Proof-Rules for Certifying Memory Bounds

by Javier de Dios and Ricardo Peña, January 2011

Contents

1 Normal form values and heap **3**

2 Useful functions and theorems from the Haskell Library or Prelude **6**

3 Normalized Safe Expressions **12**
3.1 Free Variables 13

4 Primitive operators for SVM anf JVM **19**

5 State of the SVM **19**
5.1 Sizes Table 20
5.2 Stack 20
5.3 Code Store and SafeImp program 20
5.4 Runtime State 21

6 Definitions of Upper Bounds and Least Upper Bounds **22**
6.1 Rules for the Relations $*<=$ and $<=*$ 23
6.2 Rules about the Operators *leastP*, *ub* and *lub* 23

7 The Greatest Common Divisor **24**
7.1 Specification of GCD on nats 25
7.2 GCD on nat by Euclid's algorithm 25
7.3 Derived laws for GCD 26
7.4 LCM defined by GCD 28
7.5 GCD and LCM on integers 31

8 Abstract rational numbers **34**

9 Rational numbers **45**
9.1 Rational numbers 45
9.1.1 Equivalence of fractions 45
9.1.2 The type of rational numbers 46
9.1.3 Congruence lemmas 47

9.1.4 Standard operations on rational numbers 48
9.1.5 The ordered field of rational numbers 50
9.2 Various Other Results 53
9.3 Numerals and Arithmetic 54
9.4 Embedding from Rationals to other Fields 54
9.5 Implementation of rational numbers as pairs of integers . . . 57

10 Positive real numbers **59**
10.1 *preal-of-prat*: the Injection from prat to preal 62
10.2 Properties of Ordering 62
10.3 Properties of Addition 63
10.4 Properties of Multiplication 66
10.5 Distribution of Multiplication across Addition 70
10.6 Existence of Inverse, a Positive Real 71
10.7 Gleason's Lemma 9-3.4, page 122 73
10.8 Gleason's Lemma 9-3.6 74
10.9 Existence of Inverse: Part 2 75
10.10Subtraction for Positive Reals 78
10.11proving that $S \leq R + D$ — trickier 79
10.12Completeness of type *preal* 82
10.13The Embedding from *rat* into *preal* 83

11 Defining the Reals from the Positive Reals **85**
11.1 Equivalence relation over positive reals 87
11.2 Addition and Subtraction 88
11.3 Multiplication 89
11.4 Inverse and Division 90
11.5 The Real Numbers form a Field 91
11.6 The $\leq$ Ordering 91
11.7 The Reals Form an Ordered Field 93
11.8 Theorems About the Ordering 95
11.9 More Lemmas 95
11.10Embedding numbers into the Reals 96
11.11Embedding the Naturals into the Reals 99
11.12Numerals and Arithmetic 102
11.13Simprules combining x+y and 0: ARE THEY NEEDED? . . 102
11.13.1 Density of the Reals 103
11.14Absolute Value Function for the Reals 103
11.15Implementation of rational real numbers as pairs of integers . 104

12 Completeness of the Reals; Floor and Ceiling Functions **106**
12.1 Completeness of Positive Reals 106
12.2 The Archimedean Property of the Reals 112
12.3 Floor and Ceiling Functions from the Reals to the Integers . 114

12.4 Versions for the natural numbers . . . 125

13 Non-denumerability of the Continuum. 132
13.1 Abstract . . . 132
13.2 Closed Intervals . . . 132
13.2.1 Definition . . . 132
13.2.2 Properties . . . 132
13.3 Nested Interval Property . . . 134
13.4 Generating the intervals . . . 138
13.4.1 Existence of non-singleton closed intervals . . . 138
13.5 newInt: Interval generation . . . 139
13.5.1 Definition . . . 139
13.5.2 Properties . . . 140
13.6 Final Theorem . . . 143

14 Natural powers theory 143
14.1 Literal Arithmetic Involving Powers, Type *real* . . . 145
14.2 Properties of Squares . . . 145
14.3 Squares of Reals . . . 146
14.4 Various Other Theorems . . . 148

15 Vector Spaces and Algebras over the Reals 149
15.1 Locale for additive functions . . . 149
15.2 Real vector spaces . . . 149
15.3 Embedding of the Reals into any *real-algebra-1*: *of-real* . . . 152
15.4 The Set of Real Numbers . . . 154
15.5 Real normed vector spaces . . . 156
15.6 Sign function . . . 160
15.7 Bounded Linear and Bilinear Operators . . . 161

16 Resource-Aware Operational semantics of Safe expressions 164

17 Depth-Aware Operational semantics of Safe expressions 167

18 Definitions for certifying memory bounds 174

19 Auxiliary lemmas of the soundness theorem 186

20 Proof-rules for certifying memory bounds, and soundness theorem 245

1 Normal form values and heap

theory *SafeHeap*
imports *Main*

begin

types
Location = *nat*
Constructor = *string*
FunName = *string*

— Normal form values
datatype *Val* = *Loc Location* | *IntT int* | *BoolT bool*

— Destructions of datatype val
consts *the-IntT* :: *Val* ⇒ *int*
primrec
the-IntT (*IntT i*) = *i*

consts *the-BoolT* :: *Val* ⇒ *bool*
primrec
the-BoolT (*BoolT b*) = *b*

— check if is constant bool
constdefs *isBool* :: *Val* ⇒ *bool*
isBool v ≡ (*case v of* (*BoolT* -) ⇒ *True*
| - ⇒ *False*)

A heap is a partial mapping from locations to cells. But, as it is split into regions, the mapping tells also the region where the cell lives. The second component is the highest live region k. A consistent heap (h, k) has cells only in regions $0 \ldots k$.

types
Cell = *Constructor* × *Val list*
Region = *nat*
HeapMap = *Location* ⇀ (*Region* × *Cell*)
Heap = *HeapMap* × *nat*

consts
restrictToRegion :: *Heap* => *Region* => *Heap* (**infix** ↓ *110*)

primrec
(*h*,*k*) ↓ *k0* = (*let A* = { *p* . *p* ∈ *dom h* & *fst* (*the* (*h p*)) <= *k0*}
in (*h* |‘ *A*,*k0*))

definition
rangeHeap :: *HeapMap* ⇒ *Val set* **where**
rangeHeap h = {*v*. *EX p j C vn*. *h p* = *Some* (*j*,*C*,*vn*) ∧ *v* ∈ *set vn*}

definition
fresh :: *Location* ⇒ *HeapMap* ⇒ *bool* **where**
fresh p h = (*p* ∉ *dom h* ∧ (*Loc p*) ∉ *rangeHeap h*)

definition *domLoc* :: *HeapMap* ⇒ *Val set* **where**
domLoc h = {*l*. *EX p*. *p* ∈ *dom h* ∧ *l* = *Loc p*}

declare *rangeHeap-def* [*simp del*]
declare *fresh-def* [*simp del*]
declare *domLoc-def* [*simp del*]

constdefs
getFresh :: *HeapMap* ⇒ *Location*
getFresh h ≡ *SOME b*. *fresh b h*

constdefs
self :: *string* — this identifies the topmost region referenced in a function body
self ≡ *''self''*

The constructor table tells, for each constructor, the number of arguments and a description of each one. The second nat gives the alternative $0..n-1$ corresponding to this constructor in every **case** of its type

datatype *ArgType* = *IntArg* | *BoolArg* | *NonRecursive* | *Recursive*

types
ConstructorTableType = (*Constructor* × (*nat* × *nat* × *ArgType list*)) *list*
ConstructorTableFun = *Constructor* ⇀ (*nat* × *nat* × *ArgType list*)

This is the constructor table of the Safe expressions semantics. It is assumed to be a constant which somebody else will provide. It is used in the semantic function 'copy'

consts
ConstructorTable :: *ConstructorTableFun*

constdefs *getConstructorCell* :: *Cell* ⇒ *Constructor*
getConstructorCell c ≡ *fst c*

constdefs *getValuesCell* :: *Cell* ⇒ *Val list*
getValuesCell c ≡ *snd c*

constdefs *getCell* :: *Heap* ⇒ *Location* ⇒ *Cell*
getCell h l ≡ *snd* (*the* ((*fst h*) *l*))

constdefs *getRegion* :: *Heap* ⇒ *Location* ⇒ *Region*
getRegion h l ≡ *fst* (*the* ((*fst h*) *l*))

constdefs *domHeap* :: *Heap* ⇒ *Location set*

```
domHeap h ≡ dom (fst h)

constdefs isNonBasicValue :: ArgType ⇒ bool
isNonBasicValue a == (a = NonRecursive) ∨ (a = Recursive)

constdefs isRecursive :: ArgType ⇒ bool
isRecursive a == (a = Recursive)

consts
 theLocation :: Val ⇒ Location
primrec theLocation (Loc l) = l


constdefs
getArgType C  ≡ snd (snd (the (ConstructorTable C)))

constdefs getRecursiveValuesCell :: Cell ⇒ Location set
getRecursiveValuesCell c == set (map (theLocation o snd)
                    (filter (isRecursive o fst) (zip (getArgType (getConstructorCell
c)) (getValuesCell c))))

constdefs recDescendants :: Location ⇒ Heap ⇒ Location set
recDescendants l h ≡ case ((fst h) l) of Some c ⇒ getRecursiveValuesCell (snd c)
                                  | None ⇒ {}

constdefs getNonBasicValuesCell :: Cell ⇒ Location set
getNonBasicValuesCell c == set (map (theLocation o snd)
                                  (filter (isNonBasicValue o fst) (zip (getArgType
(getConstructorCell c)) (getValuesCell c))))


constdefs descendants :: Location ⇒ Heap ⇒ Location set
descendants l h ≡ case ((fst h) l) of Some c ⇒ getNonBasicValuesCell (snd c)
                                  | None ⇒ {}


constdefs isConstant :: Val ⇒ bool
 isConstant v ≡ (case v of (IntT -)  ⇒ True
                      | (BoolT -)  ⇒ True
                      | -          ⇒ False)
end
```

2 Useful functions and theorems from the Haskell Library or Prelude

```
theory  HaskellLib
imports Main
```

begin

Function *mapAccumL* is a powerful combination of *map* and *foldl*. Functions *unzip3* and *unzip* are respectively the inverse of *zip3* and *zip*.

consts
```
  mapAccumL :: ('a => 'b => 'a × 'c) => 'a => 'b list => 'a  × 'c list
  zipWith   :: ('a => 'b => 'c) => 'a list => 'b list => 'c list
  unzip3    :: ('a × 'b× 'c) list => 'a list × 'b list × 'c list
  unzip     :: ('a × 'b) list => 'a list × 'b list
```

primrec
```
  mapAccumL f s []     = (s,[])
  mapAccumL f s (x#xs) = (let (s',y)   = f s x;
                              (s'',ys) = mapAccumL f s' xs
                          in (s'',y#ys))
```

Some lemmas about mapAccumL

lemma *mapAccumL-non-empty*:
```
  ⟦ (s'',ys) = mapAccumL f s xs;
    xs = x#xx
    ⟧ ⟹ (∃ s' y ys'.
            (s',y) = f s x
            ∧ ys = y # ys')
```
apply (*clarify*)
apply (*unfold mapAccumL.simps*)
apply (*rule-tac x=fst (f s x)* **in** *exI*)
apply (*rule-tac x=snd (f s x)* **in** *exI*)
apply (*rule-tac x=snd (mapAccumL f (fst (f s x)) xx)* **in** *exI*)
apply (*rule conjI*)
apply *simp*
apply (*case-tac f s x,simp*)
by (*case-tac mapAccumL f a xx,simp*)

lemma *mapAccumL-non-empty2*:
```
  ⟦ (s'',ys) = mapAccumL f s xs;
    xs = x#xx
    ⟧ ⟹ (∃ s' y ys'.
            (s',y) = f s x
            ∧ (s'',ys') = mapAccumL f s' xx
            ∧ ys = y # ys')
```
apply *clarify*
apply (*unfold mapAccumL.simps*)
apply (*rule-tac x=fst (f s x)* **in** *exI*)
apply (*rule-tac x=snd (f s x)* **in** *exI*)
apply (*rule-tac x=snd (mapAccumL f (fst (f s x)) xx)* **in** *exI*)
apply (*rule conjI*)
apply *simp*
apply (*rule conjI*)
apply (*case-tac f s x*) **apply** (*simp*)

apply (*case-tac mapAccumL f a xx*)
apply (*simp*)
apply (*case-tac f s x*) **apply** (*simp*)
apply (*case-tac mapAccumL f a xx*)
apply *simp*
done

axioms *mapAccumL-non-empty3*:
⟦ (*s''*,*ys*) = *mapAccumL f s xs*;
0<*length xs*
⟧ ⟹ (∃ *s' y ys'*.
(*s'*,*y*) = *f s* (*xs*!*0*)
∧ (*s''*,*ys'*) = *mapAccumL f s'* (*tl xs*))

axioms *mapAccumL-two-elements*:
⟦ (*s3*,*ys*) = *mapAccumL f s xs*;
xs = *x1*#*x2*#*xx*
⟧ ⟹ (∃ *s1 s2 y1 y2 ys3*.
(*s1*,*y1*) = *f s x1*
∧ (*s2*,*y2*) = *f s1 x2*
∧ (*s3*,*ys3*) = *mapAccumL f s2 xx*
∧ *ys* = *y1*#*y2*#*ys3*)

axioms *mapAccumL-split*:
⟦ (*s2*,*ys*) = *mapAccumL f s xs*;
xs1 @ *xs2*=*xs*
⟧ ⟹ (∃ *s1 ys1 ys2* .
(*s1*,*ys1*) = *mapAccumL f s xs1*
∧ (*s2*,*ys2*) = *mapAccumL f s1 xs2*
∧ *ys* = *ys1* @ *ys2*)

axioms *mapAccumL-one-more*:
⟦ (*s1*,*ys*) = *mapAccumL f s xs*;
(*s2*,*y*) = *f s1 x*
⟧ ⟹ (*s2*,*ys*@[*y*]) = *mapAccumL f s* (*xs*@[*x*])

