

```

```

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

```

$$(x \text{ andalso } y) = FF \longleftrightarrow x = FF \vee (x = TT \wedge y = FF)$$

*<proof>*

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

$$(x \text{ andalso } y) = \perp \longleftrightarrow x = \perp \vee (x = TT \wedge y = \perp)$$

*<proof>*

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

$$(x \text{ orelse } y) = FF \longleftrightarrow x = FF \wedge y = FF$$

*<proof>*

**lemma** *orelse-assoc* [simp]:

$$((x \text{ orelse } y) \text{ orelse } z) = (x \text{ orelse } y \text{ orelse } z)$$

*<proof>*

**lemma** *andalso-assoc* [simp]:

$$((x \text{ andalso } y) \text{ andalso } z) = (x \text{ andalso } y \text{ andalso } z)$$

*<proof>*

**lemma** *neg-orelse* [simp]:

$$\text{neg} \cdot (x \text{ orelse } y) = (\text{neg} \cdot x \text{ andalso } \text{neg} \cdot y)$$

*<proof>*

**lemma** *neg-andalso* [simp]:

$$\text{neg} \cdot (x \text{ andalso } y) = (\text{neg} \cdot x \text{ orelse } \text{neg} \cdot y)$$

*<proof>*

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

**lemma** *neg-eq-simps*:

$$\text{neg} \cdot x = TT \implies x = FF$$

$$\text{neg} \cdot x = FF \implies x = TT$$

$$\text{neg} \cdot x = \perp \implies x = \perp$$

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *neg-eq-bottom-iff* [simp]:  $\text{neg} \cdot x = \perp \longleftrightarrow x = \perp$

*<proof>*

**lemma** *neg-eq* [simp]:

$$\text{neg} \cdot x = \text{neg} \cdot y \longleftrightarrow x = y$$

*<proof>*

**lemma** *neg-neg* [*simp*]:  
 $neg \cdot (neg \cdot x) = x$   
 ⟨*proof*⟩

**lemma** *neg-comp-neg* [*simp*]:  
 $neg \circ neg = ID$   
 ⟨*proof*⟩

**end**

## 6 Data: Tuple

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

### 6.1 Datatype definitions

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

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

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

**fixrec** *uncurry* :: ( $'a \rightarrow 'b \rightarrow 'c$ )  $\rightarrow$  ( $'a$ ,  $'b$ )  $\rightarrow 'c$   
**where**  $uncurry \cdot f \cdot p = f \cdot (fst \cdot p) \cdot (snd \cdot p)$

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

**domain** ( $'a$ ,  $'b$ ,  $'c$ ) *Tuple3* ( $\langle -, -, - \rangle$ ) =  
*Tuple3* (**lazy**  $'a$ ) (**lazy**  $'b$ ) (**lazy**  $'c$ ) ( $\langle -, -, - \rangle$ )

**notation** *Tuple3* ( $\langle \rangle$ )

### 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**  
   $\langle proof \rangle$

**end**

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

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

**instance**  $\langle proof \rangle$

**end**

**lemma** *eq-Tuple2-simps* [simp]:  
   $eq \cdot \langle x1, y1 \rangle \cdot \langle x2, y2 \rangle = (eq \cdot x1 \cdot x2 \text{ andalso } eq \cdot y1 \cdot y2)$   
   $\langle proof \rangle$

**instance** *Tuple2* :: (Eq-sym, Eq-sym) Eq-sym  
 $\langle proof \rangle$

**instance** *Tuple2* :: (Eq-equiv, Eq-equiv) Eq-equiv  
 $\langle proof \rangle$

**instance** *Tuple2* :: (Eq-eq, Eq-eq) Eq-eq  
 $\langle proof \rangle$

**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**  
   $\langle proof \rangle$

**end**

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

**instance** *Tuple2* :: (Ord-linear, Ord-linear) Ord-linear  
 $\langle proof \rangle$

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

**definition**

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

**instance**  $\langle proof \rangle$

**end**

**lemma**  $eq\text{-}Tuple3\text{-simps}$  [simp]:

$eq \cdot \langle x1, y1, z1 \rangle \cdot \langle x2, y2, z2 \rangle = (eq \cdot x1 \cdot x2 \text{ andalso } eq \cdot y1 \cdot y2 \text{ andalso } eq \cdot z1 \cdot z2)$   
 $\langle proof \rangle$

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

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

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

**instantiation**  $Tuple3 :: (Ord, Ord, Ord) Ord\text{-}strict$   
**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**  $compare\text{-}Tuple3\text{-simps}$  [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**  $Tuple3 :: (Ord\text{-}linear, Ord\text{-}linear, Ord\text{-}linear) Ord\text{-}linear$   
 $\langle proof \rangle$

**end**

## 7 Data: Integers

**theory**  $Data\text{-}Integer$

**imports**



```

    Numeral-Cpo
    Data-Bool
begin

domain Integer = MkI (lazy int)

instance Integer :: flat
⟨proof⟩

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

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

instance ⟨proof⟩

end

lemma Integer-arith-strict [simp]:
  fixes x :: Integer
  shows ⊥ + x = ⊥ and x + ⊥ = ⊥
    and ⊥ * x = ⊥ and x * ⊥ = ⊥
    and ⊥ - x = ⊥ and x - ⊥ = ⊥
    and - ⊥ = (⊥::Integer)
  ⟨proof⟩

lemma Integer-arith-simps [simp]:
  MkI·a + MkI·b = MkI·(a + b)
  MkI·a * MkI·b = MkI·(a * b)
  MkI·a - MkI·b = MkI·(a - b)
  - MkI·a = MkI·(uminus a)
  ⟨proof⟩

lemma plus-MkI-MkI:
  MkI·x + MkI·y = MkI·(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*]:

$b \neq \perp \implies (a :: \text{Integer}) + b - b = a$

*<proof>*

**lemma** *zero-Integer-neq-bottom* [*simp*]:  $(0 :: \text{Integer}) \neq \perp$

*<proof>*

**lemma** *one-Integer-neq-bottom* [*simp*]:  $(1 :: \text{Integer}) \neq \perp$

*<proof>*

**lemma** *plus-Integer-eq-bottom-iff* [*simp*]:

**fixes**  $x\ y :: \text{Integer}$  **shows**  $x + y = \perp \iff x = \perp \vee y = \perp$

*<proof>*

**lemma** *diff-Integer-eq-bottom-iff* [*simp*]:

**fixes**  $x\ y :: \text{Integer}$  **shows**  $x - y = \perp \iff x = \perp \vee y = \perp$

*<proof>*

**lemma** *mult-Integer-eq-bottom-iff* [*simp*]:

**fixes**  $x\ y :: \text{Integer}$  **shows**  $x * y = \perp \iff x = \perp \vee y = \perp$

*<proof>*

**lemma** *minus-Integer-eq-bottom-iff* [*simp*]:

**fixes**  $x :: \text{Integer}$  **shows**  $-x = \perp \iff x = \perp$

*<proof>*

**lemma** *numeral-Integer-eq*:  $\text{numeral } k = \text{MkI} \cdot (\text{numeral } k)$

*<proof>*

**lemma** *numeral-Integer-neq-bottom* [*simp*]:  $(\text{numeral } k :: \text{Integer}) \neq \perp$

*<proof>*

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

**lemma** *bottom-neq-zero-Integer* [*simp*]:  $(\perp :: \text{Integer}) \neq 0$

*<proof>*

**lemma** *bottom-neq-one-Integer* [*simp*]:  $(\perp :: \text{Integer}) \neq 1$

*<proof>*

**lemma** *bottom-neq-numeral-Integer* [*simp*]:  $(\perp :: \text{Integer}) \neq \text{numeral } k$

*<proof>*

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

**begin**

**definition**

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

**definition**

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

**instance**  $\langle proof \rangle$

**end**

**lemma** *eq-MkI-MkI* [*simp*]:

$eq \cdot (MkI \cdot m) \cdot (MkI \cdot n) = (\text{if } m = n \text{ then } TT \text{ else } FF)$   
 $\langle proof \rangle$

**lemma** *compare-MkI-MkI* [*simp*]:

$compare \cdot (MkI \cdot x) \cdot (MkI \cdot y) = (\text{if } x < y \text{ then } LT \text{ else if } x > y \text{ then } GT \text{ else } EQ)$   
 $\langle proof \rangle$

**lemma** *lt-MkI-MkI* [*simp*]:

$lt \cdot (MkI \cdot x) \cdot (MkI \cdot y) = (\text{if } x < y \text{ then } TT \text{ else } FF)$   
 $\langle proof \rangle$

**lemma** *le-MkI-MkI* [*simp*]:

$le \cdot (MkI \cdot x) \cdot (MkI \cdot y) = (\text{if } x \leq y \text{ then } TT \text{ else } FF)$   
 $\langle proof \rangle$

**lemma** *eq-Integer-bottom-iff* [*simp*]:

**fixes**  $x \ y :: \text{Integer}$  **shows**  $eq \cdot x \cdot y = \perp \iff x = \perp \vee y = \perp$   
 $\langle proof \rangle$

**lemma** *compare-Integer-bottom-iff* [*simp*]:

**fixes**  $x \ y :: \text{Integer}$  **shows**  $compare \cdot x \cdot y = \perp \iff x = \perp \vee y = \perp$   
 $\langle proof \rangle$

**lemma** *lt-Integer-bottom-iff* [*simp*]:

**fixes**  $x \ y :: \text{Integer}$  **shows**  $lt \cdot x \cdot y = \perp \iff x = \perp \vee y = \perp$   
 $\langle proof \rangle$

**lemma** *le-Integer-bottom-iff* [*simp*]:

**fixes**  $x \ y :: \text{Integer}$  **shows**  $le \cdot x \cdot y = \perp \iff x = \perp \vee y = \perp$   
 $\langle proof \rangle$

**lemma** *compare-refl-Integer* [*simp*]:

$(x :: \text{Integer}) \neq \perp \implies compare \cdot x \cdot x = EQ$

$\langle \text{proof} \rangle$

**lemma** *eq-refl-Integer* [simp]:  
 $(x::\text{Integer}) \neq \perp \implies \text{eq}.x.x = \text{TT}$   
 $\langle \text{proof} \rangle$