Some integer arithmetic lemmas

lemma *sum-nat*:
⟦ (*x1*::*nat*)=*x2*;(*y1*::*nat*)=*y2*⟧ ⟹ *x1*+*y1*=*x2*+*y2*
apply *arith*
done

axioms *sum-substract*:
(*x*::*nat*)−*y*+(*z*−*x*)=*z*−*y*

axioms *additions1*:
⟦ *i*< *m*; *Suc m* + *n* ≤ *l*⟧ ⟹
m − *i* < *nat* (*int l* − *1*) − *n* + *1* − (*nat* (*int l* − *1*) − *Suc m* − *n* + *1*)
axioms *additions2*:

⟦ $i< m$; $Suc\ m + n \leq l$⟧ ⟹
$nat\ (int\ l - 1) - m + (m - Suc\ i) = nat\ (int\ l - 1) - Suc\ m + (m - i)$
axioms *additions3*:
⟦ $i< m$; $Suc\ m + n \leq l$⟧ ⟹
$nat\ (int\ l - 1) - Suc\ (m + n) + (m - i) = nat\ (int\ l - 1) - (m + n) + (m - Suc\ i)$
axioms *additions4*:
⟦ $Suc\ m + n \leq l$⟧ ⟹
$nat\ (int\ l - 1) - m = Suc\ (nat\ (int\ l - 1) - Suc\ m)$
axioms *additions5*:
⟦ $Suc\ m + n \leq l$⟧ ⟹
$Suc\ (nat\ (int\ l - 1) - Suc\ (m + n)) = nat\ (int\ l - 1 - int\ n - int\ m)$
axioms *additions6*:
⟦ $Suc\ m + n \leq l$⟧ ⟹
$n + (nat\ (int\ l - 1) - Suc\ (m + n)) < nat\ (int\ l - 1)$

Some lemmas about lists

lemma *list-non-empty*:
$0<length\ xs \Longrightarrow (\exists\ y\ ys\ .\ xs = y\ \#\ ys)$
apply *auto*
apply (*insert neq-Nil-conv* [*of xs*])
by *simp*

axioms *drop-nth*:
$n < length\ xs \Longrightarrow (\exists\ y\ ys\ .\ drop\ n\ xs = y\#ys \wedge xs!n = y)$

axioms *drop-nth3*:
$n < length\ xs \Longrightarrow drop\ n\ xs = (xs!n)\ \#\ drop\ (Suc\ n)\ xs$

axioms *drop-take-Suc*:
$xs=(take\ n\ xs)@(z\#zs) \Longrightarrow drop\ (Suc\ n)\ xs=zs$

axioms *drop-nth2*:
⟦ $n<length\ xs$; $drop\ n\ xs = ys$⟧
⟹ $ys = xs!n\ \#\ tl\ ys$

axioms *drop-append2*:
⟦ $drop\ n\ xs = zs1\ @\ ys1\ @\ ys2\ @\ zs2\ @\ rest$;
$drop\ (m-n)\ (zs1\ @\ ys1\ @\ ys2\ @\ zs2) = ys1\ @\ rest'$
⟧ ⟹
$drop\ (m+length\ ys1-n)\ (zs1\ @\ ys1\ @\ ys2\ @\ zs2) = ys2\ @\ zs2$

axioms *drop-append3*:
⟦ $drop\ n\ xs = xs1\ @\ rest$;
$drop\ (m-n)\ xs1 = ys1\ @\ ys2$
⟧ ⟹
$drop\ m\ xs = ys1\ @\ ys2\ @\ rest$

lemma *nth-via-drop-append*: *drop n xs* = (*y*#*ys*)@*zs* ==> *xs*!*n* = *y*
apply (*induct xs arbitrary*: *n*, *simp*)
by(*simp add*:*drop-Cons nth-Cons split*:*nat.splits*)

lemma *drop-Suc-append*:
drop n xs = (*y*#*ys*)@*zs* ⟹ *drop* (*Suc n*) *xs* = *ys*@*zs*
apply (*induct xs arbitrary*: *n*,*simp*)
apply (*simp add*:*drop-Cons*)
by (*simp split*:*nat.splits*)

lemma *nth-via-drop-append-2*: *drop n xs* = ((*y* # *ys*) @ *ws* @ *zs*) @ *ms* ==> *xs*!*n* = *y*
apply (*induct xs arbitrary*: *n*, *simp*)
by(*simp add*:*drop-Cons nth-Cons split*:*nat.splits*)

lemma *drop-Suc-append-2*:
drop n xs = ((*y* # *ys*) @ *ws* @ *zs*) @ *ms* ⟹ *drop* (*Suc n*) *xs* = *ys* @ *ws* @ *zs* @ *ms*
apply (*induct xs arbitrary*: *n*,*simp*)
apply (*simp add*:*drop-Cons*)
by (*simp split*:*nat.splits*)

axioms *drop-append-length*:
drop n xs = [] @ *ys* @ *zs* @ *ms* ⟹ *drop* (*n* + *length ys*) *xs* = *zs* @ *ms*

axioms *take-length*:
n ≤ *length xs* ⟹ *n* = *length* (*take n xs*)

axioms *take-append2*:
n<*length xs* ⟹ *x* # *take n xs* = *take n* (*x* # *xs*) @ [(*x* # *xs*)!*n*]

axioms *take-append3*:
Suc n ≤ *length xs* ⟹ *take* (*Suc n*) *xs* = *take n xs* @ [*xs*!*n*]

axioms *concat1*:
xs @ *y* # *ys* = (*xs* @ [*y*]) @ *ys*

axioms *concat2*:
xs1 = *xs2* ⟹ *xs1* @ *ys* = *xs2* @ *ys*

axioms *upt-length*:
n ≤ *m* ⟹ *length* [*n*..<*m*]=*m*−*n*

Some lemmas about finite maps

axioms *map-of-distinct*:
⟦ *distinct* (*map fst xys*);
l < *length xys*;
(*x*,*y*) = *xys* ! *l*
⟧ ⟹ *map-of xys x* = *Some y*

axioms *map-of-distinct2*:
map-of xys x = Some y
$\Longrightarrow$ ($\exists$ *l . l < length xys* $\wedge$ *(x,y) = xys ! l*)

axioms *map-upds-nth*:
i < m−n $\Longrightarrow$ *(A([n..<m] [↦] xs)) (n+i) = Some (xs ! i)*

— The unzip3 function of Haskell library

primrec
unzip3 [] = *([],[],[])*
unzip3 (tup#tups) = *(let (xs,ys,zs) = unzip3 tups;*
(x,y,z) = tup
in (x#xs,y#ys,z#zs))

axioms *unzip3-length*:
unzip3 xs = (ys1,ys2,ys3) $\Longrightarrow$ *length ys1 = length ys2*

primrec
unzip [] = *([],[])*
unzip (tup#tups) = *(let (xs,ys) = unzip tups;*
(x,y) = tup
in (x#xs,y#ys))

primrec
zipWith f (x#xs) yy = *(case yy of*
[] => []
| y#ys => f x y # zipWith f xs ys)
zipWith f [] yy = *[]*

axioms *zipWith-length*:
length (zipWith f xs ys) = min (length xs) (length ys)

— The Haskell sum type Either

datatype *('a,'b) Either = Left 'a | Right 'b*

— insertion sort for list of strings

constdefs
leString :: string => string => bool
leString s1 s2 == True

consts

```
  ins :: string => string list => string list
primrec
  ins s []       = [s]
  ins s (s'#ss) = (if leString s s' then s#s'#ss
                   else s'# ins s ss)

fun sort :: string list => string list
where
  sort ss = foldr ins ss []


fun subList :: 'a list ⇒ 'a list ⇒ bool
where
  subList xs ys = (∃ hs ts. ys = hs @ xs @ ts)

end
```

3 Normalized Safe Expressions

```
theory SafeExpr imports ../SafeImp/SafeHeap ../SafeImp/HaskellLib
begin
```

This is a somewhat simplified copy of the abstract syntax used by the Safe compiler. The idea is that the Haskell code generated by Isabelle for the definition of the *trProg* function, translating from CoreSafe to SafeImp, can be directly used as a phase of the compiler. The simplifications are in expresion LetE and in the definition of 'a Der, in order to avoid unnecessary mutual recursion between types. First, we define the key elements of CoreSafe abstract syntax.

```
constdefs
  intType :: string
  intType == "Int"
  boolType :: string
  boolType == "Bool"

datatype ExpTipo = VarT string
                 | ConstrT string ExpTipo list bool string list
                 | Rec

datatype AltDato = ConstrA string (ExpTipo list) string

types    DecData  = string × string list  ×  string list  × AltDato list

datatype Lit = LitN int | LitB bool

datatype 'a Patron = ConstP Lit
                   | VarP string 'a
                   | ConstrP string ('a Patron) list 'a
```

Now we define the CoreSafe expressions.

```
types
  ProgVar = string
  RegVar  = string

datatype 'a Exp = ConstE  Lit 'a
                | ConstrE string  ('a Exp) list RegVar 'a
                | VarE    ProgVar 'a
                | CopyE   ProgVar RegVar 'a        ( - @ - - 90)
                | ReuseE  ProgVar 'a
                | AppE    FunName ('a Exp) list RegVar list 'a
                | LetE    string  ('a Exp) ('a Exp) 'a
                                        (Let - = - In - - 95)

                | CaseE   ('a Exp) ('a Patron × 'a Exp) list  'a
                                        (Case - Of - - 95)
                | CaseDE  ('a Exp) ('a Patron × 'a Exp) list 'a
                                        (CaseD - Of - - 95)
```

Now, the rest of the abstract syntax.

```
datatype 'a Der    = Simple ('a Exp) int list

types
  'a Izq   = string × ('a Patron  × bool) list  × string list
  'a Def   = ExpTipo list × 'a Izq × 'a Der
  'a Prog  = DecData list × ('a Def) list ×  'a Exp
```

3.1 Free Variables

```
fun pat2var :: 'a Patron => string
where
  pat2var (VarP x -) = x


fun extractP :: 'a Patron ⇒ (string × string list)
where
  extractP (ConstrP C ps -) = (
             let xs = map pat2var ps
             in (C,xs))
| extractP  - = ([],[])


fun extractVar :: 'a Patron ⇒ string list
where
  extractVar (ConstrP C ps a) = map pat2var ps
| extractVar (ConstP l) = []
| extractVar (VarP v a) = [v]

consts varProgPat   :: 'a Patron ⇒ string set
```

$varProgPats :: 'a\ Patron\ list \Rightarrow string\ set$

primrec
$varProgPat\ (ConstP\ l) = \{\}$
$varProgPat\ (VarP\ x\ a) = \{x\}$
$varProgPat\ (ConstrP\ C\ pats\ a) = varProgPats\ pats$

$varProgPats\ [] = \{\}$
$varProgPats\ (pat\#pats) = varProgPat\ pat \cup varProgPats\ pats$

consts $varProg :: 'a\ Exp \Rightarrow ProgVar\ set$
$varProgs :: 'a\ Exp\ list \Rightarrow ProgVar\ set$
$varProgs' :: 'a\ Exp\ list \Rightarrow ProgVar\ set$
$varProgAlts :: ('a\ Patron \times 'a\ Exp)\ list \Rightarrow string\ set$
$varProgAlts' :: ('a\ Patron \times 'a\ Exp)\ list \Rightarrow string\ set$
$varProgTup :: 'a\ Patron \times 'a\ Exp \Rightarrow string\ set$
$varProgTup' :: 'a\ Patron \times 'a\ Exp \Rightarrow string\ set$

primrec
$varProg\ (ConstE\ \ Lit\ a) = \{\}$
$varProg\ (ConstrE\ C\ exps\ r\ a) = varProgs\ exps$
$varProg\ (VarE\ \ x\ a) = \{x\}$
$varProg\ (CopyE\ \ x\ r\ a) = \{x\}$
$varProg\ (ReuseE\ \ x\ a) = \{x\}$
$varProg\ (AppE\ \ fn\ exps\ rs\ a) = varProgs'\ exps$
$varProg\ (LetE\ \ x1\ \ e1\ e2\ a) = varProg\ e1 \cup varProg\ e2 \cup \{x1\}$
$varProg\ (CaseE\ \ exp\ alts\ a) = varProg\ exp \cup varProgAlts\ alts$
$varProg\ (CaseDE\ \ exp\ alts\ a) = varProg\ exp \cup varProgAlts'\ alts$

$varProgs\ [] = \{\}$
$varProgs\ (exp\#exps) = varProg\ exp \cup varProgs\ exps$

$varProgs'\ [] = \{\}$
$varProgs'\ (exp\#exps) = varProg\ exp \cup varProgs'\ exps$

$varProgAlts\ [] = \{\}$
$varProgAlts\ (alt\#alts) = varProgTup\ alt \cup varProgAlts\ alts$

$varProgAlts'\ [] = \{\}$
$varProgAlts'\ (alt\#alts) = varProgTup'\ alt \cup varProgAlts'\ alts$

$varProgTup\ (pat,e) = varProgPat\ pat \cup varProg\ e$

$varProgTup'\ (pat,e) = varProgPat\ pat \cup varProg\ e$

consts $fv :: 'a\ Exp \Rightarrow string\ set$
$fvs :: 'a\ Exp\ list \Rightarrow string\ set$
$fvs' :: 'a\ Exp\ list \Rightarrow string\ set$

$fvAlts :: ('a\ Patron \times 'a\ Exp)\ list \Rightarrow string\ set$
$fvAlts' :: ('a\ Patron \times 'a\ Exp)\ list \Rightarrow string\ set$
$fvTup :: 'a\ Patron \times 'a\ Exp \Rightarrow string\ set$
$fvTup' :: 'a\ Patron \times 'a\ Exp \Rightarrow string\ set$

primrec
$fv\ (ConstE\ \ Lit\ a) = \{\}$
$fv\ (ConstrE\ C\ exps\ rv\ a) = fvs\ exps$
$fv\ (VarE\ \ \ x\ a) = \{x\}$
$fv\ (CopyE\ \ x\ rv\ a) = \{x\}$
$fv\ (ReuseE\ \ x\ a) = \{x\}$
$fv\ (AppE\ \ \ fn\ exps\ rvs\ a) = fvs'\ exps$
$fv\ (LetE\ \ \ x1\ \ e1\ e2\ a) = fv\ e1 \cup fv\ e2 - \{x1\}$
$fv\ (CaseE\ \ \ exp\ patexps\ a) = fvAlts\ patexps \cup\ fv\ exp$
$fv\ (CaseDE\ \ exp\ patexps\ a) = fvAlts'\ patexps \cup\ fv\ exp$

$fvs\ [] = \{\}$
$fvs\ (exp\#exps) = fv\ exp \cup fvs\ exps$

$fvs'\ [] = \{\}$
$fvs'\ (exp\#exps) = fv\ exp \cup fvs'\ exps$

$fvAlts\ [] = \{\}$
$fvAlts\ (alt\#alts) = fvTup\ alt \cup fvAlts\ alts$

$fvAlts'\ [] = \{\}$
$fvAlts'\ (alt\#alts) = fvTup'\ alt \cup fvAlts'\ alts$

$fvTup\ (pat,e) = fv\ e\ -\ set\ (snd\ (extractP\ pat))$

$fvTup'\ (pat,e) = fv\ e\ -\ set\ (snd\ (extractP\ pat))$

consts $fvReg \quad :: 'a\ Exp \Rightarrow string\ set$
$fvsReg \quad :: 'a\ Exp\ list \Rightarrow string\ set$
$fvsReg' \quad :: 'a\ Exp\ list \Rightarrow string\ set$
$fvAltsReg :: ('a\ Patron \times 'a\ Exp)\ list \Rightarrow string\ set$
$fvAltsReg' :: ('a\ Patron \times 'a\ Exp)\ list \Rightarrow string\ set$
$fvTupReg \quad :: 'a\ Patron \times 'a\ Exp \Rightarrow string\ set$
$fvTupReg' :: 'a\ Patron \times 'a\ Exp \Rightarrow string\ set$

primrec
$fvReg\ (ConstE\ \ Lit\ a) = \{\}$
$fvReg\ (ConstrE\ C\ exps\ r\ a) = \{r\}$
$fvReg\ (VarE\ \ \ x\ a) = \{\}$
$fvReg\ (CopyE\ \ x\ r\ a) = \{r\}$
$fvReg\ (ReuseE\ \ x\ a) = \{\}$
$fvReg\ (AppE\ \ \ fn\ exps\ rvs\ a) = set\ rvs$
$fvReg\ (LetE\ \ \ x1\ \ e1\ e2\ a) = fvReg\ e1 \cup fvReg\ e2$

$fvReg\ (CaseE\quad exp\ patexps\ a) = fvAltsReg\ patexps$
$fvReg\ (CaseDE\ \ exp\ patexps\ a) = fvAltsReg'\ patexps$

$fvAltsReg\ [] = \{\}$
$fvAltsReg\ (alt\#alts) = fvTupReg\ alt \cup fvAltsReg\ alts$

$fvAltsReg'\ [] = \{\}$
$fvAltsReg'\ (alt\#alts) = fvTupReg'\ alt \cup fvAltsReg'\ alts$

$fvTupReg\ (pat,e) = fvReg\ e$

$fvTupReg'\ (pat,e) = fvReg\ e$

consts $boundVar\quad\quad :: 'a\ Exp \Rightarrow string\ set$
$boundVars\quad\quad :: 'a\ Exp\ list \Rightarrow string\ set$
$boundVars'\quad\quad :: 'a\ Exp\ list \Rightarrow string\ set$
$boundVarAlts\ :: ('a\ Patron \times 'a\ Exp)\ list \Rightarrow string\ set$
$boundVarAlts' :: ('a\ Patron \times 'a\ Exp)\ list \Rightarrow string\ set$
$boundVarTup\quad :: 'a\ Patron \times 'a\ Exp \Rightarrow string\ set$
$boundVarTup'\ :: 'a\ Patron \times 'a\ Exp \Rightarrow string\ set$

primrec
$boundVar\ (ConstE\ \ Lit\ a) = \{\}$
$boundVar\ (ConstrE\ C\ exps\ rv\ a) = boundVars\ exps$
$boundVar\ (VarE\quad x\ a) = \{\}$
$boundVar\ (CopyE\quad x\ rv\ a) = \{\}$
$boundVar\ (ReuseE\ \ x\ a) = \{\}$
$boundVar\ (AppE\quad fn\ exps\ rvs\ a) = \{\}$
$boundVar\ (LetE\quad x1\ \ e1\ e2\ a) = \{x1\}$
$boundVar\ (CaseE\quad exp\ patexps\ a) = boundVarAlts\ patexps$
$boundVar\ (CaseDE\ \ exp\ patexps\ a) = boundVarAlts'\ patexps$

$boundVars\ [] = \{\}$
$boundVars\ (exp\#exps) = boundVar\ exp \cup boundVars\ exps$

$boundVarAlts\ [] = \{\}$
$boundVarAlts\ (alt\#alts) = boundVarTup\ alt \cup boundVarAlts\ alts$

$boundVarAlts'\ [] = \{\}$
$boundVarAlts'\ (alt\#alts) = boundVarTup'\ alt \cup boundVarAlts'\ alts$

$boundVarTup\ (pat,e) = set\ (extractVar\ pat)$

$boundVarTup'\ (pat,e) = set\ (extractVar\ pat)$

A runtime environment consists of two partial mappings: one from program variables to normal form values, and one from region variables to actual

regions.

```
types
  Environment = (ProgVar ⇀ Val) × (RegVar ⇀ Region)
```

The runtime system provides a 'copy' function which generates a new data structure from a given location by copying those cells pointed to by recursive argument positions of data constructors. The Σ environment provides the textual definitions of previously defined Safe functions. Some auxiliary functions: 'extend' extends an environment with a collection of bindings from variables to values; 'fresh' is a predicate telling whether a variable is fresh with respect to a heap; 'atom2val', given an environment and an atom (a program variable or a literal expression) returns its corresponding value; 'atom2var', given and expression return its corresponding variable; 'atom', is a predicate telling whether a expression is a atom.

```
constdefs recursiveArgs :: Constructor ⇒ bool list
 recursiveArgs C ≡  (let
                      (-,-,args) = the (ConstructorTable C)
                     in map (%a. a = Recursive) args)

function copy' :: [Region,HeapMap,(Val × bool)] ⇒ (HeapMap × Val)
where

   copy' j h (v,False)        = (h,v)
|  copy' j h (Val.Loc p,True) = (let
                                  (k,C,ps) = the (h p);
                                  bs       = recursiveArgs C;
                                  pbs      = zip ps bs;
                                  (h',ps') = mapAccumL (copy' j) h pbs;
                                  p'       = getFresh h'
                                  in (h'(p'↦(j,C,ps')),Val.Loc p'))
|  copy' j h (IntT i,True)     = (h,IntT i)
|  copy' j h (BoolT b,True)    = (h,BoolT b)
by  pat-completeness auto

function copy :: [Heap, Location, Region] ⇒ (Heap × Location)
where
 copy (h,k) p j = (let
                     (h',p') = copy' j h (Val.Loc p,True)
                    in case p' of (Val.Loc q) ⇒ ((h',k), q))
by pat-completeness auto
termination by (relation {}) simp


types FunDefEnv = string ⇀ ProgVar list × RegVar list × unit Exp

consts
  Σf :: FunDefEnv
```

constdefs *bodyAPP* :: *FunDefEnv* ⇒ *string* ⇒ *unit Exp*
bodyAPP Σ *f* == (*case* Σ *f of Some* (*xs*,*rs*,*ef*) ⇒ *ef*)

constdefs *varsAPP* :: *FunDefEnv* ⇒ *string* ⇒ *string list*
varsAPP Σ *f* ≡ (*case* Σ *f of Some* (*xs*,*rs*,*ef*) ⇒ *xs*)

constdefs *regionsAPP* :: *FunDefEnv* ⇒ *string* ⇒ *string list*
regionsAPP Σ *f* ≡ (*case* Σ *f of Some* (*xs*,*rs*,*ef*) ⇒ *rs*)

definition
extend :: [*string* ⇀ *Val*, *ProgVar list*, *Val list*] ⇒(*ProgVar* ⇀ *Val*) **where**
extend E xs vs = *E* ++ *map-of* (*zip xs vs*)

definition
def-extend :: [*string* ⇀ *Val*, *ProgVar list*, *Val list*] ⇒ *bool* **where**
def-extend E xs vs = (*set xs* ∩ *dom E* = {} ∧ *length xs* = *length vs* ∧ *distinct xs* ∧ (∀ *x*∈ *set xs*. *x* ≠ *self*))

fun *atom2val* :: (*ProgVar* ⇀ *Val*) ⇒ *'a Exp* ⇒ *Val*
where
atom2val E (*ConstE* (*LitN i*) *a*) = *IntT i*
| *atom2val E* (*ConstE* (*LitB b*) *a*) = *BoolT b*
| *atom2val E* (*VarE x a*) = *the* (*E x*)

fun *atom2var* :: *'a Exp* ⇒ *string*
where
atom2var (*VarE x a*) = *x*

fun *atom* :: *'a Exp* ⇒ *bool*
where
atom e = (*case e of*
(*VarE x a*) ⇒ *True*
| - ⇒ *False*)

Lemmas for extend function

lemma *extend-monotone*: *x* ∉ *set xs* ⟹ *E x* = *extend E xs vs x*
apply (*induct xs vs rule*:*list-induct2'*)
apply (*simp add*: *extend-def*)
apply (*simp add*: *extend-def*)
apply (*simp add*: *extend-def*)
apply (*subgoal-tac x* ∉ *set xs*,*simp*)
apply (*subgoal-tac extend E* (*xa* # *xs*) (*y* # *ys*) = (*extend E xs ys*)(*xa* ↦*y*))
apply *simp*
apply (*simp add*: *extend-def*)

by *simp*

lemma *list-induct3*:
[| *P* [] *0*;
!!*x xs*. *P* (*x*#*xs*) *0*;
!!*i*. *P* [] (*Suc i*);
!!*x xs i*. *P xs i* ==> *P* (*x*#*xs*) (*Suc i*) |]
==> *P xs i*
by (*induct xs arbitrary*: *i*) (*case-tac x*, *auto*)+

lemma *extend-monotone-i* [*rule-format*]:
i < *length alts* ⟶
length alts > *0* ⟶
x ∉ *set* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) ⟶
E x = *extend E* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs x*
apply (*induct alts i rule*: *list-induct3*, *simp-all*)
apply (*rule impI*)
apply (*erule extend-monotone*)
apply (*rule impI*, *simp*)
by (*case-tac xs*=[],*simp-all*)

lemma *extend-prop1*:
⟦ *z* ∈ *dom* (*extend E xs vs*); *z* ∉ *set xs*; *length xs* = *length vs* ⟧ ⟹ *z* ∈ *dom E*
apply (*simp add*: *extend-def*)
apply (*erule disjE*)
apply (*simp add*: *dom-def*)
by *simp*

end

4 Primitive operators for SVM anf JVM

theory *BinOP*
imports *Main*
begin

Primitive operators

datatype *PrimOp* = *Add* | *Substract* | *Times* | *Divide* | *LessThan* | *LessEqual*
| *Equal* | *GreaterThan* | *GreaterEqual* | *NotEqual*

end

5 State of the SVM

theory *SVMState*

imports *SafeHeap ../JVMSAFE/BinOP*
begin

5.1 Sizes Table

This gives statically inferred information about the maximum number of heap cells, of heap regions, and of stack words needed by the compiled program.