**lemma** *lt-refl-Integer* [simp]:  
 $(x::\text{Integer}) \neq \perp \implies \text{lt}.x.x = \text{FF}$   
 $\langle \text{proof} \rangle$

**lemma** *le-refl-Integer* [simp]:  
 $(x::\text{Integer}) \neq \perp \implies \text{le}.x.x = \text{TT}$   
 $\langle \text{proof} \rangle$

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

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

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

*<proof>*

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

*le*·(*0::Integer*)·*0* = *TT*

*le*·(*0::Integer*)·*1* = *TT*

*le*·(*1::Integer*)·*0* = *FF*

*le*·(*1::Integer*)·*1* = *TT*

*le*·(*0::Integer*)·(*numeral k*) = *TT*

*le*·(*numeral k*)·(*0::Integer*) = *FF*

*le*·(*1::Integer*)·(*numeral k*) = *TT*

*Num.One* < *k*  $\implies$  *le*·(*numeral k*)·(*1::Integer*) = *FF*

*le*·(*numeral k::Integer*)·(*numeral l*) = (*if k*  $\leq$  *l* then *TT* else *FF*)

*<proof>*

**lemma** *MkI-eq-0-iff* [*simp*]: *MkI*·*n* = *0*  $\longleftrightarrow$  *n* = *0*

*<proof>*

**lemma** *MkI-eq-1-iff* [*simp*]: *MkI*·*n* = *1*  $\longleftrightarrow$  *n* = *1*

*<proof>*

**lemma** *MkI-eq-numeral-iff* [*simp*]: *MkI*·*n* = *numeral k*  $\longleftrightarrow$  *n* = *numeral k*

*<proof>*

**lemma** *MkI-0*: *MkI*·*0* = *0*

*<proof>*

**lemma** *MkI-1*: *MkI*·*1* = *1*

*<proof>*

**lemma** *le-plus-1*:

**fixes** *m* :: *Integer*

**assumes** *le*·*m*·*n* = *TT*

**shows** *le*·*m*·(*n* + *1*) = *TT*

*<proof>*

## 7.1 Induction rules that do not break the abstraction

**lemma** *nonneg-Integer-induct* [*consumes 1, case-names 0 step*]:

**fixes** *i* :: *Integer*

**assumes** *i*-*nonneg*: *le*·*0*·*i* = *TT*

**and** *zero*: *P 0*

**and** *step*:  $\bigwedge i. le \cdot 1 \cdot i = TT \implies P (i - 1) \implies P i$

**shows** *P i*

*<proof>*

**end**

## 8 Data: List

**theory** *Data-List*

**imports**

*Type-Classes*

*Data-Function*

*Data-Bool*

*Data-Tuple*

*Data-Integer*

*Numeral-Cpo*

**begin**

**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*) ≡  $\Lambda$  *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-list.(x : xs).(y : ys) = (eq.x.y andalso eq-list.xs.ys)

instance ⟨proof⟩

end

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

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

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

instantiation list :: (Ord) Ord-strict
begin

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

instance
  ⟨proof⟩

end

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

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

definition zip :: ['a] → ['b] → [( 'a, 'b)] where
  zip = zipWith.<,>

fixrec zipWith3 :: ('a → 'b → 'c → 'd) → ['a] → ['b] → ['c] → ['d] where
  zipWith3.f.(x : xs).(y : ys).(z : zs) = f.x.y.z : zipWith3.f.xs.ys.zs |
  (unchecked) zipWith3.f.xs.ys.zs = []

definition zip3 :: ['a] → ['b] → ['c] → [( 'a, 'b, 'c)] where
  zip3 = zipWith3.<,>

fixrec map :: ('a → 'b) → ['a] → ['b] where

```

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

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

**fixrec**  $repeat :: 'a \rightarrow [a]$  **where**  
 $[simp \ del]: repeat \cdot x = x : repeat \cdot x$

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

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

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

**fixrec**  $break :: ('a \rightarrow tr) \rightarrow [a] \rightarrow \langle [a], [a] \rangle$  **where**  
 $break \cdot p = span \cdot (neg \ o o \ p)$

**fixrec**  $nth :: [a] \rightarrow Integer \rightarrow 'a$  **where**  
 $nth \cdot [] \cdot n = \perp \mid$   
 $nth \cdot Cons \ [simp \ del]:$   
 $nth \cdot (x : xs) \cdot n =$  *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**  $partition :: ('a \rightarrow tr) \rightarrow [a] \rightarrow \langle [a], [a] \rangle$  **where**  
 $partition = (\Lambda \ P \ xs. \langle filter \cdot P \cdot xs, filter \cdot (neg \ o o \ P) \cdot xs \rangle)$

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

**fixrec**  $foldl :: ('a \rightarrow 'b \rightarrow 'a) \rightarrow 'a \rightarrow [b] \rightarrow 'a$  **where**  
 $foldl \cdot f \cdot z \cdot [] = z \mid$   
 $foldl \cdot f \cdot z \cdot (x : xs) = foldl \cdot f \cdot (f \cdot z \cdot x) \cdot xs$

**fixrec**  $foldl1 :: ('a \rightarrow 'a \rightarrow 'a) \rightarrow [a] \rightarrow 'a$  **where**  
 $foldl1 \cdot f \cdot [] = \perp \mid$   
 $foldl1 \cdot f \cdot (x : xs) = foldl \cdot f \cdot x \cdot xs$



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

**fixrec** *foldr1* :: ('a → 'a → 'a) → ['a] → 'a **where**  
*foldr1*·f·[] = ⊥ |  
*foldr1*·f·[x] = x |  
*foldr1*·f·(x : (x':xs)) = f·x·(*foldr1*·f·(x':xs))

**fixrec** *elem* :: 'a::Eq → ['a] → tr **where**  
*elem*·x·[] = FF |  
*elem*·x·(y : ys) = (eq·y·x orelse *elem*·x·ys)

**fixrec** *notElem* :: 'a::Eq → ['a] → tr **where**  
*notElem*·x·[] = TT |  
*notElem*·x·(y : ys) = (neq·y·x andalso *notElem*·x·ys)

**fixrec** *append* :: ['a] → ['a] → ['a] **where**  
*append*·[]·ys = ys |  
*append*·(x : xs)·ys = x : *append*·xs·ys

**abbreviation** *append-syn* :: ['a] ⇒ ['a] ⇒ ['a] (**infixr** ++ 65) **where**  
xs ++ ys ≡ *append*·xs·ys

**definition** *concat* :: [['a]] → ['a] **where**  
*concat* = *foldr*·*append*·[]

**definition** *concatMap* :: ('a → ['b]) → ['a] → ['b] **where**  
*concatMap* = (Λ f. *concat* oo map·f)

**fixrec** *last* :: ['a] -> 'a **where**  
*last*·[x] = x |  
*last*·(-:(x:xs)) = *last*·(x:xs)

**fixrec** *init* :: ['a] -> ['a] **where**  
*init*·[x] = [] |  
*init*·(x:(y:xs)) = x:(*init*·(y:xs))

**fixrec** *reverse* :: ['a] -> ['a] **where**  
[simp del]:*reverse* = *foldl*·(flip·(·))·[]

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

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

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

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

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

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

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

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

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

**definition** *scanl1* :: ('a → 'a → 'a) → ['a] → ['a] **where**  
*scanl1* = (λ f ls. (case ls of  
 [] ⇒ []  
 | x : xs ⇒ *scanl*·f·x·xs))

### 8.2.1 Arithmetic Sequences

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

**fixrec** *intsFrom* :: Integer → [Integer] **where**

```

[simp del]: intsFrom·x = seq·x·(x : intsFrom·(x+1))

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

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

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

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

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

end

abbreviation enumFromTo-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 → ['a] → ['a] **where**  
*intersperse.sep*·[] = [] |  
*intersperse.sep*·[x] = [x] |  
*intersperse.sep*·(x:y:xs) = x:sep:intersperse.sep·(y:xs)

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

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

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

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

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

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

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

### 8.3 Logical predicates on lists

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

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

**lemma** *finite-list-upwards*:  
**assumes** *finite-list* xs **and** xs ⊆ ys  
**shows** *finite-list* ys  
 ⟨*proof*⟩

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

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

*<proof>*

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

**lemma** *listmem-simps* [*simp*]:  
  **shows**  $\neg \text{listmem } x \perp$  **and**  $\neg \text{listmem } x []$   
  **and**  $\text{listmem } x (y : ys) \longleftrightarrow x = y \vee \text{listmem } x ys$   
  *<proof>*

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

**lemma** *set-simps* [*simp*]:  
  **shows**  $\text{set } \perp = \{\}$  **and**  $\text{set } [] = \{\}$   
  **and**  $\text{set } (x : xs) = \text{insert } x (\text{set } xs)$   
  *<proof>*

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

## 8.4 Properties

**lemma** *map-strict* [*simp*]:  
   $\text{map} \cdot P \cdot \perp = \perp$   
  *<proof>*

**lemma** *map-ID* [*simp*]:  
   $\text{map} \cdot \text{ID} \cdot xs = xs$   
  *<proof>*

**lemma** *enumFrom-intsFrom-conv* [*simp*]:  
   $\text{enumFrom} = \text{intsFrom}$   
  *<proof>*

**lemma** *enumFromTo-upto-conv* [*simp*]:  
   $\text{enumFromTo} = \text{upto}$   
  *<proof>*

**lemma** *zipWith-strict* [*simp*]:  
   $\text{zipWith} \cdot f \cdot \perp \cdot ys = \perp$   
   $\text{zipWith} \cdot f \cdot (x : xs) \cdot \perp = \perp$   
  *<proof>*

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

$zip \cdot (x : xs) \cdot \perp = \perp$   
 $zip \cdot [] \cdot ys = []$   
 $zip \cdot \perp \cdot ys = \perp$   
 ⟨proof⟩