types *ncell* = *nat*
sizeRegions = *nat*
sizeStackS = *nat*

types *SizesTable* = *ncell* × *sizeRegions* × *sizeStackS*

5.2 Stack

types
CodeLabel = *nat*
Continuation = *Region* × *CodeLabel*

datatype *StackObject* = *Val Val* | *Reg Region* | *Cont Continuation*

The SVM stack may contain normal form values, region arguments for functions or constructors, and continuations. A continuation (k_0, p) contains a jump p to a code sequence and an adjustment k_0 for the heap watermark k_0 of the SVM state.

types
Stack = *StackObject list*
StackOffset = *nat*

5.3 Code Store and SafeImp program

— Items are the components of environments and closures
datatype *Item* = *ItemConst Val*
| *ItemVar StackOffset*
| *ItemRegSelf*

— The SVM instruction repertory
datatype *SafeInstr* = *DECREGION*
| *POPCONT*
| *PUSHCONT CodeLabel*
| *COPY*
| *REUSE*
| *CALL CodeLabel*
| *PRIMOP PrimOp*
| *MATCH StackOffset* (*CodeLabel list*)

```
  | MATCHD StackOffset (CodeLabel list)
  | MATCHN StackOffset nat nat (CodeLabel list)
  | BUILDENV (Item list)
  | BUILDCLS Constructor (Item list)  Item
  | SLIDE nat nat
```

```
fun pushcont :: SafeInstr => bool
where
  pushcont (PUSHCONT p) = True
| pushcont -            = False

fun popcont :: SafeInstr => bool
where
  popcont POPCONT = True
| popcont -       = False
```

A Safe program, when translated into SafeImp, produces four components (1) a map from labels to pairs consisting of a code sequence and a function name. It is given as a list in order to be able to 'traverse' the map; (2) a map from function names to pairs consisting of a label and a list of labels. The firts points to the starting sequence of the function and the second collects, for each function body, the code labels corresponding to continuations. The map is also given as a list; (3) the code label of the main expression; and (4) a constructor table collecting the properties of all the constructors.

```
types
  CodeSequence = SafeInstr list
  SVMCode    = (CodeLabel × CodeSequence × FunName) list
  ContinuationMap = (FunName × CodeLabel × CodeLabel list) list
  CodeStore = SVMCode  × ContinuationMap
  SafeImpProg = CodeStore × CodeLabel × ConstructorTableType × SizesTable
```

5.4 Runtime State

```
types
  PC = CodeLabel × nat
  SVMState = Heap × Region × PC × Stack


consts
  incrPC :: PC => PC

primrec
  incrPC (l,i) = (l,i+1)
```

This is the correspondence betweeen primitive operators in CoreSafe and SafeImp.

constdefs

```
primops :: string ⇀ PrimOp

primops ≡ map-of [("+",Add),
                  ("−",Substract),
                  ("∗",Times),
                  ("%",Divide),
                  ("<",LessThan),
                  ("<=",LessEqual),
                  ("==",Equal),
                  (">",GreaterThan),
                  (">=",GreaterEqual)
                 ]
```

— Define primitive operations
consts
execOp :: [*PrimOp*, *Val*, *Val*] => *Val*

primrec
execOp Equal b1 b2 = *BoolT* (*the-IntT*(*b1*) = *the-IntT*(*b2*))
execOp NotEqual b1 b2 = *BoolT* (*the-IntT*(*b1*) ≠ *the-IntT*(*b2*))
execOp GreaterEqual b1 b2 = *BoolT* (*the-IntT*(*b1*) ≥ *the-IntT*(*b2*))
execOp GreaterThan b1 b2 = *BoolT* (*the-IntT*(*b1*) > *the-IntT*(*b2*))
execOp LessThan b1 b2 = *BoolT* (*the-IntT*(*b1*) < *the-IntT*(*b2*))
execOp LessEqual b1 b2 = *BoolT* (*the-IntT*(*b1*) ≤ *the-IntT*(*b2*))

execOp Add b1 b2 = *IntT* (*the-IntT*(*b1*) + *the-IntT*(*b2*))
execOp Substract b1 b2 = *IntT* (*the-IntT*(*b1*) − *the-IntT*(*b2*))
execOp Times b1 b2 = *IntT* (*the-IntT*(*b1*) ∗ *the-IntT*(*b2*))
execOp Divide b1 b2 = *IntT* (*the-IntT*(*b1*) *div the-IntT*(*b2*))

end

6 Definitions of Upper Bounds and Least Upper Bounds

theory *Lubs*
imports *Main*
begin

Thanks to suggestions by James Margetson

definition
setle :: ['*a set*, '*a*::*ord*] => *bool* (**infixl** ∗<= *70*) **where**
S ∗<= *x* = (*ALL y*: *S*. *y* <= *x*)

definition
setge :: ['*a*::*ord*, '*a set*] => *bool* (**infixl** <=∗ *70*) **where**

$x <=* S = (ALL\ y{:}\ S.\ x <= y)$

definition
$leastP :: ['a => bool, 'a::ord] => bool$ **where**
$leastP\ P\ x = (P\ x\ \&\ x <=* Collect\ P)$

definition
$isUb :: ['a\ set,\ 'a\ set,\ 'a::ord] => bool$ **where**
$isUb\ R\ S\ x = (S\ *<=\ x\ \&\ x{:}\ R)$

definition
$isLub :: ['a\ set,\ 'a\ set,\ 'a::ord] => bool$ **where**
$isLub\ R\ S\ x = leastP\ (isUb\ R\ S)\ x$

definition
$ubs :: ['a\ set,\ 'a::ord\ set] => 'a\ set$ **where**
$ubs\ R\ S = Collect\ (isUb\ R\ S)$

6.1 Rules for the Relations $*<=$ and $<=*$

lemma *setleI*: $ALL\ y{:}\ S.\ y <= x ==> S\ *<=\ x$
by (*simp add*: *setle-def*)

lemma *setleD*: $[|\ S\ *<=\ x;\ y{:}\ S\ |] ==> y <= x$
by (*simp add*: *setle-def*)

lemma *setgeI*: $ALL\ y{:}\ S.\ x<=\ y ==> x <=*\ S$
by (*simp add*: *setge-def*)

lemma *setgeD*: $[|\ x <=*\ S;\ y{:}\ S\ |] ==> x <= y$
by (*simp add*: *setge-def*)

6.2 Rules about the Operators *leastP*, *ub* and *lub*

lemma *leastPD1*: $leastP\ P\ x ==> P\ x$
by (*simp add*: *leastP-def*)

lemma *leastPD2*: $leastP\ P\ x ==> x <=*\ Collect\ P$
by (*simp add*: *leastP-def*)

lemma *leastPD3*: $[|\ leastP\ P\ x;\ y{:}\ Collect\ P\ |] ==> x <= y$
by (*blast dest*!: *leastPD2 setgeD*)

lemma *isLubD1*: $isLub\ R\ S\ x ==> S\ *<=\ x$
by (*simp add*: *isLub-def isUb-def leastP-def*)

lemma *isLubD1a*: $isLub\ R\ S\ x ==> x{:}\ R$
by (*simp add*: *isLub-def isUb-def leastP-def*)

lemma *isLub-isUb*: $isLub\ R\ S\ x ==> isUb\ R\ S\ x$

apply (*simp add*: *isUb-def*)
apply (*blast dest*: *isLubD1 isLubD1a*)
done

lemma *isLubD2*: [| *isLub R S x*; *y* : *S* |] ==> *y* <= *x*
by (*blast dest*!: *isLubD1 setleD*)

lemma *isLubD3*: *isLub R S x* ==> *leastP*(*isUb R S*) *x*
by (*simp add*: *isLub-def*)

lemma *isLubI1*: *leastP*(*isUb R S*) *x* ==> *isLub R S x*
by (*simp add*: *isLub-def*)

lemma *isLubI2*: [| *isUb R S x*; *x* <=* *Collect* (*isUb R S*) |] ==> *isLub R S x*
by (*simp add*: *isLub-def leastP-def*)

lemma *isUbD*: [| *isUb R S x*; *y* : *S* |] ==> *y* <= *x*
by (*simp add*: *isUb-def setle-def*)

lemma *isUbD2*: *isUb R S x* ==> *S* *<= *x*
by (*simp add*: *isUb-def*)

lemma *isUbD2a*: *isUb R S x* ==> *x*: *R*
by (*simp add*: *isUb-def*)

lemma *isUbI*: [| *S* *<= *x*; *x*: *R* |] ==> *isUb R S x*
by (*simp add*: *isUb-def*)

lemma *isLub-le-isUb*: [| *isLub R S x*; *isUb R S y* |] ==> *x* <= *y*
apply (*simp add*: *isLub-def*)
apply (*blast intro*!: *leastPD3*)
done

lemma *isLub-ubs*: *isLub R S x* ==> *x* <=* *ubs R S*
apply (*simp add*: *ubs-def isLub-def*)
apply (*erule leastPD2*)
done

end

7 The Greatest Common Divisor

theory *GCD*
imports *Main*
begin

See [**?**].

7.1 Specification of GCD on nats

definition
is-gcd :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where** — *gcd* as a relation
is-gcd p m n ⟷ *p dvd m* ∧ *p dvd n* ∧
(∀ *d*. *d dvd m* ⟶ *d dvd n* ⟶ *d dvd p*)

Uniqueness

lemma *is-gcd-unique*: *is-gcd m a b* ⟹ *is-gcd n a b* ⟹ *m* = *n*
by (*simp add*: *is-gcd-def*) (*blast intro*: *dvd-anti-sym*)

Connection to divides relation

lemma *is-gcd-dvd*: *is-gcd m a b* ⟹ *k dvd a* ⟹ *k dvd b* ⟹ *k dvd m*
by (*auto simp add*: *is-gcd-def*)

Commutativity

lemma *is-gcd-commute*: *is-gcd k m n* = *is-gcd k n m*
by (*auto simp add*: *is-gcd-def*)

7.2 GCD on nat by Euclid's algorithm

fun
gcd :: *nat* × *nat* => *nat*
where
gcd (*m*, *n*) = (*if n* = *0 then m else gcd* (*n*, *m mod n*))

lemma *gcd-induct*:
fixes *m n* :: *nat*
assumes ⋀*m*. *P m 0*
and ⋀*m n*. *0* < *n* ⟹ *P n* (*m mod n*) ⟹ *P m n*
shows *P m n*
apply (*rule gcd.induct* [*of split P* (*m*, *n*), *unfolded Product-Type.split*])
apply (*case-tac n* = *0*)
apply *simp-all*
using *assms* **apply** *simp-all*
done

lemma *gcd-0* [*simp*]: *gcd* (*m*, *0*) = *m*
by *simp*

lemma *gcd-0-left* [*simp*]: *gcd* (*0*, *m*) = *m*
by *simp*

lemma *gcd-non-0*: *n* > *0* ⟹ *gcd* (*m*, *n*) = *gcd* (*n*, *m mod n*)
by *simp*

lemma *gcd-1* [*simp*]: *gcd* (*m*, *Suc 0*) = *1*
by *simp*

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

gcd (*m*, *n*) divides *m* and *n*. The conjunctions don't seem provable separately.

lemma *gcd-dvd1* [*iff*]: *gcd* (*m*, *n*) *dvd m*
and *gcd-dvd2* [*iff*]: *gcd* (*m*, *n*) *dvd n*
apply (*induct m n rule*: *gcd-induct*)
apply (*simp-all add*: *gcd-non-0*)
apply (*blast dest*: *dvd-mod-imp-dvd*)
done

Maximality: for all *m*, *n*, *k* naturals, if *k* divides *m* and *k* divides *n* then *k* divides *gcd* (*m*, *n*).

lemma *gcd-greatest*: *k dvd m* $\Longrightarrow$ *k dvd n* $\Longrightarrow$ *k dvd gcd* (*m*, *n*)
by (*induct m n rule*: *gcd-induct*) (*simp-all add*: *gcd-non-0 dvd-mod*)

Function gcd yields the Greatest Common Divisor.

lemma *is-gcd*: *is-gcd* (*gcd* (*m*, *n*)) *m n*
by (*simp add*: *is-gcd-def gcd-greatest*)

7.3 Derived laws for GCD

lemma *gcd-greatest-iff* [*iff*]: *k dvd gcd* (*m*, *n*) $\longleftrightarrow$ *k dvd m* $\wedge$ *k dvd n*
by (*blast intro*!: *gcd-greatest intro*: *dvd-trans*)

lemma *gcd-zero*: *gcd* (*m*, *n*) = *0* $\longleftrightarrow$ *m* = *0* $\wedge$ *n* = *0*
by (*simp only*: *dvd-0-left-iff* [*symmetric*] *gcd-greatest-iff*)

lemma *gcd-commute*: *gcd* (*m*, *n*) = *gcd* (*n*, *m*)
apply (*rule is-gcd-unique*)
apply (*rule is-gcd*)
apply (*subst is-gcd-commute*)
apply (*simp add*: *is-gcd*)
done

lemma *gcd-assoc*: *gcd* (*gcd* (*k*, *m*), *n*) = *gcd* (*k*, *gcd* (*m*, *n*))
apply (*rule is-gcd-unique*)
apply (*rule is-gcd*)
apply (*simp add*: *is-gcd-def*)
apply (*blast intro*: *dvd-trans*)
done

lemma *gcd-1-left* [*simp*]: *gcd* (*Suc 0*, *m*) = *1*
by (*simp add*: *gcd-commute*)

Multiplication laws

lemma *gcd-mult-distrib2*: *k* $*$ *gcd* (*m*, *n*) = *gcd* (*k* $*$ *m*, *k* $*$ *n*)

— [**?**, page 27]
apply (*induct m n rule*: *gcd-induct*)
apply *simp*
apply (*case-tac k = 0*)
apply (*simp-all add*: *mod-geq gcd-non-0 mod-mult-distrib2*)
done

lemma *gcd-mult* [*simp*]: *gcd* (*k*, *k* ∗ *n*) = *k*
apply (*rule gcd-mult-distrib2* [*of k 1 n*, *simplified*, *symmetric*])
done

lemma *gcd-self* [*simp*]: *gcd* (*k*, *k*) = *k*
apply (*rule gcd-mult* [*of k 1*, *simplified*])
done

lemma *relprime-dvd-mult*: *gcd* (*k*, *n*) = *1* ==> *k dvd m* ∗ *n* ==> *k dvd m*
apply (*insert gcd-mult-distrib2* [*of m k n*])
apply *simp*
apply (*erule-tac t = m* **in** *ssubst*)
apply *simp*
done

lemma *relprime-dvd-mult-iff*: *gcd* (*k*, *n*) = *1* ==> (*k dvd m* ∗ *n*) = (*k dvd m*)
apply (*blast intro*: *relprime-dvd-mult dvd-trans*)
done

lemma *gcd-mult-cancel*: *gcd* (*k*, *n*) = *1* ==> *gcd* (*k* ∗ *m*, *n*) = *gcd* (*m*, *n*)
apply (*rule dvd-anti-sym*)
apply (*rule gcd-greatest*)
apply (*rule-tac n = k* **in** *relprime-dvd-mult*)
apply (*simp add*: *gcd-assoc*)
apply (*simp add*: *gcd-commute*)
apply (*simp-all add*: *mult-commute*)
apply (*blast intro*: *dvd-trans*)
done

Addition laws

lemma *gcd-add1* [*simp*]: *gcd* (*m* + *n*, *n*) = *gcd* (*m*, *n*)
apply (*case-tac n = 0*)
apply (*simp-all add*: *gcd-non-0*)
done

lemma *gcd-add2* [*simp*]: *gcd* (*m*, *m* + *n*) = *gcd* (*m*, *n*)
proof −
have *gcd* (*m*, *m* + *n*) = *gcd* (*m* + *n*, *m*) **by** (*rule gcd-commute*)
also have ... = *gcd* (*n* + *m*, *m*) **by** (*simp add*: *add-commute*)
also have ... = *gcd* (*n*, *m*) **by** *simp*
also have ... = *gcd* (*m*, *n*) **by** (*rule gcd-commute*)
finally show *?thesis* **.**

qed

lemma *gcd-add2′* [*simp*]: *gcd* (*m*, *n* + *m*) = *gcd* (*m*, *n*)
apply (*subst add-commute*)
apply (*rule gcd-add2*)
done

lemma *gcd-add-mult*: *gcd* (*m*, *k* ∗ *m* + *n*) = *gcd* (*m*, *n*)
by (*induct k*) (*simp-all add*: *add-assoc*)

lemma *gcd-dvd-prod*: *gcd* (*m*, *n*) *dvd m* ∗ *n*
using *mult-dvd-mono* [*of 1*] **by** *auto*

Division by gcd yields rrelatively primes.

lemma *div-gcd-relprime*:
assumes *nz*: *a* ≠ *0* ∨ *b* ≠ *0*
shows *gcd* (*a div gcd*(*a*,*b*), *b div gcd*(*a*,*b*)) = *1*
proof −
let *?g* = *gcd* (*a*, *b*)
let *?a′* = *a div ?g*
let *?b′* = *b div ?g*
let *?g′* = *gcd* (*?a′*, *?b′*)
have *dvdg*: *?g dvd a ?g dvd b* **by** *simp-all*
have *dvdg′*: *?g′ dvd ?a′ ?g′ dvd ?b′* **by** *simp-all*
from *dvdg dvdg′* **obtain** *ka kb ka′ kb′* **where**
kab: *a* = *?g* ∗ *ka b* = *?g* ∗ *kb ?a′* = *?g′* ∗ *ka′ ?b′* = *?g′* ∗ *kb′*
unfolding *dvd-def* **by** *blast*
then have *?g* ∗ *?a′* = (*?g* ∗ *?g′*) ∗ *ka′ ?g* ∗ *?b′* = (*?g* ∗ *?g′*) ∗ *kb′* **by** *simp-all*
then have *dvdgg′*:*?g* ∗ *?g′ dvd a ?g*∗ *?g′ dvd b*
by (*auto simp add*: *dvd-mult-div-cancel* [*OF dvdg*(*1*)]
dvd-mult-div-cancel [*OF dvdg*(*2*)] *dvd-def*)
have *?g* ≠ *0* **using** *nz* **by** (*simp add*: *gcd-zero*)
then have *gp*: *?g* > *0* **by** *simp*
from *gcd-greatest* [*OF dvdgg′*] **have** *?g* ∗ *?g′ dvd ?g* **.**
with *dvd-mult-cancel1* [*OF gp*] **show** *?g′* = *1* **by** *simp*
qed

7.4 LCM defined by GCD

definition
lcm :: *nat* × *nat* ⇒ *nat*
where
lcm = (λ(*m*, *n*). *m* ∗ *n div gcd* (*m*, *n*))

lemma *lcm-def*:
lcm (*m*, *n*) = *m* ∗ *n div gcd* (*m*, *n*)
unfolding *lcm-def* **by** *simp*

lemma *prod-gcd-lcm*:

```
  m * n = gcd (m, n) * lcm (m, n)
  unfolding lcm-def by (simp add: dvd-mult-div-cancel [OF gcd-dvd-prod])

lemma lcm-0 [simp]: lcm (m, 0) = 0
  unfolding lcm-def by simp

lemma lcm-1 [simp]: lcm (m, 1) = m
  unfolding lcm-def by simp

lemma lcm-0-left [simp]: lcm (0, n) = 0
  unfolding lcm-def by simp

lemma lcm-1-left [simp]: lcm (1, m) = m
  unfolding lcm-def by simp

lemma dvd-pos:
  fixes n m :: nat
  assumes n > 0 and m dvd n
  shows m > 0
using assms by (cases m) auto

lemma lcm-least:
  assumes m dvd k and n dvd k
  shows lcm (m, n) dvd k
proof (cases k)
  case 0 then show ?thesis by auto
next
  case (Suc -) then have pos-k: k > 0 by auto
  from assms dvd-pos [OF this] have pos-mn: m > 0 n > 0 by auto
  with gcd-zero [of m n] have pos-gcd: gcd (m, n) > 0 by simp
  from assms obtain p where k-m: k = m * p using dvd-def by blast
  from assms obtain q where k-n: k = n * q using dvd-def by blast
  from pos-k k-m have pos-p: p > 0 by auto
  from pos-k k-n have pos-q: q > 0 by auto
  have k * k * gcd (q, p) = k * gcd (k * q, k * p)
    by (simp add: mult-ac gcd-mult-distrib2)
  also have ... = k * gcd (m * p * q, n * q * p)
    by (simp add: k-m [symmetric] k-n [symmetric])
  also have ... = k * p * q * gcd (m, n)
    by (simp add: mult-ac gcd-mult-distrib2)
  finally have (m * p) * (n * q) * gcd (q, p) = k * p * q * gcd (m, n)
    by (simp only: k-m [symmetric] k-n [symmetric])
  then have p * q * m * n * gcd (q, p) = p * q * k * gcd (m, n)
    by (simp add: mult-ac)
  with pos-p pos-q have m * n * gcd (q, p) = k * gcd (m, n)
    by simp
  with prod-gcd-lcm [of m n]
  have lcm (m, n) * gcd (q, p) * gcd (m, n) = k * gcd (m, n)
    by (simp add: mult-ac)
```

```
  with pos-gcd have lcm (m, n) * gcd (q, p) = k by simp
  then show ?thesis using dvd-def by auto
qed

lemma lcm-dvd1 [iff]:
  m dvd lcm (m, n)
proof (cases m)
  case 0 then show ?thesis by simp
next
  case (Suc -)
  then have mpos: m > 0 by simp
  show ?thesis
  proof (cases n)
    case 0 then show ?thesis by simp
  next
    case (Suc -)
    then have npos: n > 0 by simp
    have gcd (m, n) dvd n by simp
    then obtain k where n = gcd (m, n) * k using dvd-def by auto
    then have m * n div gcd (m, n) = m * (gcd (m, n) * k) div gcd (m, n) by
(simp add: mult-ac)
    also have ... = m * k using mpos npos gcd-zero by simp
    finally show ?thesis by (simp add: lcm-def)
  qed
qed

lemma lcm-dvd2 [iff]:
  n dvd lcm (m, n)
proof (cases n)
  case 0 then show ?thesis by simp
next
  case (Suc -)
  then have npos: n > 0 by simp
  show ?thesis
  proof (cases m)
    case 0 then show ?thesis by simp
  next
    case (Suc -)
    then have mpos: m > 0 by simp
    have gcd (m, n) dvd m by simp
    then obtain k where m = gcd (m, n) * k using dvd-def by auto
    then have m * n div gcd (m, n) = (gcd (m, n) * k) * n div gcd (m, n) by
(simp add: mult-ac)
    also have ... = n * k using mpos npos gcd-zero by simp
    finally show ?thesis by (simp add: lcm-def)
  qed
qed
```

7.5 GCD and LCM on integers

definition
igcd :: *int* ⇒ *int* ⇒ *int* **where**
igcd i j = *int* (*gcd* (*nat* (*abs i*), *nat* (*abs j*)))

lemma *igcd-dvd1* [*simp*]: *igcd i j dvd i*
by (*simp add*: *igcd-def int-dvd-iff*)

lemma *igcd-dvd2* [*simp*]: *igcd i j dvd j*
by (*simp add*: *igcd-def int-dvd-iff*)

lemma *igcd-pos*: *igcd i j* ≥ *0*
by (*simp add*: *igcd-def*)

lemma *igcd0* [*simp*]: (*igcd i j* = *0*) = (*i* = *0* ∧ *j* = *0*)
by (*simp add*: *igcd-def gcd-zero*) *arith*

lemma *igcd-commute*: *igcd i j* = *igcd j i*
unfolding *igcd-def* **by** (*simp add*: *gcd-commute*)

lemma *igcd-neg1* [*simp*]: *igcd* (− *i*) *j* = *igcd i j*
unfolding *igcd-def* **by** *simp*

lemma *igcd-neg2* [*simp*]: *igcd i* (− *j*) = *igcd i j*
unfolding *igcd-def* **by** *simp*

lemma *zrelprime-dvd-mult*: *igcd i j* = *1* ⟹ *i dvd k* ∗ *j* ⟹ *i dvd k*
unfolding *igcd-def*
proof −
assume *int* (*gcd* (*nat* |*i*|, *nat* |*j*|)) = *1 i dvd k* ∗ *j*
then have *g*: *gcd* (*nat* |*i*|, *nat* |*j*|) = *1* **by** *simp*
from ‹*i dvd k* ∗ *j*› **obtain** *h* **where** *h*: *k*∗*j* = *i*∗*h* **unfolding** *dvd-def* **by** *blast*
have *th*: *nat* |*i*| *dvd nat* |*k*| ∗ *nat* |*j*|
unfolding *dvd-def*
by (*rule-tac x*= *nat* |*h*| **in** *exI*, *simp add*: *h nat-abs-mult-distrib* [*symmetric*])
from *relprime-dvd-mult* [*OF g th*] **obtain** *h*′ **where** *h*′: *nat* |*k*| = *nat* |*i*| ∗ *h*′
unfolding *dvd-def* **by** *blast*
from *h*′ **have** *int* (*nat* |*k*|) = *int* (*nat* |*i*| ∗ *h*′) **by** *simp*
then have |*k*| = |*i*| ∗ *int h*′ **by** (*simp add*: *int-mult*)
then show *?thesis*
apply (*subst zdvd-abs1* [*symmetric*])
apply (*subst zdvd-abs2* [*symmetric*])
apply (*unfold dvd-def*)
apply (*rule-tac x* = *int h*′ **in** *exI*, *simp*)
done
qed

lemma *int-nat-abs*: *int* (*nat* (*abs x*)) = *abs x* **by** *arith*

lemma *igcd-greatest*:
assumes *k dvd m* **and** *k dvd n*
shows *k dvd igcd m n*
proof −
let *?k′* = *nat* |*k*|
let *?m′* = *nat* |*m*|
let *?n′* = *nat* |*n*|
from ‹*k dvd m*› **and** ‹*k dvd n*› **have** *dvd′*: *?k′ dvd ?m′ ?k′ dvd ?n′*
unfolding *zdvd-int* **by** (*simp-all only*: *int-nat-abs zdvd-abs1 zdvd-abs2*)
from *gcd-greatest* [*OF dvd′*] **have** *int* (*nat* |*k*|) *dvd igcd m n*
unfolding *igcd-def* **by** (*simp only*: *zdvd-int*)
then have |*k*| *dvd igcd m n* **by** (*simp only*: *int-nat-abs*)
then show *k dvd igcd m n* **by** (*simp add*: *zdvd-abs1*)
qed