**lemma** *zip-Nil2* [simp]:  
 $xs \neq \perp \implies zip \cdot xs \cdot [] = []$   
 ⟨proof⟩

**lemma** *nth-strict* [simp]:  
 $nth \cdot \perp \cdot n = \perp$   
 $nth \cdot xs \cdot \perp = \perp$   
 ⟨proof⟩

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

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

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

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

**lemma** *nth-Cons-split*:  

$$P (nth \cdot (x : xs) \cdot n) = ((eq \cdot n \cdot 0 = FF \implies P (nth \cdot (x : xs) \cdot n)) \wedge$$

$$(eq \cdot n \cdot 0 = TT \implies P (nth \cdot (x : xs) \cdot n)) \wedge$$

$$(n = \perp \implies P (nth \cdot (x : xs) \cdot n)))$$

⟨proof⟩

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

**lemma** *take-strict* [*simp*]:

$$\text{take} \cdot \perp \cdot xs = \perp$$

*<proof>*

**lemma** *take-strict-2* [*simp*]:

$$\text{le} \cdot 1 \cdot i = TT \implies \text{take} \cdot i \cdot \perp = \perp$$

*<proof>*

**lemma** *drop-strict* [*simp*]:

$$\text{drop} \cdot \perp \cdot xs = \perp$$

*<proof>*

**lemma** *isPrefixOf-strict* [*simp*]:

$$\text{isPrefixOf} \cdot \perp \cdot xs = \perp$$

$$\text{isPrefixOf} \cdot (x:xs) \cdot \perp = \perp$$

*<proof>*

**lemma** *last-strict* [*simp*]:

$$\text{last} \cdot \perp = \perp$$

$$\text{last} \cdot (x:\perp) = \perp$$

*<proof>*

**lemma** *last-nil* [*simp*]:

$$\text{last} \cdot [] = \perp$$

*<proof>*

**lemma** *last-spine-strict*:  $\neg \text{finite-list } xs \implies \text{last} \cdot xs = \perp$

*<proof>*

**lemma** *init-strict* [*simp*]:

$$\text{init} \cdot \perp = \perp$$

$$\text{init} \cdot (x:\perp) = \perp$$

*<proof>*

**lemma** *init-nil* [*simp*]:

$$\text{init} \cdot [] = \perp$$

*<proof>*

**lemma** *strict-foldr-strict2* [*simp*]:

$$(\bigwedge x. f \cdot x \cdot \perp = \perp) \implies \text{foldr} \cdot f \cdot \perp \cdot xs = \perp$$

*<proof>*

**lemma** *foldr-strict* [*simp*]:

$$\text{foldr} \cdot f \cdot d \cdot \perp = \perp$$

$$\text{foldr} \cdot f \cdot \perp \cdot [] = \perp$$

$$\text{foldr} \cdot \perp \cdot d \cdot (x : xs) = \perp$$

*<proof>*

**lemma** *foldr-Cons-Nil* [*simp*]:

$foldr \cdot (\cdot) \cdot [] \cdot xs = xs$   
 $\langle proof \rangle$

**lemma** *append-strict1* [*simp*]:  
 $\perp ++ ys = \perp$   
 $\langle proof \rangle$

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

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

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

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

**lemma** *foldl-spine-strict*:  
 $\neg \text{finite-list } xs \implies foldl \cdot f \cdot x \cdot xs = \perp$   
 $\langle proof \rangle$

**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**  $foldl \cdot f \cdot z \cdot xs = foldr \cdot f \cdot z \cdot xs$   
 $\langle proof \rangle$

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

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

**lemma** *list-eq-nil* [*simp*]:  
 $eq \cdot l \cdot [] = TT \iff l = []$   
 $eq \cdot [] \cdot l = TT \iff l = []$   
 $\langle proof \rangle$



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

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

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

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

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

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

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

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

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

**lemma** *take-intsFrom-enumFrom* [*simp*]:  
 $\text{take} \cdot (\text{MkI} \cdot n) \cdot [\text{MkI} \cdot i ..] = [\text{MkI} \cdot i .. \text{MkI} \cdot (n+i) - 1]$   
 ⟨*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*:  
 $finite\text{-}list\ xs \implies last \cdot (xs ++ [x]) = x$   
*<proof>*

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

**lemma** *append-Nil2* [simp]:  
 $xs ++ [] = xs$   
*<proof>*

**lemma** *append-assoc* [simp]:  
 $(xs ++ ys) ++ zs = xs ++ ys ++ zs$   
*<proof>*

**lemma** *concat-simps* [simp]:  
 $concat \cdot [] = []$   
 $concat \cdot (xs : xss) = xs ++ concat \cdot xss$   
 $concat \cdot \perp = \perp$   
*<proof>*

**lemma** *concatMap-simps* [simp]:  
 $concatMap \cdot f \cdot [] = []$   
 $concatMap \cdot f \cdot (x : xs) = f \cdot x ++ concatMap \cdot f \cdot xs$   
 $concatMap \cdot f \cdot \perp = \perp$   
*<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]:  
 $filter \cdot (\Lambda \_ \cdot TT) \cdot xs = xs$   
*<proof>*

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

$\langle \text{proof} \rangle$

**lemma** *inits-strict* [*simp*]:

$\text{inits}.\perp = \perp$

$\langle \text{proof} \rangle$

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

$\text{the-and}.\perp = \perp$

$\langle \text{proof} \rangle$

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

$\text{the-or}.\perp = \perp$

$\langle \text{proof} \rangle$

**lemma** *all-strict* [*simp*]:

$\text{all}.P.\perp = \perp$

$\langle \text{proof} \rangle$

**lemma** *any-strict* [*simp*]:

$\text{any}.P.\perp = \perp$

$\langle \text{proof} \rangle$

**lemma** *tails-neq-Nil* [*simp*]:

$\text{tails}.xs \neq []$

$\langle \text{proof} \rangle$

**lemma** *inits-neq-Nil* [*simp*]:

$\text{inits}.xs \neq []$

$\langle \text{proof} \rangle$

**lemma** *Nil-neq-tails* [*simp*]:

$[] \neq \text{tails}.xs$

$\langle \text{proof} \rangle$

**lemma** *Nil-neq-inits* [*simp*]:

$[] \neq \text{inits}.xs$

$\langle \text{proof} \rangle$

**lemma** *finite-list-not-bottom* [*simp*]:

**assumes** *finite-list* *xs* **shows**  $xs \neq \perp$

$\langle \text{proof} \rangle$

**lemma** *head-append* [*simp*]:

$\text{head}.(xs ++ ys) = \text{If } \text{null}.xs \text{ then } \text{head}.ys \text{ else } \text{head}.xs$

$\langle \text{proof} \rangle$

**lemma** *filter-cong*:

$\forall x \in \text{set } xs. p.x = q.x \implies \text{filter}.p.xs = \text{filter}.q.xs$

$\langle \text{proof} \rangle$

**lemma** *filter-TT* [*simp*]:  
**assumes**  $\forall x \in \text{set } xs. P \cdot x = TT$   
**shows**  $\text{filter} \cdot P \cdot xs = xs$   
 $\langle \text{proof} \rangle$

**lemma** *filter-FF* [*simp*]:  
**assumes** *finite-list*  $xs$   
**and**  $\forall x \in \text{set } xs. P \cdot x = FF$   
**shows**  $\text{filter} \cdot P \cdot xs = []$   
 $\langle \text{proof} \rangle$

**lemma** *map-cong*:  
 $\forall x \in \text{set } xs. p \cdot x = q \cdot x \implies \text{map} \cdot p \cdot xs = \text{map} \cdot q \cdot xs$   
 $\langle \text{proof} \rangle$

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

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

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

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

**lemma** *Nil-append-iff* [*iff*]:  
 $xs ++ ys = [] \iff xs = [] \wedge ys = []$   
 $\langle \text{proof} \rangle$

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

**lemma** *bottom-neq-Nil* [*simp*]:  $\perp \neq []$   
 $\langle \text{proof} \rangle$

Simproc to rewrite  $[] = x$  to  $x = []$ .  
 $\langle ML \rangle$

**lemma** *set-append* [*simp*]:  
**assumes** *finite-list xs*  
**shows**  $\text{set } (xs ++ ys) = \text{set } xs \cup \text{set } ys$   
 $\langle \text{proof} \rangle$

**lemma** *distinct-Cons* [*simp*]:  
 $\text{distinct } (x : xs) \longleftrightarrow \text{distinct } xs \wedge x \notin \text{set } xs$   
**(is ?l = ?r)**  
 $\langle \text{proof} \rangle$

**lemma** *finite-list-append* [*iff*]:  
 $\text{finite-list } (xs ++ ys) \longleftrightarrow \text{finite-list } xs \wedge \text{finite-list } ys$   
**(is ?l = ?r)**  
 $\langle \text{proof} \rangle$

**lemma** *distinct-append* [*simp*]:  
**assumes** *finite-list (xs ++ ys)*  
**shows**  $\text{distinct } (xs ++ ys) \longleftrightarrow \text{distinct } xs \wedge \text{distinct } ys \wedge \text{set } xs \cap \text{set } ys = \{\}$   
**(is ?P xs ys)**  
 $\langle \text{proof} \rangle$

**lemma** *finite-set* [*simp*]:  
**assumes** *distinct xs*  
**shows** *finite (set xs)*  
 $\langle \text{proof} \rangle$

**lemma** *distinct-card*:  
**assumes** *distinct xs*  
**shows**  $\text{MkI} \cdot (\text{int } (\text{card } (\text{set } xs))) = \text{length} \cdot xs$   
 $\langle \text{proof} \rangle$

**lemma** *set-delete* [*simp*]:  
**fixes**  $xs :: ['a::Eq-eq]$   
**assumes** *distinct xs*  
**and**  $\forall x \in \text{set } xs. \text{eq} \cdot a \cdot x \neq \perp$   
**shows**  $\text{set } (\text{delete} \cdot a \cdot xs) = \text{set } xs - \{a\}$   
 $\langle \text{proof} \rangle$