lemma *div-igcd-relprime*:
assumes *nz*: *a* ≠ *0* ∨ *b* ≠ *0*
shows *igcd* (*a div* (*igcd a b*)) (*b div* (*igcd a b*)) = *1*
proof −
from *nz* **have** *nz′*: *nat* |*a*| ≠ *0* ∨ *nat* |*b*| ≠ *0* **by** *arith*
let *?g* = *igcd a b*
let *?a′* = *a div ?g*
let *?b′* = *b div ?g*
let *?g′* = *igcd ?a′ ?b′*
have *dvdg*: *?g dvd a ?g dvd b* **by** (*simp-all add*: *igcd-dvd1 igcd-dvd2*)
have *dvdg′*: *?g′ dvd ?a′ ?g′ dvd ?b′* **by** (*simp-all add*: *igcd-dvd1 igcd-dvd2*)
from *dvdg dvdg′* **obtain** *ka kb ka′ kb′* **where**
kab: *a* = *?g*ka b* = *?g*kb ?a′* = *?g′*ka′ ?b′* = *?g′* ∗ *kb′*
unfolding *dvd-def* **by** *blast*
then have *?g*∗ ?a′* = (*?g* ∗ *?g′*) ∗ *ka′ ?g*∗ ?b′* = (*?g* ∗ *?g′*) ∗ *kb′* **by** *simp-all*
then have *dvdgg′*:*?g* ∗ *?g′ dvd a ?g*∗ *?g′ dvd b*
by (*auto simp add*: *zdvd-mult-div-cancel* [*OF dvdg*(*1*)]
zdvd-mult-div-cancel [*OF dvdg*(*2*)] *dvd-def*)
have *?g* ≠ *0* **using** *nz* **by** *simp*
then have *gp*: *?g* ≠ *0* **using** *igcd-pos*[**where** *i*=*a* **and** *j*=*b*] **by** *arith*
from *igcd-greatest* [*OF dvdgg′*] **have** *?g* ∗ *?g′ dvd ?g* **.**
with *zdvd-mult-cancel1* [*OF gp*] **have** |*?g′*| = *1* **by** *simp*
with *igcd-pos* **show** *?g′* = *1* **by** *simp*
qed

definition *ilcm* = (λ*i j*. *int* (*lcm*(*nat*(*abs i*),*nat*(*abs j*))))

lemma *dvd-ilcm-self1*[*simp*]: *i dvd ilcm i j*
by(*simp add*:*ilcm-def dvd-int-iff*)

lemma *dvd-ilcm-self2*[*simp*]: *j dvd ilcm i j*
by(*simp add*:*ilcm-def dvd-int-iff*)

lemma *dvd-imp-dvd-ilcm1*:
  assumes *k dvd i* **shows** *k dvd* (*ilcm i j*)
proof −
  have *nat*(*abs k*) *dvd nat*(*abs i*) **using** ‹*k dvd i*›
    by(*simp add*:*int-dvd-iff*[*symmetric*] *dvd-int-iff*[*symmetric*] *zdvd-abs1*)
  thus *?thesis* **by**(*simp add*:*ilcm-def dvd-int-iff*)(*blast intro*: *dvd-trans*)
qed

lemma *dvd-imp-dvd-ilcm2*:
  assumes *k dvd j* **shows** *k dvd* (*ilcm i j*)
proof −
  have *nat*(*abs k*) *dvd nat*(*abs j*) **using** ‹*k dvd j*›
    by(*simp add*:*int-dvd-iff*[*symmetric*] *dvd-int-iff*[*symmetric*] *zdvd-abs1*)
  thus *?thesis* **by**(*simp add*:*ilcm-def dvd-int-iff*)(*blast intro*: *dvd-trans*)
qed

lemma *zdvd-self-abs1*: (*d*::*int*) *dvd* (*abs d*)
by (*case-tac d <0*, *simp-all*)

lemma *zdvd-self-abs2*: (*abs* (*d*::*int*)) *dvd d*
by (*case-tac d<0*, *simp-all*)

lemma *lcm-pos*:
  assumes *mpos*: *m* > *0*
  and *npos*: *n>0*
  shows *lcm* (*m*,*n*) > *0*
proof(*rule ccontr*, *simp add*: *lcm-def gcd-zero*)
assume *h*:*m*∗*n div gcd*(*m*,*n*) = *0*
from *mpos npos* **have** *gcd* (*m*,*n*) ≠ *0* **using** *gcd-zero* **by** *simp*
hence *gcdp*: *gcd*(*m*,*n*) > *0* **by** *simp*
with *h*
have *m*∗*n* < *gcd*(*m*,*n*)
  by (*cases m* ∗ *n* < *gcd* (*m*, *n*)) (*auto simp add*: *div-if*[*OF gcdp*, **where** *m*=*m*∗*n*])
moreover
have *gcd*(*m*,*n*) *dvd m* **by** *simp*
 with *mpos dvd-imp-le* **have** *t1*:*gcd*(*m*,*n*) ≤ *m* **by** *simp*
 with *npos* **have** *t1*:*gcd*(*m*,*n*)∗*n* ≤ *m*∗*n* **by** *simp*
 have *gcd*(*m*,*n*) ≤ *gcd*(*m*,*n*)∗*n* **using** *npos* **by** *simp*
 with *t1* **have** *gcd*(*m*,*n*) ≤ *m*∗*n* **by** *arith*
ultimately show *False* **by** *simp*
qed

lemma *ilcm-pos*:
  assumes *anz*: *a* ≠ *0*
  and *bnz*: *b* ≠ *0*
  shows *0* < *ilcm a b*

proof−
let *?na* = *nat* (*abs a*)
let *?nb* = *nat* (*abs b*)
have *nap*: *?na* >*0* **using** *anz* **by** *simp*
have *nbp*: *?nb* >*0* **using** *bnz* **by** *simp*
have *0* < *lcm* (*?na*,*?nb*) **by** (*rule lcm-pos*[*OF nap nbp*])
thus *?thesis* **by** (*simp add*: *ilcm-def*)
qed

end

8 Abstract rational numbers

theory *Abstract-Rat*
imports *GCD*
begin

types *Num* = *int* × *int*

abbreviation
Num0-syn :: *Num* (*0*$_N$)
where *0*$_N$ ≡ (*0*, *0*)

abbreviation
Numi-syn :: *int* ⇒ *Num* (-$_N$)
where *i*$_N$ ≡ (*i*, *1*)

definition
isnormNum :: *Num* ⇒ *bool*
where
isnormNum = (λ(*a*,*b*). (*if a* = *0 then b* = *0 else b* > *0* ∧ *igcd a b* = *1*))

definition
normNum :: *Num* ⇒ *Num*
where
normNum = (λ(*a*,*b*). (*if a*=*0* ∨ *b* = *0 then* (*0*,*0*) *else*
(*let g* = *igcd a b*
in if b > *0 then* (*a div g*, *b div g*) *else* (− (*a div g*), − (*b div g*)))))

lemma *normNum-isnormNum* [*simp*]: *isnormNum* (*normNum x*)
proof −
have ∃ *a b*. *x* = (*a*,*b*) **by** *auto*
then obtain *a b* **where** *x*[*simp*]: *x* = (*a*,*b*) **by** *blast*
{assume *a*=*0* ∨ *b* = *0* **hence** *?thesis* **by** (*simp add*: *normNum-def isnormNum-def*)**}**

moreover
{assume *anz*: *a* ≠ *0* **and** *bnz*: *b* ≠ *0*
let *?g* = *igcd a b*

let *?a′ = a div ?g*
let *?b′ = b div ?g*
let *?g′ = igcd ?a′ ?b′*
from *anz bnz* **have** *?g ≠ 0* **by** *simp* **with** *igcd-pos*[*of a b*]
have *gpos*: *?g > 0* **by** *arith*
have *gdvd*: *?g dvd a ?g dvd b* **by** (*simp-all add*: *igcd-dvd1 igcd-dvd2*)
from *zdvd-mult-div-cancel*[*OF gdvd*(*1*)] *zdvd-mult-div-cancel*[*OF gdvd*(*2*)]
anz bnz
have *nz′*:*?a′ ≠ 0 ?b′ ≠ 0*
by − (*rule notI*,*simp add*:*igcd-def*)+
from *anz bnz* **have** *stupid*: *a ≠ 0 ∨ b ≠ 0* **by** *blast*
from *div-igcd-relprime*[*OF stupid*] **have** *gp1*: *?g′ = 1* **.**
from *bnz* **have** *b < 0 ∨ b > 0* **by** *arith*
moreover
{assume *b*: *b > 0*
from *pos-imp-zdiv-nonneg-iff*[*OF gpos*] *b*
have *?b′ ≥ 0* **by** *simp*
with *nz′* **have** *b′*: *?b′ > 0* **by** *simp*
from *b b′ anz bnz nz′ gp1* **have** *?thesis*
by (*simp add*: *isnormNum-def normNum-def Let-def split-def fst-conv snd-conv*)**}**
moreover {assume *b*: *b < 0*
{assume *b′*: *?b′ ≥ 0*
from *gpos* **have** *th*: *?g ≥ 0* **by** *arith*
from *mult-nonneg-nonneg*[*OF th b′*] *zdvd-mult-div-cancel*[*OF gdvd*(*2*)]
have *False* **using** *b* **by** *simp* **}**
hence *b′*: *?b′ < 0* **by** (*presburger add*: *linorder-not-le*[*symmetric*])
from *anz bnz nz′ b b′ gp1* **have** *?thesis*
by (*simp add*: *isnormNum-def normNum-def Let-def split-def fst-conv snd-conv*)**}**
ultimately have *?thesis* **by** *blast*
}
ultimately show *?thesis* **by** *blast*
qed

Arithmetic over Num

definition
Nadd :: *Num ⇒ Num ⇒ Num* (**infixl** $+_N$ *60*)
where
Nadd = (*λ*(*a*,*b*) (*a′*,*b′*). *if a = 0 ∨ b = 0 then normNum*(*a′*,*b′*)
else if a′=0 ∨ b′ = 0 then normNum(*a*,*b*)
else normNum(*a∗b′ + b∗a′*, *b∗b′*))

definition
Nmul :: *Num ⇒ Num ⇒ Num* (**infixl** $*_N$ *60*)
where
Nmul = (*λ*(*a*,*b*) (*a′*,*b′*). *let g = igcd* (*a∗a′*) (*b∗b′*)
in (*a∗a′ div g*, *b∗b′ div g*))

definition
Nneg :: *Num* ⇒ *Num* ($\sim_N$)
where
Nneg ≡ (λ(*a*,*b*). (−*a*,*b*))

definition
Nsub :: *Num* ⇒ *Num* ⇒ *Num* (**infixl** $-_N$ *60*)
where
Nsub = (λ*a b*. *a* $+_N$ $\sim_N$ *b*)

definition
Ninv :: *Num* ⇒ *Num*
where
Ninv ≡ λ(*a*,*b*). *if a* < *0 then* (−*b*, |*a*|) *else* (*b*,*a*)

definition
Ndiv :: *Num* ⇒ *Num* ⇒ *Num* (**infixl** $\div_N$ *60*)
where
Ndiv ≡ λ*a b*. *a* $*_N$ *Ninv b*

lemma *Nneg-normN*[*simp*]: *isnormNum x* ⟹ *isnormNum* ($\sim_N$ *x*)
by(*simp add*: *isnormNum-def Nneg-def split-def*)
lemma *Nadd-normN*[*simp*]: *isnormNum* (*x* $+_N$ *y*)
by (*simp add*: *Nadd-def split-def*)
lemma *Nsub-normN*[*simp*]: ⟦ *isnormNum y*⟧ ⟹ *isnormNum* (*x* $-_N$ *y*)
by (*simp add*: *Nsub-def split-def*)
lemma *Nmul-normN*[*simp*]: **assumes** *xn*:*isnormNum x* **and** *yn*: *isnormNum y*
shows *isnormNum* (*x* $*_N$ *y*)
proof−
have ∃ *a b*. *x* = (*a*,*b*) **and** ∃ *a′ b′*. *y* = (*a′*,*b′*) **by** *auto*
then obtain *a b a′ b′* **where** *ab*: *x* = (*a*,*b*) **and** *ab′*: *y* = (*a′*,*b′*) **by** *blast*
{assume *a* = *0*
hence *?thesis* **using** *xn ab ab′*
by (*simp add*: *igcd-def isnormNum-def Let-def Nmul-def split-def*)**}**
moreover
{assume *a′* = *0*
hence *?thesis* **using** *yn ab ab′*
by (*simp add*: *igcd-def isnormNum-def Let-def Nmul-def split-def*)**}**
moreover
{assume *a*: *a* ≠*0* **and** *a′*: *a′*≠*0*
hence *bp*: *b* > *0 b′* > *0* **using** *xn yn ab ab′* **by** (*simp-all add*: *isnormNum-def*)
from *mult-pos-pos*[*OF bp*] **have** *x* $*_N$ *y* = *normNum* (*a*∗*a′*, *b*∗*b′*)
using *ab ab′ a a′ bp* **by** (*simp add*: *Nmul-def Let-def split-def normNum-def*)
hence *?thesis* **by** *simp***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *Ninv-normN*[*simp*]: *isnormNum x* ⟹ *isnormNum* (*Ninv x*)
by (*simp add*: *Ninv-def isnormNum-def split-def*)