**lemma** *distinct-delete* [*simp*]:  
**fixes**  $xs :: ['a::Eq-eq]$   
**assumes** *distinct xs*  
**and**  $\forall x \in \text{set } xs. \text{eq} \cdot a \cdot x \neq \perp$   
**shows** *distinct (delete · a · xs)*  
 $\langle \text{proof} \rangle$

**lemma** *set-diff* [*simp*]:  
**fixes**  $xs \ ys :: ['a::Eq-eq]$   
**assumes** *distinct ys* **and** *distinct xs*  
**and**  $\forall a \in \text{set } ys. \forall x \in \text{set } xs. \text{eq} \cdot a \cdot x \neq \perp$

**shows**  $set (xs \setminus ys) = set\ xs - set\ ys$   
 ⟨proof⟩

**lemma** *distinct-delete-filter*:  
**fixes**  $xs :: [a::Eq\ eq]$   
**assumes** *distinct xs*  
**and**  $\forall x \in set\ xs. eq\ a\ x \neq \perp$   
**shows**  $delete\ a\ xs = filter\ (\lambda x. neg\ a\ x)\ xs$   
 ⟨proof⟩

**lemma** *distinct-diff-filter*:  
**fixes**  $xs\ ys :: [a::Eq\ eq]$   
**assumes** *finite-list ys*  
**and** *distinct xs*  
**and**  $\forall a \in set\ ys. \forall x \in set\ xs. eq\ a\ x \neq \perp$   
**shows**  $xs \setminus ys = filter\ (\lambda x. neg\ (elem\ x\ ys))\ xs$   
 ⟨proof⟩

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

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

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

**lemma** *upto-eq-bottom-iff* [*simp*]:  
 $upto\ m\ n = \perp \longleftrightarrow m = \perp \vee n = \perp$   
 ⟨proof⟩

**lemma** *seq-eq-bottom-iff* [*simp*]:  
 $seq\ x\ y = \perp \longleftrightarrow x = \perp \vee y = \perp$   
 ⟨proof⟩

**lemma** *intsFrom-eq-bottom-iff* [*simp*]:  
 $intsFrom\ m = \perp \longleftrightarrow m = \perp$   
 ⟨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**  $\text{filter} \cdot (\Lambda (MkI \cdot i) \cdot \text{Def} (P i)) \cdot [MkI \cdot (n+1) ..] = \text{filter} \cdot (\Lambda (MkI \cdot i) \cdot \text{Def} (P i)) \cdot [MkI \cdot n' ..]$   
 $\langle \text{proof} \rangle$

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

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

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

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

**lemma** *length-strict* [simp]:  
 $\text{length} \cdot \perp = \perp$   
 $\langle \text{proof} \rangle$

**lemma** *repeat-neq-bottom* [simp]:  
 $\text{repeat} \cdot x \neq \perp$   
 $\langle \text{proof} \rangle$

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

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

**lemma** *replicate-strict* [simp]:  
 $\text{replicate} \cdot \perp \cdot x = \perp$   
 $\langle \text{proof} \rangle$

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

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

**lemma** *replicate-MkI-plus-1* [simp]:

$$0 \leq n \implies \text{replicate} \cdot (\text{MkI} \cdot (n+1)) \cdot x = x : \text{replicate} \cdot (\text{MkI} \cdot n) \cdot x$$

$$0 \leq n \implies \text{replicate} \cdot (\text{MkI} \cdot (1+n)) \cdot x = x : \text{replicate} \cdot (\text{MkI} \cdot n) \cdot x$$

*<proof>*

**lemma** *replicate-append-plus-conv*:

**assumes**  $0 \leq m$  **and**  $0 \leq n$

**shows**  $\text{replicate} \cdot (\text{MkI} \cdot m) \cdot x ++ \text{replicate} \cdot (\text{MkI} \cdot n) \cdot x$

$$= \text{replicate} \cdot (\text{MkI} \cdot m + \text{MkI} \cdot n) \cdot x$$

*<proof>*

**lemma** *replicate-MkI-1* [simp]:

$$\text{replicate} \cdot (\text{MkI} \cdot 1) \cdot x = x : []$$

*<proof>*

**lemma** *length-replicate* [simp]:

**assumes**  $0 \leq n$

**shows**  $\text{length} \cdot (\text{replicate} \cdot (\text{MkI} \cdot n) \cdot x) = \text{MkI} \cdot n$

*<proof>*

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

$$\text{map} \cdot f \cdot (\text{map} \cdot g \cdot xs) = \text{map} \cdot (f \text{ oo } g) \cdot xs$$

*<proof>*

**lemma** *nth-Cons-MkI* [simp]:

$$0 < i \implies (a : xs) !! (\text{MkI} \cdot i) = xs !! (\text{MkI} \cdot (i - 1))$$

*<proof>*

**lemma** *map-plus-intsFrom*:

$$\text{map} \cdot (+ \text{MkI} \cdot n) \cdot (\text{intsFrom} \cdot (\text{MkI} \cdot m)) = \text{intsFrom} \cdot (\text{MkI} \cdot (m+n)) \text{ (is ?l = ?r)}$$

*<proof>*

**lemma** *plus-eq-MkI-conv*:

$$l + n = \text{MkI} \cdot m \iff (\exists l' n'. l = \text{MkI} \cdot l' \wedge n = \text{MkI} \cdot n' \wedge m = l' + n')$$

*<proof>*

**lemma** *length-ge-0*:

$$\text{length} \cdot xs = \text{MkI} \cdot n \implies n \geq 0$$

*<proof>*

**lemma** *length-0-conv* [simp]:

$$\text{length} \cdot xs = \text{MkI} \cdot 0 \iff xs = []$$

*<proof>*

**lemma** *length-ge-1* [simp]:

$$\text{length} \cdot xs = \text{MkI} \cdot (1 + \text{int } n)$$

$$\iff (\exists u \text{ us. } xs = u : us \wedge \text{length} \cdot us = \text{MkI} \cdot (\text{int } n))$$

(is ?l = ?r)



*<proof>*

**lemma** *finite-list-length-conv*:

*finite-list*  $xs \longleftrightarrow (\exists n. \text{length}\cdot xs = \text{MkI}\cdot(\text{int } n))$  (**is**  $?l = ?r$ )  
*<proof>*

**lemma** *nth-append*:

**assumes**  $\text{length}\cdot xs = \text{MkI}\cdot n$  **and**  $n \leq m$   
**shows**  $(xs ++ ys) !! \text{MkI}\cdot m = ys !! \text{MkI}\cdot(m - n)$   
*<proof>*

**lemma** *replicate-nth [simp]*:

**assumes**  $0 \leq n$   
**shows**  $(\text{replicate}\cdot(\text{MkI}\cdot n)\cdot x ++ xs) !! \text{MkI}\cdot n = xs !! \text{MkI}\cdot 0$   
*<proof>*

**lemma** *map2-zip*:

$\text{map}\cdot(\Lambda\langle x, y\rangle. \langle x, f\cdot y\rangle)\cdot(\text{zip}\cdot xs\cdot ys) = \text{zip}\cdot xs\cdot(\text{map}\cdot f\cdot ys)$   
*<proof>*

**lemma** *map2-filter*:

$\text{map}\cdot(\Lambda\langle x, y\rangle. \langle x, f\cdot y\rangle)\cdot(\text{filter}\cdot(\Lambda\langle x, y\rangle. P\cdot x)\cdot xs)$   
 $= \text{filter}\cdot(\Lambda\langle x, y\rangle. P\cdot x)\cdot(\text{map}\cdot(\Lambda\langle x, y\rangle. \langle x, f\cdot y\rangle)\cdot xs)$   
*<proof>*

**lemma** *map-map-snd*:

$f\cdot\perp = \perp \implies \text{map}\cdot f\cdot(\text{map}\cdot \text{snd}\cdot xs)$   
 $= \text{map}\cdot \text{snd}\cdot(\text{map}\cdot(\Lambda\langle x, y\rangle. \langle x, f\cdot y\rangle)\cdot xs)$   
*<proof>*

**lemma** *findIndices-Cons [simp]*:

$\text{findIndices}\cdot P\cdot(a : xs) =$   
*If*  $P\cdot a$  *then*  $0 : \text{map}\cdot(+1)\cdot(\text{findIndices}\cdot P\cdot xs)$   
*else*  $\text{map}\cdot(+1)\cdot(\text{findIndices}\cdot P\cdot xs)$   
*<proof>*

**lemma** *filter-alt-def*:

**fixes**  $xs :: [!a]$   
**shows**  $\text{filter}\cdot P\cdot xs = \text{map}\cdot(\text{nth}\cdot xs)\cdot(\text{findIndices}\cdot P\cdot xs)$   
*<proof>*

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

**lemma** *set-map*:

$\text{set } (\text{map}\cdot f\cdot xs) = f \cdot \text{set } xs$  (**is**  $?l = ?r$ )  
*<proof>*

## 8.5 reverse and reverse induction

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

**lemma** *reverse-Nil* [*simp*]:

$reverse.\ [] = []$   
*<proof>*

**lemma** *reverse-singleton* [*simp*]:

$reverse.[x] = [x]$   
*<proof>*

**lemma** *reverse-strict* [*simp*]:

$reverse.\ \perp = \perp$   
*<proof>*

**lemma** *foldl-flip-Cons-append*:

$foldl.(flip.(:)).ys.xs = foldl.(flip.(:)).[].xs ++ ys$   
*<proof>*

**lemma** *reverse-Cons* [*simp*]:

$reverse.(x:xs) = reverse.xs ++ [x]$   
*<proof>*

**lemma** *reverse-append-below*:

$reverse.(xs ++ ys) \sqsubseteq reverse.ys ++ reverse.xs$   
*<proof>*

**lemma** *reverse-reverse-below*:

$reverse.(reverse.xs) \sqsubseteq xs$   
*<proof>*

**lemma** *reverse-append* [*simp*]:

**assumes** *finite-list xs*

**shows**  $reverse.(xs ++ ys) = reverse.ys ++ reverse.xs$   
*<proof>*

**lemma** *reverse-spine-strict*:

$\neg \text{finite-list } xs \implies reverse.xs = \perp$   
*<proof>*

**lemma** *reverse-finite* [*simp*]:

**assumes** *finite-list xs* **shows** *finite-list (reverse.xs)*  
*<proof>*

**lemma** *reverse-reverse* [*simp*]:

**assumes** *finite-list xs* **shows**  $reverse.(reverse.xs) = xs$   
*<proof>*

**lemma** *reverse-induct* [*consumes 1, case-names Nil snoc*]:  

$$\llbracket \text{finite-list } xs; P \llbracket; \bigwedge x \text{ xs} . \text{finite-list } xs \implies P \text{ xs} \implies P (xs ++ [x]) \rrbracket \implies P \text{ xs}$$
*<proof>*

**lemma** *length-plus-not-0*:  

$$le \cdot 1 \cdot n = TT \implies le \cdot (\text{length} \cdot xs + n) \cdot 0 = TT \implies \text{False}$$
*<proof>*

**lemma** *take-length-plus-1*:  

$$\text{length} \cdot xs \neq \perp \implies \text{take} \cdot (\text{length} \cdot xs + 1) \cdot (y : ys) = y : \text{take} \cdot (\text{length} \cdot xs) \cdot ys$$
*<proof>*

**lemma** *le-length-plus*:  

$$\text{length} \cdot xs \neq \perp \implies n \neq \perp \implies le \cdot n \cdot (\text{length} \cdot xs + n) = TT$$
*<proof>*

**lemma** *eq-take-length-isPrefixOf*:  

$$eq \cdot xs \cdot (\text{take} \cdot (\text{length} \cdot xs) \cdot ys) \sqsubseteq \text{isPrefixOf} \cdot xs \cdot ys$$
*<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* → (*'a* → *'b*) → *'a Maybe* → *'b* **where**  
*maybe* ≡ *Maybe-case*

**fixrec** *isJust* :: *'a Maybe* → *tr* **where**

*isJust* · (*Just* · *a*) = *TT* |

*isJust* · *Nothing* = *FF*

**fixrec** *isNothing* :: *'a Maybe* → *tr* **where**

*isNothing* = *neg oo isJust*

**fixrec** *fromJust* :: *'a Maybe* → *'a* **where**

*fromJust* · (*Just* · *a*) = *a* |

*fromJust* · *Nothing* =  $\perp$

**fixrec** *fromMaybe* :: *'a* → *'a Maybe* → *'a* **where**

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

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

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

**fixrec**  $catMaybes :: [a \text{ Maybe}] \rightarrow [a]$  **where**  
 $catMaybes = concatMap \cdot maybeToList$

**fixrec**  $mapMaybe :: ('a \rightarrow 'b \text{ Maybe}) \rightarrow [a] \rightarrow [b]$  **where**  
 $mapMaybe \cdot f = catMaybes \circ\circ map \cdot f$

**instantiation**  $Maybe :: (Eq) \text{ Eq-strict}$   
**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**  $\langle proof \rangle$

**end**

**lemma**  $eq\text{-Maybe-simps}$   $[simp]$ :

$eq \cdot Nothing \cdot Nothing = TT$   
 $eq \cdot Nothing \cdot (Just \cdot y) = FF$   
 $eq \cdot (Just \cdot x) \cdot Nothing = FF$   
 $eq \cdot (Just \cdot x) \cdot (Just \cdot y) = eq \cdot x \cdot y$   
 $\langle proof \rangle$

**instance**  $Maybe :: (Eq\text{-sym}) \text{ Eq-sym}$   
 $\langle proof \rangle$

**instance**  $Maybe :: (Eq\text{-equiv}) \text{ Eq-equiv}$   
 $\langle proof \rangle$

**instance**  $Maybe :: (Eq\text{-eq}) \text{ Eq-eq}$   
 $\langle proof \rangle$

**instantiation**  $Maybe :: (Ord) \text{ Ord-strict}$   
**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**  $\langle proof \rangle$

**end**

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

*compare.Nothing.Nothing* = *EQ*

*compare.Nothing.(Just.y)* = *LT*

*compare.(Just.x).Nothing* = *GT*

*compare.(Just.x).(Just.y)* = *compare.x.y*

$\langle proof \rangle$

**instance** *Maybe* :: (*Ord-linear*) *Ord-linear*

$\langle proof \rangle$

**lemma** *isJust-strict* [simp]: *isJust*. $\perp$  =  $\perp$   $\langle proof \rangle$

**lemma** *fromMaybe-strict* [simp]: *fromMaybe*. $x$ . $\perp$  =  $\perp$   $\langle proof \rangle$

**lemma** *maybeToList-strict* [simp]: *maybeToList*. $\perp$  =  $\perp$   $\langle proof \rangle$

**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*  $\Rightarrow$  *bool* **where**

*defined*  $x$  = ( $x \neq \perp$ )

**lemma** *defined-bottom* [simp]:  $\neg$  *defined*  $\perp$

$\langle proof \rangle$

**lemma** *defined-seq* [simp]: *defined*  $x \Longrightarrow$  *seq*. $x$ . $y$  =  $y$

$\langle proof \rangle$

**consts** *val* :: 'a::*type*  $\Rightarrow$  'b::*type* ( $\llbracket - \rrbracket$ )

*val* for booleans

**definition** *val-Bool* :: *tr*  $\Rightarrow$  *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*  $\implies$  [[ *b* ]]  $\implies$  (If *b* then *x* else *y*) = *x*  
*defined* *b*  $\implies$  [[ *b* ]] = False  $\implies$  (If *b* then *x* else *y*) = *y*  
⟨*proof*⟩

**lemma** *defined-neg* [simp]: *defined* (neg.*b*)  $\longleftrightarrow$  *defined* *b*

⟨*proof*⟩

**lemma** *val-Bool-neg* [simp]: *defined* *b*  $\implies$  [[ neg . *b* ]] = ( $\neg$  [[ *b* ]])

⟨*proof*⟩

val for integers

**definition** *val-Integer* :: Integer  $\Rightarrow$  int **where**

*val-Integer* *i* = (THE *j*. *i* = MkI.*j*)

**ad hoc-overloading**

val *val-Integer*

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

*defined* (MkI.*i*)  
*defined* (0::Integer)  
*defined* (1::Integer)  
⟨*proof*⟩

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

⟨*proof*⟩

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

[[MkI.*i*]] = *i*  
[[0]] = 0  
[[1]] = 1  
⟨*proof*⟩

**lemma** *val-Integer-numeral* [simp]: [[ numeral *x* :: Integer ]] = numeral *x*

*<proof>*

**lemma** *val-Integer-to-MkI*:  
defined  $i \implies i = (\text{MkI} \cdot \llbracket i \rrbracket)$   
*<proof>*

**lemma** *defined-Integer-minus [simp]*: defined  $i \implies$  defined  $j \implies$  defined  $(i - (j::\text{Integer}))$   
*<proof>*

**lemma** *val-Integer-minus [simp]*: defined  $i \implies$  defined  $j \implies \llbracket i - j \rrbracket = \llbracket i \rrbracket - \llbracket j \rrbracket$   
*<proof>*

**lemma** *defined-Integer-plus [simp]*: defined  $i \implies$  defined  $j \implies$  defined  $(i + (j::\text{Integer}))$   
*<proof>*

**lemma** *val-Integer-plus [simp]*: defined  $i \implies$  defined  $j \implies \llbracket i + j \rrbracket = \llbracket i \rrbracket + \llbracket j \rrbracket$   
*<proof>*

**lemma** *defined-Integer-eq [simp]*: defined  $(\text{eq} \cdot a \cdot b) \iff$  defined  $a \wedge$  defined  $(b::\text{Integer})$   
*<proof>*

**lemma** *val-Integer-eq [simp]*: defined  $a \implies$  defined  $b \implies \llbracket \text{eq} \cdot a \cdot b \rrbracket = (\llbracket a \rrbracket = \llbracket b \rrbracket :: \text{int})$   
*<proof>*

Full induction for non-negative integers

**lemma** *nonneg-full-Int-induct [consumes 1, case-names neg Suc]*:  
assumes defined: defined  $i$   
assumes neg:  $\bigwedge i. \text{defined } i \implies \llbracket i \rrbracket < 0 \implies P i$   
assumes step:  $\bigwedge i. \text{defined } i \implies 0 \leq \llbracket i \rrbracket \implies (\bigwedge j. \text{defined } j \implies \llbracket j \rrbracket < \llbracket i \rrbracket \implies P j) \implies P i$   
shows  $P (i::\text{Integer})$   
*<proof>*

Some list lemmas re-done with the new setup.