(*cases fst x = 0*, *auto simp add*: *igcd-commute*)

lemma *isnormNum-int*[*simp*]:
isnormNum 0_N *isnormNum* $(1::int)_N$ $i \neq 0 \Longrightarrow$ *isnormNum* i_N
by (*simp-all add*: *isnormNum-def igcd-def*)

Relations over Num

definition
Nlt0:: *Num* $\Rightarrow$ *bool* $(0>_N)$
where
Nlt0 = $(\lambda(a,b).\ a < 0)$

definition
Nle0:: *Num* $\Rightarrow$ *bool* $(0\geq_N)$
where
Nle0 = $(\lambda(a,b).\ a \leq 0)$

definition
Ngt0:: *Num* $\Rightarrow$ *bool* $(0<_N)$
where
Ngt0 = $(\lambda(a,b).\ a > 0)$

definition
Nge0:: *Num* $\Rightarrow$ *bool* $(0\leq_N)$
where
Nge0 = $(\lambda(a,b).\ a \geq 0)$

definition
Nlt :: *Num* $\Rightarrow$ *Num* $\Rightarrow$ *bool* (**infix** $<_N$ 55)
where
Nlt = $(\lambda a\ b.\ 0>_N\ (a\ -_N\ b))$

definition
Nle :: *Num* $\Rightarrow$ *Num* $\Rightarrow$ *bool* (**infix** $\leq_N$ 55)
where
Nle = $(\lambda a\ b.\ 0\geq_N\ (a\ -_N\ b))$

definition
INum = $(\lambda(a,b).$ *of-int a* / *of-int b*$)$

lemma *INum-int* [*simp*]: *INum* i_N = ((*of-int i*) ::'*a*::*field*) *INum* 0_N = (0::'*a*::*field*)
by (*simp-all add*: *INum-def*)

lemma *isnormNum-unique*[*simp*]:
assumes *na*: *isnormNum x* **and** *nb*: *isnormNum y*
shows ((*INum x* ::'*a*::{*ring-char-0*,*field*, *division-by-zero*}) = *INum y*) = (x = y) (**is** *?lhs* = *?rhs*)
proof
have $\exists\ a\ b\ a'\ b'.\ x = (a,b) \wedge y = (a',b')$ **by** *auto*

```
  then obtain a b a' b' where xy[simp]: x = (a,b) y=(a',b') by blast
  assume H: ?lhs
  {assume a = 0 ∨ b = 0 ∨ a' = 0 ∨ b' = 0 hence ?rhs
     using na nb H
     apply (simp add: INum-def split-def isnormNum-def)
     apply (cases a = 0, simp-all)
     apply (cases b = 0, simp-all)
     apply (cases a' = 0, simp-all)
     apply (cases a' = 0, simp-all add: of-int-eq-0-iff)
     done}
  moreover
  { assume az: a ≠ 0 and bz: b ≠ 0 and a'z: a'≠0 and b'z: b'≠0
   from az bz a'z b'z na nb have pos: b > 0 b' > 0 by (simp-all add: isnormNum-def)
    from prems have eq:a * b' = a'*b
     by (simp add: INum-def  eq-divide-eq divide-eq-eq of-int-mult[symmetric] del:
of-int-mult)
    from prems have gcd1: igcd a b = 1 igcd b a = 1 igcd a' b' = 1 igcd b' a' =
1
     by (simp-all add: isnormNum-def add: igcd-commute)
    from eq have raw-dvd: a dvd a'*b b dvd b'*a a' dvd a*b' b' dvd b*a'
     apply(unfold dvd-def)
     apply (rule-tac x=b' in exI, simp add: mult-ac)
     apply (rule-tac x=a' in exI, simp add: mult-ac)
     apply (rule-tac x=b in exI, simp add: mult-ac)
     apply (rule-tac x=a in exI, simp add: mult-ac)
     done
    from zdvd-dvd-eq[OF bz zrelprime-dvd-mult[OF gcd1(2) raw-dvd(2)]
     zrelprime-dvd-mult[OF gcd1(4) raw-dvd(4)]]
     have eq1: b = b' using pos by simp-all
     with eq have a = a' using pos by simp
     with eq1 have ?rhs by simp}
  ultimately show ?rhs by blast
next
  assume ?rhs thus ?lhs by simp
qed


lemma isnormNum0[simp]: isnormNum x ⟹ (INum x = (0::'a::{ring-char-0,
field,division-by-zero})) = (x = 0_N)
  unfolding INum-int(2)[symmetric]
  by (rule isnormNum-unique, simp-all)

lemma of-int-div-aux: d ~= 0 ==> ((of-int x)::'a::{field, ring-char-0}) / (of-int
d) =
    of-int (x div d) + (of-int (x mod d)) / ((of-int d)::'a)
proof -
  assume d ~= 0
  hence dz: of-int d ≠ (0::'a) by (simp add: of-int-eq-0-iff)
  let ?t = of-int (x div d) * ((of-int d)::'a) + of-int(x mod d)
```

let *?f* = $\lambda x.\ x$ / *of-int d*
have $x = (x\ div\ d) * d + x\ mod\ d$
by *auto*
then have *eq*: *of-int x* = *?t*
by (*simp only*: *of-int-mult*[*symmetric*] *of-int-add* [*symmetric*])
then have *of-int x* / *of-int d* = *?t* / *of-int d*
using *cong*[*OF refl*[*of ?f*] *eq*] **by** *simp*
then show *?thesis* **by** (*simp add*: *add-divide-distrib ring-simps prems*)
qed

lemma *of-int-div*: (*d*::*int*) ~= *0* ==> *d dvd n* ==>
(*of-int*(*n div d*)::'*a*::{*field*, *ring-char-0*}) = *of-int n* / *of-int d*
apply (*frule of-int-div-aux* [*of d n*, **where** *?'a* = '*a*])
apply *simp*
apply (*simp add*: *zdvd-iff-zmod-eq-0*)
done

lemma *normNum*[*simp*]: *INum* (*normNum x*) = (*INum x* :: '*a*::{*ring-char-0*,*field*, *division-by-zero*})
proof−
have ∃ *a b*. *x* = (*a*,*b*) **by** *auto*
then obtain *a b* **where** *x*[*simp*]: *x* = (*a*,*b*) **by** *blast*
{assume *a*=*0* ∨ *b* = *0* **hence** *?thesis*
by (*simp add*: *INum-def normNum-def split-def Let-def*)**}**
moreover
{assume *a*: *a*≠*0* **and** *b*: *b*≠*0*
let *?g* = *igcd a b*
from *a b* **have** *g*: *?g* ≠ *0***by** *simp*
from *of-int-div*[*OF g*, **where** *?'a* = '*a*]
have *?thesis* **by** (*auto simp add*: *INum-def normNum-def split-def Let-def*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *INum-normNum-iff* [*code*]: (*INum x* ::'*a*::{*field*, *division-by-zero*, *ring-char-0*})
= *INum y* ⟷ *normNum x* = *normNum y* (**is** *?lhs* = *?rhs*)
proof −
have *normNum x* = *normNum y* ⟷ (*INum* (*normNum x*) :: '*a*) = *INum* (*normNum y*)
by (*simp del*: *normNum*)
also have ... = *?lhs* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *Nadd*[*simp*]: *INum* (*x* +$_N$ *y*) = *INum x* + (*INum y* :: '*a* :: {*ring-char-0*,*division-by-zero*,*field*})
proof−
let *?z* = *0*:: '*a*
have ∃ *a b*. *x* = (*a*,*b*) ∃ *a' b'*. *y* = (*a'*,*b'*) **by** *auto*
then obtain *a b a' b'* **where** *x*[*simp*]: *x* = (*a*,*b*)

```
  and y[simp]: y = (a',b') by blast
 {assume a=0 ∨ a'= 0 ∨ b =0 ∨ b' = 0 hence ?thesis
    apply (cases a=0,simp-all add: Nadd-def)
    apply (cases b= 0,simp-all add: INum-def)
     apply (cases a'= 0,simp-all)
     apply (cases b'= 0,simp-all)
     done }
 moreover
 {assume aa':a ≠ 0 a'≠ 0 and bb': b ≠ 0 b' ≠ 0
  {assume z: a * b' + b * a' = 0
    hence of-int (a*b' + b*a') / (of-int b* of-int b') = ?z by simp
   hence of-int b' * of-int a / (of-int b * of-int b') + of-int b * of-int a' / (of-int
b * of-int b') = ?z  by (simp add:add-divide-distrib)
    hence th: of-int a / of-int b + of-int a' / of-int b' = ?z using bb' aa' by
simp
    from z aa' bb' have ?thesis
     by (simp add: th Nadd-def normNum-def INum-def split-def)}
  moreover {assume z: a * b' + b * a' ≠ 0
    let ?g = igcd (a * b' + b * a') (b*b')
    have gz: ?g ≠ 0 using z by simp
    have ?thesis using aa' bb' z gz
      of-int-div[where ?'a = 'a,
      OF gz igcd-dvd1[where i=a * b' + b * a' and j=b*b']]
      of-int-div[where ?'a = 'a,
      OF gz igcd-dvd2[where i=a * b' + b * a' and j=b*b']]
    by (simp add: x y Nadd-def INum-def normNum-def Let-def add-divide-distrib)}
  ultimately have ?thesis using aa' bb'
    by (simp add: Nadd-def INum-def normNum-def x y Let-def) }
 ultimately show ?thesis by blast
qed

lemma Nmul[simp]: INum (x *N y) = INum x * (INum y:: 'a :: {ring-char-0,division-by-zero,field})

proof−
 let ?z = 0::'a
 have ∃ a b. x = (a,b)  ∃ a' b'. y = (a',b') by auto
 then obtain a b a' b' where x: x = (a,b) and y: y = (a',b') by blast
 {assume a=0 ∨ a'= 0 ∨ b = 0 ∨ b' = 0 hence ?thesis
    apply (cases a=0,simp-all add: x y Nmul-def INum-def Let-def)
    apply (cases b=0,simp-all)
    apply (cases a'=0,simp-all)
    done }
 moreover
 {assume z: a ≠ 0 a' ≠ 0 b ≠ 0 b' ≠ 0
  let ?g=igcd (a*a') (b*b')
  have gz: ?g ≠ 0 using z by simp
 from z of-int-div[where ?'a = 'a, OF gz igcd-dvd1[where i=a*a' and j=b*b']]

    of-int-div[where ?'a = 'a , OF gz igcd-dvd2[where i=a*a' and j=b*b']]
```

have *?thesis* **by** (*simp add*: *Nmul-def x y Let-def INum-def*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *Nneg*[*simp*]: *INum* ($\sim_N$ *x*) = − (*INum x* ::′*a*:: *field*)
by (*simp add*: *Nneg-def split-def INum-def*)

lemma *Nsub*[*simp*]: **shows** *INum* (*x* −$_N$ *y*) = *INum x* − (*INum y*:: ′*a* :: {*ring-char-0*,*division-by-zero*,*field*})
by (*simp add*: *Nsub-def split-def*)

lemma *Ninv*[*simp*]: *INum* (*Ninv x*) = (*1*::′*a* :: {*division-by-zero*,*field*}) / (*INum x*)
by (*simp add*: *Ninv-def INum-def split-def*)

lemma *Ndiv*[*simp*]: *INum* (*x* ÷$_N$ *y*) = *INum x* / (*INum y* ::′*a* :: {*ring-char-0*, *division-by-zero*,*field*}) **by** (*simp add*: *Ndiv-def*)

lemma *Nlt0-iff*[*simp*]: **assumes** *nx*: *isnormNum x*
shows ((*INum x* :: ′*a* :: {*ring-char-0*,*division-by-zero*,*ordered-field*})< *0*) = *0*>$_N$ *x*
proof−
have ∃ *a b*. *x* = (*a*,*b*) **by** *simp*
then obtain *a b* **where** *x*[*simp*]:*x* = (*a*,*b*) **by** *blast*
{assume *a* = *0* **hence** *?thesis* **by** (*simp add*: *Nlt0-def INum-def*) **}**
moreover
{assume *a*: *a*≠*0* **hence** *b*: (*of-int b*::′*a*) > *0* **using** *nx* **by** (*simp add*: *isnormNum-def*)
from *pos-divide-less-eq*[*OF b*, **where** *b*=*of-int a* **and** *a*=*0*::′*a*]
have *?thesis* **by** (*simp add*: *Nlt0-def INum-def*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *Nle0-iff*[*simp*]:**assumes** *nx*: *isnormNum x*
shows ((*INum x* :: ′*a* :: {*ring-char-0*,*division-by-zero*,*ordered-field*}) ≤ *0*) = *0*≥$_N$ *x*
proof−
have ∃ *a b*. *x* = (*a*,*b*) **by** *simp*
then obtain *a b* **where** *x*[*simp*]:*x* = (*a*,*b*) **by** *blast*
{assume *a* = *0* **hence** *?thesis* **by** (*simp add*: *Nle0-def INum-def*) **}**
moreover
{assume *a*: *a*≠*0* **hence** *b*: (*of-int b* :: ′*a*) > *0* **using** *nx* **by** (*simp add*: *isnormNum-def*)
from *pos-divide-le-eq*[*OF b*, **where** *b*=*of-int a* **and** *a*=*0*::′*a*]
have *?thesis* **by** (*simp add*: *Nle0-def INum-def*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *Ngt0-iff*[*simp*]:**assumes** *nx*: *isnormNum x* **shows** ((*INum x* :: ′*a* :: {*ring-char-0*,*division-by-zero*,*ordere*
0) = *0*<$_N$ *x*
proof−
have ∃ *a b*. *x* = (*a*,*b*) **by** *simp*