**lemma** *nth-tail*:  
defined  $n \implies \llbracket n \rrbracket \geq 0 \implies \text{tail} \cdot xs !! n = xs !! (1 + n)$   
*<proof>*

**lemma** *nth-zip With*:  
assumes  $f1 [simp]$ :  $\bigwedge y. f \cdot \perp \cdot y = \perp$   
assumes  $f2 [simp]$ :  $\bigwedge x. f \cdot x \cdot \perp = \perp$   
shows  $\text{zip With} \cdot f \cdot xs \cdot ys !! n = f \cdot (xs !! n) \cdot (ys !! n)$   
*<proof>*

**lemma** *nth-neg* [*simp*]:  $\text{defined } n \implies \llbracket n \rrbracket < 0 \implies \text{nth} \cdot \text{xs} \cdot n = \perp$   
 ⟨*proof*⟩

**lemma** *nth-Cons-simp* [*simp*]:  
 $\text{defined } n \implies \llbracket n \rrbracket = 0 \implies \text{nth} \cdot (x : \text{xs}) \cdot n = x$   
 $\text{defined } n \implies \llbracket n \rrbracket > 0 \implies \text{nth} \cdot (x : \text{xs}) \cdot n = \text{nth} \cdot \text{xs} \cdot (n - 1)$   
 ⟨*proof*⟩

**end**

## 11 List Comprehension

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

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

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

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

**translations**  
 $[e \mid p \leftarrow \text{xs}] \Rightarrow \text{CONST } \text{concatMap} \cdot (\text{-llc-abs } p \ [e]) \cdot \text{xs}$   
 $\text{-llc } e \ (\text{-llc-gen } p \ \text{xs}) \ (\text{-llc-quals } q \ \text{qs})$   
 $\Rightarrow \text{CONST } \text{concatMap} \cdot (\text{-llc-abs } p \ (\text{-llc } e \ q \ \text{qs})) \cdot \text{xs}$   
 $[e \mid b] \Rightarrow \text{If } b \ \text{then } [e] \ \text{else } []$   
 $\text{-llc } e \ (\text{-llc-guard } b) \ (\text{-llc-quals } q \ \text{qs})$   
 $\Rightarrow \text{If } b \ \text{then } (\text{-llc } e \ q \ \text{qs}) \ \text{else } []$   
 $\text{-llc } e \ (\text{-llc-let } b) \ (\text{-llc-quals } q \ \text{qs})$   
 $\Rightarrow \text{-Let } b \ (\text{-llc } e \ q \ \text{qs})$

⟨*ML*⟩

**lemma** *concatMap-singleton* [*simp*]:  
 $\text{concatMap} \cdot (\lambda x. [f \cdot x]) \cdot \text{xs} = \text{map} \cdot f \cdot \text{xs}$   
 ⟨*proof*⟩

**lemma** *listcompr-filter* [*simp*]:  
 $[x \mid x \leftarrow \text{xs}, P \cdot x] = \text{filter} \cdot P \cdot \text{xs}$   
 ⟨*proof*⟩



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

```
end
```

## 12 The Num Class

```
theory Num-Class
```

```
  imports
```

```
    Type-Classes
```

```
    Data-Integer
```

```
    Data-Tuple
```

```
begin
```

### 12.1 Num class

```
class Num-syn =
```

```
  Eq +
```

```
  plus +
```

```
  minus +
```

```
  times +
```

```
  zero +
```

```
  one +
```

```
  fixes negate :: 'a → 'a
```

```
  and abs :: 'a → 'a
```

```
  and signum :: 'a → 'a
```

```
  and fromInteger :: Integer → 'a
```

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

```
class Num-strict = Num +
```

```
  assumes plus-strict[simp]:
```

```
    x + ⊥ = (⊥::'a::Num)
```

```
    ⊥ + x = ⊥
```

```
  assumes minus-strict[simp]:
```

```
    x - ⊥ = ⊥
```

```
    ⊥ - x = ⊥
```

```
  assumes mult-strict[simp]:
```

```
    x * ⊥ = ⊥
```

```
    ⊥ * x = ⊥
```

```
  assumes negate-strict[simp]:
```

```
    negate.⊥ = ⊥
```

```
  assumes abs-strict[simp]:
```

```
    abs.⊥ = ⊥
```

```
  assumes signum-strict[simp]:
```

```
    signum.⊥ = ⊥
```

```
  assumes fromInteger-strict[simp]:
```

$fromInteger \cdot \perp = \perp$

**class** *Num-faithful* =

*Num-syn* +

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

**class** *Integral* =

*Num* +

**fixes** *div mod* ::  $'a \rightarrow 'a \rightarrow ('a :: Num)$

**fixes** *toInteger* ::  $'a \rightarrow Integer$

**begin**

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

**fixrec** *even* ::  $'a \rightarrow tr$  **where**  $even \cdot x = eq \cdot (div \cdot x \cdot (fromInteger \cdot 2)) \cdot 0$

**fixrec** *odd* ::  $'a \rightarrow tr$  **where**  $odd \cdot x = neg \cdot (even \cdot x)$

**end**

**class** *Integral-strict* = *Integral* +

**assumes** *div-strict[simp]*:

$div \cdot x \cdot \perp = (\perp :: 'a :: Integral)$

$div \cdot \perp \cdot x = \perp$

**assumes** *mod-strict[simp]*:

$mod \cdot x \cdot \perp = \perp$

$mod \cdot \perp \cdot x = \perp$

**assumes** *toInteger-strict[simp]*:

$toInteger \cdot \perp = \perp$

**class** *Integral-faithful* =

*Integral* +

*Num-faithful* +

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

## 12.2 Instances for Integer

**instantiation** *Integer* :: *Num-syn*

**begin**

**definition** *negate* = ( $\Lambda (MkI \cdot x). MkI \cdot (uminus \ x)$ )

**definition** *abs* = ( $\Lambda (MkI \cdot x). MkI \cdot (|x|)$ )

```

definition signum = (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·(+).fibs·(tail·fibs)

```

```

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

```

```

declare fib.simps [simp del]

```

```

lemma fibs-0 [simp]:
  fibs !! 0 = 0
  ⟨proof⟩

```

```

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

```

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

```

lemma [simp]: -1 +  $\llbracket i \rrbracket$  =  $\llbracket i \rrbracket$  - 1 ⟨proof⟩

```

```

lemma [simp]: -2 +  $\llbracket i \rrbracket$  =  $\llbracket i \rrbracket$  - 2 ⟨proof⟩

```

```

lemma nth-fibs:

```

```

  assumes defined i and  $\llbracket i \rrbracket \geq 0$  shows defined (fibs !! i) and  $\llbracket fibs !! i \rrbracket = fib$ 
   $\llbracket i \rrbracket$ 
  ⟨proof⟩

```

```

end

```

```

theory Sieve-Primes

```

```

  imports

```

```

    HOL-Computational-Algebra.Primes

```

```

    ../Num-Class

```

```

    ../HOLCF-Prelude

```

```

begin

```

## 14 The Sieve of Eratosthenes

```

declare  $\llbracket coercion\ int \rrbracket$ 

```

```

declare  $\llbracket coercion-enabled \rrbracket$ 

```

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  $\exists n'. n' > n \wedge \text{prime } n' \wedge (\forall k. n < k \longrightarrow k < n' \longrightarrow \neg \text{prime } k)$ 
<proof>

```

Simplification for andalso:

```

lemma andAlso-Def[simp]: ((Def x) andalso (Def y)) = Def (x  $\wedge$  y)
<proof>

```

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

**definition** prim-bisim:

```

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

```

```

lemma prim-bisim-is-bisim: list-bisim prim-bisim
<proof>

```

Now we apply coinduction:

**lemma** sieve-produces-primes:

```

fixes n :: nat
assumes prime n
shows sieve.(filter.( $\Lambda$  (MkI.i). Def (( $\forall d::\text{int}. d > 1 \longrightarrow d < n \longrightarrow \neg (d \text{ dvd } i)$ )))).[MkI.n..])
  = filter.( $\Lambda$  (MkI.i). Def (prime (nat |i|))).[MkI.n..]
<proof>

```

And finally show the correctness of primes.

**theorem** primes:

```

shows primes = filter.( $\Lambda$  (MkI.i). Def (prime (nat |i|))).[MkI.2..]
<proof>

```

**end**

## 15 GHC's "fold/build" Rule

**theory** GHC-Rewrite-Rules

**imports** ../HOLCF-Prelude

**begin**

### 15.1 Approximating the Rewrite Rule

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

```

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

```

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

**abbreviation** (*input*) *abstract-list* **where**

$abstract-list\ xs \equiv (\Lambda\ c\ n.\ foldr \cdot c \cdot n \cdot xs)$

**cpodef** ( $'a, 'b$ ) *listfun* =

$\{(f :: ('a \rightarrow 'b \rightarrow 'b) \rightarrow 'b \rightarrow 'b, xs). f = abstract-list\ xs\}$   
 $\langle proof \rangle$

**definition** *listfun* :: ( $'a, 'b$ ) *listfun*  $\rightarrow ('a \rightarrow 'b \rightarrow 'b) \rightarrow 'b \rightarrow 'b$  **where**

$listfun = (\Lambda\ g.\ Product-Type.fst\ (Rep-listfun\ g))$

**definition** *build* :: ( $'a, 'b$ ) *listfun*  $\rightarrow ['a]$  **where**

$build = (\Lambda\ g.\ Product-Type.snd\ (Rep-listfun\ g))$

**definition** *augment* :: ( $'a, 'b$ ) *listfun*  $\rightarrow ['a] \rightarrow ['a]$  **where**

$augment = (\Lambda\ g\ xs.\ build \cdot g\ ++\ xs)$

**definition** *listfun-comp* :: ( $'a, 'b$ ) *listfun*  $\rightarrow ('a, 'b)$  *listfun*  $\rightarrow ('a, 'b)$  *listfun* **where**

$listfun-comp = (\Lambda\ g\ h.$

$Abs-listfun\ (\Lambda\ c\ n.\ listfun \cdot g \cdot c \cdot (listfun \cdot h \cdot c \cdot n),\ build \cdot g\ ++\ build \cdot h))$

**abbreviation**

$listfun-comp-infix :: ('a, 'b)$  *listfun*  $\Rightarrow ('a, 'b)$  *listfun*  $\Rightarrow ('a, 'b)$  *listfun* (**infixl**  $\circ lf$  55)

**where**

$g\ \circ lf\ h \equiv listfun-comp \cdot g \cdot h$

**fixrec** *mapFB* :: ( $'b \rightarrow 'c \rightarrow 'c$ )  $\rightarrow ('a \rightarrow 'b) \rightarrow 'a \rightarrow 'c \rightarrow 'c$  **where**

$mapFB \cdot c \cdot f = (\Lambda\ x\ ys.\ c \cdot (f \cdot x) \cdot ys)$

## 15.2 Lemmas

**lemma** *cont-listfun-body* [*simp*]:

$cont\ (\lambda g.\ Product-Type.fst\ (Rep-listfun\ g))$

$\langle proof \rangle$

**lemma** *cont-build-body* [*simp*]:

$cont\ (\lambda g.\ Product-Type.snd\ (Rep-listfun\ g))$

$\langle proof \rangle$

**lemma** *build-Abs-listfun*:

**assumes**  $abstract-list\ xs = f$

**shows**  $build \cdot (Abs-listfun\ (f, xs)) = xs$

$\langle proof \rangle$

**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 (λg. 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 (λh. Abs-listfun (λ c n. listfun.g.c.(listfun.h.c.n), build.g ++ build.h))*  
*<proof>*

**lemma** *cont-listfun-comp-body2* [simp]:

*cont (λg. Abs-listfun (λ c n. listfun.g.c.(listfun.h.c.n), build.g ++ build.h))*  
*<proof>*

**lemma** *cont-listfun-comp-body* [simp]:

*cont (λg. λ h. Abs-listfun (λ c n. listfun.g.c.(listfun.h.c.n), build.g ++ build.h))*  
*<proof>*

**lemma** *abstract-list-build-append*:



*abstract-list* (*build*·*g* ++ *build*·*h*) = ( $\Lambda$  *c n*. *listfun*·*g*·*c*·(*listfun*·*h*·*c*·*n*))  
 ⟨*proof*⟩

**lemma** *augment/build*:  
*augment*·*g*·(*build*·*h*) = *build*·(*g* ○*lf* *h*)  
 ⟨*proof*⟩

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

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

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

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

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

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

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

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

### 15.3 Examples

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

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

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

**definition**  $down :: 'b \text{ itself} \Rightarrow Integer \rightarrow [Integer]$  **where**  
 $down \ C\text{-type} = (\Lambda \ v. \ build.(Abs\text{-listfun} (\$   
 $(down' :: Integer \rightarrow (Integer \rightarrow 'b \rightarrow 'b) \rightarrow 'b \rightarrow 'b).v,$   
 $down'.v.(:\cdot[])))$

**lemma**  $abstract\text{-list}\text{-down}' [simp]:$   
 $abstract\text{-list} (down'.v.(:\cdot[])) = down'.v$   
 $\langle proof \rangle$

**lemma**  $cont\text{-Abs}\text{-listfun}\text{-down}' [simp]:$   
 $cont (\lambda v. Abs\text{-listfun} (down'.v, down'.v.(:\cdot[])))$   
 $\langle proof \rangle$

**lemma**  $sum\text{-down}:$   
 $sum.((down \ TYPE(Integer)).v) = down'.v.(+)\cdot 0$   
 $\langle proof \rangle$

**end**  
**theory**  $HLint$   
**imports**  
 $../HOLCF\text{-Prelude}$   
 $../List\text{-Comprehension}$   
**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**  $(eq.(x::'a::Eq\text{-sym}).a \ orelse \ eq.x.b \ orelse \ eq.x.c) = elem.x.[a, b, c]$   
 $\langle proof \rangle$

$x /= a \ \&\& \ x /= b \ \&\& \ x /= c \ ==> \ x \ 'notElem' \ [a,b,c]$

**lemma**  $(neq.(x::'a::Eq\text{-sym}).a \ andalso \ neq.x.b \ andalso \ neq.x.c) = notElem.x.[a, b, c]$   
 $\langle proof \rangle$

### 16.2 List

$concat \ (map \ f \ x) \ ==> \ concatMap \ f \ x$

**lemma**  $concat.(map.f.x) = concatMap.f.x$

```

    <proof>
concat [a, b] ==> a ++ b
lemma concat.[a, b] = a ++ b
    <proof>
map f (map g x) ==> map (f . g) x
lemma map.f.(map.g.x) = map.(f oo g).x
    <proof>
x !! 0 ==> head x
lemma x !! 0 = head.x
    <proof>
take n (repeat x) ==> replicate n x
lemma take.n.(repeat.x) = replicate.n.x
    <proof>
lemma "head\<cdot>(reverse\<cdot>x) = last\<cdot>x"
lemma head.(reverse.x) = last.x
    <proof>
head (drop n x) ==> x !! n where note = "if the index is non-negative"
lemma
    assumes le.0.n ≠ FF
    shows head.(drop.n.x) = x !! n
    <proof>
reverse (tail (reverse x)) ==> init x
lemma reverse.(tail.(reverse.x)) ⊆ init.x
    <proof>
take (length x - 1) x ==> init x
lemma
    assumes x ≠ []
    shows take.(length.x - 1).x ⊆ init.x
    <proof>
foldr (++) [] ==> concat
lemma foldr-append-concat:foldr.append.[] = concat
    <proof>
foldl (++) [] ==> concat
lemma foldl.append.[] ⊆ concat
    <proof>
span (not . p) ==> break p

```

```

lemma span.(neg oo p) = break.p
  <proof>

  break (not . p) ==> span p
lemma break.(neg oo p) = span.p
  <proof>

  or (map p x) ==> any p x
lemma the-or.(map.p.x) = any.p.x
  <proof>

  and (map p x) ==> all p x
lemma the-and.(map.p.x) = all.p.x
  <proof>

  zipWith (,) ==> zip
lemma zipWith.<,> = zip
  <proof>

  zipWith3 (,,) ==> zip3
lemma zipWith3.<,,> = zip3
  <proof>

  length x == 0 ==> null x where note = "increases laziness"
lemma eq.(length.x).0 ⊆ null.x
  <proof>

  length x /= 0 ==> not (null x)
lemma neq.(length.x).0 ⊆ neg.(null.x)
  <proof>

  map (uncurry f) (zip x y) ==> zipWith f x y
lemma map.(uncurry.f).(zip.x.y) = zipWith.f.x.y
  <proof>

  map f (zip x y) ==> zipWith (curry f) x y where _ = isVar f
lemma map.f.(zip.x.y) = zipWith.(curry.f).x.y
  <proof>

  not (elem x y) ==> notElem x y
lemma neg.(elem.x.y) = notElem.x.y
  <proof>

  foldr f z (map g x) ==> foldr (f . g) z x
lemma foldr.f.z.(map.g.x) = foldr.(f oo g).z.x
  <proof>

```

```

null (filter f x) ==> not (any f x)
lemma null.(filter.f.x) = neg.(any.f.x)
<proof>

filter f x == [] ==> not (any f x)
lemma eq.(filter.f.x).[] = neg.(any.f.x)
<proof>

filter f x /= [] ==> any f x
lemma neq.(filter.f.x).[] = any.f.x
<proof>

any (== a) ==> elem a
lemma any.( $\Lambda z. eq.z.a$ ) = elem.a
<proof>

any ((==) a) ==> elem a
lemma any.(eq.(a::'a::Eq-sym)) = elem.a
<proof>

any (a ==) ==> elem a
lemma any.( $\Lambda z. eq.(a::'a::Eq-sym).z$ ) = elem.a
<proof>

all (/= a) ==> notElem a
lemma all.( $\Lambda z. neq.z.(a::'a::Eq-sym)$ ) = notElem.a
<proof>

all (a /=) ==> notElem a
lemma all.( $\Lambda z. neq.(a::'a::Eq-sym).z$ ) = notElem.a
<proof>

```

### 16.3 Folds

```

foldr (&&) True ==> and
lemma foldr.trand.TT = the-and
<proof>

foldl (&&) True ==> and
lemma foldl-to-and:foldl.trand.TT  $\sqsubseteq$  the-and
<proof>

foldr1 (&&) ==> and
lemma foldr1.trand  $\sqsubseteq$  the-and
<proof>

```

```

foldl1 (&&) ==> and
lemma foldl1.trand  $\sqsubseteq$  the-and
<proof>

foldr (||) False ==> or
lemma foldr.tror.FF = the-or
<proof>

foldl (||) False ==> or
lemma foldl-to-or: foldl.tror.FF  $\sqsubseteq$  the-or
<proof>

foldr1 (||) ==> or
lemma foldr1.tror  $\sqsubseteq$  the-or
<proof>

foldl1 (||) ==> or
lemma foldl1.tror  $\sqsubseteq$  the-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>

( $\Lambda$ (x,y) -> y) ==> fst where _ = notIn x y
lemma ( $\Lambda$  <x, y>. x) = fst
<proof>

( $\Lambda$ (x,y) -> y) ==> snd where _ = notIn x y
lemma ( $\Lambda$  <x, y>. y) = snd
<proof>

(\x y-> f (x,y)) ==> curry f where _ = notIn [x,y] f
lemma ( $\Lambda$  x y. f.<x, y>) = curry.f
<proof>

( $\Lambda$ (x,y) -> f x y) ==> uncurry f where _ = notIn [x,y] f
lemma ( $\Lambda$  <x, y>. f.x.y)  $\sqsubseteq$  uncurry.f
<proof>

```

$(\backslash x \rightarrow y) \implies \text{const } y$  where  $\_ = \text{isAtom } y \ \&\& \ \text{notIn } x \ y$   
**lemma**  $(\Lambda x. y) = \text{const} \cdot y$   
 $\langle \text{proof} \rangle$

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

## 16.5 Bool

$a == \text{True} \implies a$

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

$a == \text{False} \implies \text{not } a$

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

$(\text{if } a \text{ then } x \text{ else } x) \implies x$  where  $\text{note} = \text{"reduces strictness"}$

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

$(\text{if } a \text{ then } \text{True} \text{ else } \text{False}) \implies a$

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

$(\text{if } a \text{ then } \text{False} \text{ else } \text{True}) \implies \text{not } a$

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

$(\text{if } a \text{ then } t \text{ else } (\text{if } b \text{ then } t \text{ else } f)) \implies \text{if } a \ || \ b \text{ then } t \text{ else } f$

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

$(\text{if } a \text{ then } (\text{if } b \text{ then } t \text{ else } f) \text{ else } f) \implies \text{if } a \ \&\& \ b \text{ then } t \text{ else } f$

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

$(\text{if } x \text{ then } \text{True} \text{ else } y) \implies x \ || \ y$  where  $\_ = \text{notEq } y \ \text{False}$

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

$(\text{if } x \text{ then } y \text{ else } \text{False}) \implies x \ \&\& \ y$  where  $\_ = \text{notEq } y \ \text{True}$

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

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

lemma (If c then <TT, x> else <FF, x>)  $\sqsubseteq$  <c, x>  
<proof>

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

lemma (If c then <FF, x> else <TT, x>)  $\sqsubseteq$  <neg·c, x>  
<proof>

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

lemma the-or·[x, y] = (x orelse y)  
<proof>

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

lemma the-or·[x, y, z] = (x orelse y orelse z)  
<proof>

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

lemma the-and·[x, y] = (x andalso y)  
<proof>

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

lemma the-and·[x, y, z] = (x andalso y andalso z)  
<proof>

## 16.6 Arrow

(fst x, snd x) ==> x

lemma x  $\sqsubseteq$  <fst·x, snd·x>  
<proof>

## 16.7 Seq

x 'seq' x ==> x

lemma seq·x·x = x <proof>

## 16.8 Evaluate

True && x ==> x

lemma (TT andalso x) = x <proof>

False && x ==> False

lemma (FF andalso x) = FF <proof>



```

True || x ==> True
lemma (TT orelse x) = TT <proof>

False || x ==> x
lemma (FF orelse x) = x <proof>

not True ==> False
lemma neg.TT = FF <proof>

not False ==> True
lemma neg.FF = TT <proof>

fst (x,y) ==> x
lemma fst.<x, y> = x <proof>

snd (x,y) ==> y
lemma snd.<x, y> = y <proof>

f (fst p) (snd p) ==> uncurry f p
lemma f.(fst.p).(snd.p) = uncurry.f.p
<proof>

init [x] ==> []
lemma init.[x] = [] <proof>

null [] ==> True
lemma null.[] = TT <proof>

length [] ==> 0
lemma length.[] = 0 <proof>

foldl f z [] ==> z
lemma foldl.f.z.[] = z <proof>

foldr f z [] ==> z
lemma foldr.f.z.[] = z <proof>

foldr1 f [x] ==> x
lemma foldr1.f.[x] = x <proof>

scanr f z [] ==> [z]
lemma scanr.f.z.[] = [z] <proof>

scanr1 f [] ==> []
lemma scanr1.f.[] = [] <proof>

```

```

scanr1 f [x] ==> [x]
lemma scanr1.f.[x] = [x] <proof>

take n [] ==> []
lemma take.n.[] ⊆ [] <proof>

drop n [] ==> []
lemma drop.n.[] ⊆ []
<proof>

takeWhile p [] ==> []
lemma takeWhile.p.[] = [] <proof>

dropWhile p [] ==> []
lemma dropWhile.p.[] = [] <proof>

span p [] ==> ([], [])
lemma span.p.[] = ([], []) <proof>

concat [a] ==> a
lemma concat.[a] = a <proof>

concat [] ==> []
lemma concat.[] = [] <proof>

zip [] [] ==> []
lemma zip.[].[] = [] <proof>

id x ==> x
lemma ID.x = x <proof>

const x y ==> x
lemma const.x.y = x <proof>

```

## 16.9 Complex hints

```

take (length t) s == t ==> t 'Data.List.isPrefixOf' s
lemma
  fixes t :: ['a::Eq-sym]
  shows eq.(take.(length.t).s).t ⊆ isPrefixOf.t.s
  <proof>

  (take i s == t) ==> _eval_ ((i >= length t) && (t 'Data.List.isPrefixOf'
s))

```

The hint is not true in general, as the following two lemmas show:

**lemma**  
**assumes**  $t = []$  **and**  $s = x : xs$  **and**  $i = 1$   
**shows**  $\neg (eq.(take.i.s).t \sqsubseteq (le.(length.t).i \text{ andalso } isPrefixOf.t.s))$   
 $\langle proof \rangle$

**lemma**  
**assumes**  $le.0.i = TT$  **and**  $le.i.0 = FF$   
**and**  $s = \perp$  **and**  $t = []$   
**shows**  $\neg ((le.(length.t).i \text{ andalso } isPrefixOf.t.s) \sqsubseteq eq.(take.i.s).t)$   
 $\langle proof \rangle$

**lemma**  $neg.(eq.a.b) = neg.a.b \langle proof \rangle$

**not**  $(a \neq b) \implies a = b$

**lemma**  $neg.(neg.a.b) = eq.a.b \langle proof \rangle$

**map id**  $\implies id$

**lemma**  $map-id:map.ID = ID \langle proof \rangle$

$x == [] \implies null\ x$

**lemma**  $eq.x.[] = null.x \langle proof \rangle$

**any id**  $\implies or$

**lemma**  $any.ID = the-or \langle proof \rangle$

**all id**  $\implies and$

**lemma**  $all.ID = the-and \langle proof \rangle$

$(if\ x\ then\ False\ else\ y) \implies (not\ x \ \&\&\ y)$

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

$(if\ x\ then\ y\ else\ True) \implies (not\ x \ ||\ y)$

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

**not**  $(not\ x) \implies x$

**lemma**  $neg.(neg.x) = x \langle proof \rangle$

$(if\ c\ then\ f\ x\ else\ f\ y) \implies f\ (if\ c\ then\ x\ else\ y)$

**lemma**  $(If\ c\ then\ f.x\ else\ f.y) \sqsubseteq f.(If\ c\ then\ x\ else\ y) \langle proof \rangle$

$(\ \backslash\ x\ \rightarrow\ [x]) \implies (: \ [])$

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

```

True == a ==> a
lemma eq.TT.a = a <proof>
False == a ==> not a
lemma eq.FF.a = neg.a <proof>
a /= True ==> not a
lemma neg.a.TT = neg.a <proof>
a /= False ==> a
lemma neg.a.FF = a <proof>
True /= a ==> not a
lemma neg.TT.a = neg.a <proof>
False /= a ==> a
lemma neg.FF.a = a <proof>
not (isNothing x) ==> isJust x
lemma neg.(isNothing.x) = isJust.x <proof>
not (isJust x) ==> isNothing x
lemma neg.(isJust.x) = isNothing.x <proof>
x == Nothing ==> isNothing x
lemma eq.x.Nothing = isNothing.x <proof>
Nothing == x ==> isNothing x
lemma eq.Nothing.x = isNothing.x <proof>
x /= Nothing ==> Data.Maybe.isJust x
lemma neg.x.Nothing = isJust.x <proof>
Nothing /= x ==> Data.Maybe.isJust x
lemma neg.Nothing.x = isJust.x <proof>
(if isNothing x then y else fromJust x) ==> fromMaybe y x
lemma (If isNothing.x then y else fromJust.x) = fromMaybe.y.x <proof>
(if isJust x then fromJust x else y) ==> fromMaybe y x
lemma (If isJust.x then fromJust.x else y) = fromMaybe.y.x <proof>
(isJust x && (fromJust x == y)) ==> x == Just y
lemma (isJust.x andalso (eq.(fromJust.x).y)) = eq.x.(Just.y) <proof>

```

```

elem True ==> or
lemma elem·TT = the-or
⟨proof⟩

notElem False ==> and
lemma notElem·FF = the-and
⟨proof⟩

all ((/=) a) ==> notElem a
lemma all·(neg·(a::'a::Eq-sym)) = notElem·a
⟨proof⟩

maybe x id ==> Data.Maybe.fromMaybe x
lemma maybe·x·ID = fromMaybe·x
⟨proof⟩

maybe False (const True) ==> Data.Maybe.isJust
lemma maybe·FF·(const·TT) = isJust
⟨proof⟩

maybe True (const False) ==> Data.Maybe.isNothing
lemma maybe·TT·(const·FF) = isNothing
⟨proof⟩

maybe [] (: []) ==> maybeToList
lemma maybe·[]·(λ z. z : []) = maybeToList
⟨proof⟩

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

(if isNothing x then y else f (fromJust x)) ==> maybe y f x
lemma (If isNothing·x then y else f·(fromJust·x)) = maybe·y·f·x ⟨proof⟩

(if isJust x then f (fromJust x) else y) ==> maybe y f x
lemma (If isJust·x then f·(fromJust·x) else y) = maybe·y·f·x ⟨proof⟩

(map fromJust . filter isJust) ==> Data.Maybe.catMaybes
lemma (map·fromJust oo filter·isJust) = catMaybes
⟨proof⟩

concatMap (maybeToList . f) ==> Data.Maybe.mapMaybe f
lemma concatMap·(maybeToList oo f) = mapMaybe·f
⟨proof⟩

concatMap maybeToList ==> catMaybes

```

```

lemma concatMap.maybeToList = catMaybes <proof>
mapMaybe f (map g x) ==> mapMaybe (f . g) x
lemma mapMaybe.f.(map.g.x) = mapMaybe.(f oo g).x <proof>
(( $\$$ ) . f) ==> f
lemma (dollar oo f) = f <proof>
(f  $\$$ ) ==> f
lemma ( $\Lambda z. dollar.f.z$ ) = f <proof>
( $\backslash a b \rightarrow g (f a) (f b)$ ) ==> g 'Data.Function.on' f
lemma ( $\Lambda a b. g.(f.a).(f.b)$ ) = on.g.f <proof>
id  $\$!$  x ==> x
lemma dollarBang.ID.x = x <proof>
[x | x <- y] ==> y
lemma [x | x <- y] = y <proof>
isPrefixOf (reverse x) (reverse y) ==> isSuffixOf x y
lemma isPrefixOf.(reverse.x).(reverse.y) = isSuffixOf.x.y <proof>
concat (intersperse x y) ==> intercalate x y
lemma concat.(intersperse.x.y) = intercalate.x.y <proof>
x 'seq' y ==> y
lemma
  assumes  $x \neq \perp$  shows seq.x.y = y
  <proof>
f  $\$!$  x ==> f x
lemma assumes  $x \neq \perp$  shows dollarBang.f.x = f.x
  <proof>
maybe (f x) (f . g) ==> (f . maybe x g)
lemma maybe.(f.x).(f oo g)  $\sqsubseteq$  (f oo maybe.x.g)
  <proof>
end

```

## Acknowledgments

We thank Lars Hupel for his help with the final AFP submission.

## References

- [1] J. Breitner, B. Huffman, N. Mitchell, and C. Sternagel. Certified HLints with Isabelle/HOLCF-Prelude, June 2013. Haskell And Rewriting Techniques (HART).
- [2] S. Peyton Jones. Haskell 98 - Standard Prelude. *Journal of Functional Programming*, 13(1):103–124, 2003. doi:[10.1017/S0956796803001011](https://doi.org/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.