then obtain *a b* **where** *x*[*simp*]:*x* = (*a*,*b*) **by** *blast*
{assume *a* = *0* **hence** *?thesis* **by** (*simp add*: *Ngt0-def INum-def*) **}**
moreover
{assume *a*: *a*≠*0* **hence** *b*: (*of-int b*::′*a*) > *0* **using** *nx* **by** (*simp add*: *isnormNum-def*)
from *pos-less-divide-eq*[*OF b*, **where** *b*=*of-int a* **and** *a*=*0*::′*a*]
have *?thesis* **by** (*simp add*: *Ngt0-def INum-def*)**}**
ultimately show *?thesis* **by** *blast*
qed
lemma *Nge0-iff*[*simp*]:**assumes** *nx*: *isnormNum x*
shows ((*INum x* :: ′*a* :: {*ring-char-0*,*division-by-zero*,*ordered-field*}) ≥ *0*) = *0*≤$_N$ *x*
proof−
have ∃ *a b*. *x* = (*a*,*b*) **by** *simp*
then obtain *a b* **where** *x*[*simp*]:*x* = (*a*,*b*) **by** *blast*
{assume *a* = *0* **hence** *?thesis* **by** (*simp add*: *Nge0-def INum-def*) **}**
moreover
{assume *a*: *a*≠*0* **hence** *b*: (*of-int b*::′*a*) > *0* **using** *nx* **by** (*simp add*: *isnormNum-def*)
from *pos-le-divide-eq*[*OF b*, **where** *b*=*of-int a* **and** *a*=*0*::′*a*]
have *?thesis* **by** (*simp add*: *Nge0-def INum-def*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *Nlt-iff*[*simp*]: **assumes** *nx*: *isnormNum x* **and** *ny*: *isnormNum y*
shows ((*INum x* :: ′*a* :: {*ring-char-0*,*division-by-zero*,*ordered-field*}) < *INum y*) = (*x* <$_N$ *y*)
proof−
let *?z* = *0*::′*a*
have ((*INum x* ::′*a*) < *INum y*) = (*INum* (*x* −$_N$ *y*) < *?z*) **using** *nx ny* **by** *simp*
also have ... = (*0*>$_N$ (*x* −$_N$ *y*)) **using** *Nlt0-iff*[*OF Nsub-normN*[*OF ny*]] **by** *simp*
finally show *?thesis* **by** (*simp add*: *Nlt-def*)
qed

lemma *Nle-iff*[*simp*]: **assumes** *nx*: *isnormNum x* **and** *ny*: *isnormNum y*
shows ((*INum x* :: ′*a* :: {*ring-char-0*,*division-by-zero*,*ordered-field*})≤ *INum y*) = (*x* ≤$_N$ *y*)
proof−
have ((*INum x* ::′*a*) ≤ *INum y*) = (*INum* (*x* −$_N$ *y*) ≤ (*0*::′*a*)) **using** *nx ny* **by** *simp*
also have ... = (*0*≥$_N$ (*x* −$_N$ *y*)) **using** *Nle0-iff*[*OF Nsub-normN*[*OF ny*]] **by** *simp*
finally show *?thesis* **by** (*simp add*: *Nle-def*)
qed

lemma *Nadd-commute*: *x* +$_N$ *y* = *y* +$_N$ *x*
proof−
have *n*: *isnormNum* (*x* +$_N$ *y*) *isnormNum* (*y* +$_N$ *x*) **by** *simp-all*
have (*INum* (*x* +$_N$ *y*)::′*a* :: {*ring-char-0*,*division-by-zero*,*field*}) = *INum* (*y* +$_N$ *x*) **by** *simp*

with *isnormNum-unique*[*OF n*] **show** *?thesis* **by** *simp*
qed

lemma[*simp*]: $(0,\ b) +_N y = normNum\ y\ (a,\ 0) +_N y = normNum\ y$
$x +_N (0,\ b) = normNum\ x\ x +_N (a,\ 0) = normNum\ x$
apply (*simp add*: *Nadd-def split-def*, *simp add*: *Nadd-def split-def*)
apply (*subst Nadd-commute*,*simp add*: *Nadd-def split-def*)
apply (*subst Nadd-commute*,*simp add*: *Nadd-def split-def*)
done

lemma *normNum-nilpotent-aux*[*simp*]: **assumes** *nx*: *isnormNum x*
shows $normNum\ x = x$
proof−
let *?a* = *normNum x*
have *n*: *isnormNum ?a* **by** *simp*
have *th*:*INum ?a* = (*INum x* ::*'a* :: {*ring-char-0*, *division-by-zero*,*field*}) **by** *simp*
with *isnormNum-unique*[*OF n nx*]
show *?thesis* **by** *simp*
qed

lemma *normNum-nilpotent*[*simp*]: $normNum\ (normNum\ x) = normNum\ x$
by *simp*
lemma *normNum0*[*simp*]: $normNum\ (0,b) = 0_N\ normNum\ (a,0) = 0_N$
by (*simp-all add*: *normNum-def*)
lemma *normNum-Nadd*: $normNum\ (x +_N y) = x +_N y$ **by** *simp*
lemma *Nadd-normNum1*[*simp*]: $normNum\ x +_N y = x +_N y$
proof−
have *n*: *isnormNum* (*normNum x* $+_N$ *y*) *isnormNum* (*x* $+_N$ *y*) **by** *simp-all*
have *INum* (*normNum x* $+_N$ *y*) = *INum x* + (*INum y* :: *'a* :: {*ring-char-0*, *division-by-zero*,*field*}) **by** *simp*
also have ... = *INum* (*x* $+_N$ *y*) **by** *simp*
finally show *?thesis* **using** *isnormNum-unique*[*OF n*] **by** *simp*
qed
lemma *Nadd-normNum2*[*simp*]: $x +_N normNum\ y = x +_N y$
proof−
have *n*: *isnormNum* (*x* $+_N$ *normNum y*) *isnormNum* (*x* $+_N$ *y*) **by** *simp-all*
have *INum* (*x* $+_N$ *normNum y*) = *INum x* + (*INum y* :: *'a* :: {*ring-char-0*, *division-by-zero*,*field*}) **by** *simp*
also have ... = *INum* (*x* $+_N$ *y*) **by** *simp*
finally show *?thesis* **using** *isnormNum-unique*[*OF n*] **by** *simp*
qed

lemma *Nadd-assoc*: $x +_N y +_N z = x +_N (y +_N z)$
proof−
have *n*: *isnormNum* (*x* $+_N$ *y* $+_N$ *z*) *isnormNum* (*x* $+_N$ (*y* $+_N$ *z*)) **by** *simp-all*
have *INum* (*x* $+_N$ *y* $+_N$ *z*) = (*INum* (*x* $+_N$ (*y* $+_N$ *z*)) :: *'a* :: {*ring-char-0*, *division-by-zero*,*field*}) **by** *simp*
with *isnormNum-unique*[*OF n*] **show** *?thesis* **by** *simp*
qed

lemma *Nmul-commute*: $isnormNum\ x \Longrightarrow isnormNum\ y \Longrightarrow x *_N y = y *_N x$
by (*simp add*: *Nmul-def split-def Let-def igcd-commute mult-commute*)

lemma *Nmul-assoc*: **assumes** *nx*: *isnormNum x* **and** *ny*:*isnormNum y* **and** *nz*:*isnormNum z*
shows $x *_N y *_N z = x *_N (y *_N z)$
proof−
from *nx ny nz* **have** *n*: $isnormNum\ (x *_N y *_N z)\ isnormNum\ (x *_N (y *_N z))$

by *simp-all*
have $INum\ (x +_N y +_N z) = (INum\ (x +_N (y +_N z)) :: 'a :: \{$*ring-char-0, division-by-zero,field*$\})$ **by** *simp*
with *isnormNum-unique*[*OF n*] **show** *?thesis* **by** *simp*
qed

lemma *Nsub0*: **assumes** *x*: *isnormNum x* **and** *y*:*isnormNum y* **shows** $(x -_N y = 0_N) = (x = y)$
proof−
{fix $h :: 'a :: \{$*ring-char-0,division-by-zero,ordered-field*$\}$
from *isnormNum-unique*[**where** $?'a = 'a$, *OF Nsub-normN*[*OF y*], **where** $y=0_N$]
have $(x -_N y = 0_N) = (INum\ (x -_N y) = (INum\ 0_N :: 'a))$ **by** *simp*
also have $\ldots = (INum\ x = (INum\ y :: 'a))$ **by** *simp*
also have $\ldots = (x = y)$ **using** *x y* **by** *simp*
finally show *?thesis* **.}**
qed

lemma *Nmul0*[*simp*]: $c *_N 0_N = 0_N \quad 0_N *_N c = 0_N$
by (*simp-all add*: *Nmul-def Let-def split-def*)

lemma *Nmul-eq0*[*simp*]: **assumes** *nx*:*isnormNum x* **and** *ny*: *isnormNum y*
shows $(x *_N y = 0_N) = (x = 0_N \vee y = 0_N)$
proof−
{fix $h :: 'a :: \{$*ring-char-0,division-by-zero,ordered-field*$\}$
have $\exists\ a\ b\ a'\ b'.\ x = (a,b) \wedge y = (a',b')$ **by** *auto*
then obtain $a\ b\ a'\ b'$ **where** *xy*[*simp*]: $x = (a,b)\ y = (a',b')$ **by** *blast*
have *n0*: $isnormNum\ 0_N$ **by** *simp*
show *?thesis* **using** *nx ny*
apply (*simp only*: *isnormNum-unique*[**where** $?'a = 'a$, *OF Nmul-normN*[*OF nx ny*] *n0*, *symmetric*] *Nmul*[**where** $?'a = 'a$])
apply (*simp add*: *INum-def split-def isnormNum-def fst-conv snd-conv*)
apply (*cases a=0*,*simp-all*)
apply (*cases a'=0*,*simp-all*)
done }
qed
lemma *Nneg-Nneg*[*simp*]: $\sim_N (\sim_N c) = c$
by (*simp add*: *Nneg-def split-def*)

lemma *Nmul1*[*simp*]:
  isnormNum c $\Longrightarrow$ $1_N *_N c = c$
  isnormNum c $\Longrightarrow$ $c *_N 1_N = c$
  apply (*simp-all add*: *Nmul-def Let-def split-def isnormNum-def*)
  by (*cases fst c* = *0*, *simp-all*,*cases c*, *simp-all*)+

end

9 Rational numbers

theory *Rational*
imports *Abstract-Rat*
uses (*rat-arith.ML*)
begin

9.1 Rational numbers

9.1.1 Equivalence of fractions

definition
  fraction :: (*int* × *int*) *set* **where**
  fraction = {*x*. *snd x* ≠ *0*}

definition
  ratrel :: ((*int* × *int*) × (*int* × *int*)) *set* **where**
  ratrel = {(*x*,*y*). *snd x* ≠ *0* ∧ *snd y* ≠ *0* ∧ *fst x* ∗ *snd y* = *fst y* ∗ *snd x*}

lemma *fraction-iff* [*simp*]: (*x* ∈ *fraction*) = (*snd x* ≠ *0*)
by (*simp add*: *fraction-def*)

lemma *ratrel-iff* [*simp*]:
  ((*x*,*y*) ∈ *ratrel*) =
   (*snd x* ≠ *0* ∧ *snd y* ≠ *0* ∧ *fst x* ∗ *snd y* = *fst y* ∗ *snd x*)
by (*simp add*: *ratrel-def*)

lemma *refl-ratrel*: *refl fraction ratrel*
by (*auto simp add*: *refl-def fraction-def ratrel-def*)

lemma *sym-ratrel*: *sym ratrel*
by (*simp add*: *ratrel-def sym-def*)

lemma *trans-ratrel-lemma*:
  assumes *1*: *a* ∗ *b′* = *a′* ∗ *b*
  assumes *2*: *a′* ∗ *b″* = *a″* ∗ *b′*
  assumes *3*: *b′* ≠ (*0*::*int*)
  shows *a* ∗ *b″* = *a″* ∗ *b*
proof −
  have *b′* ∗ (*a* ∗ *b″*) = *b″* ∗ (*a* ∗ *b′*) **by** *simp*
  also note *1*

also have $b'' * (a' * b) = b * (a' * b'')$ **by** *simp*
also note *2*
also have $b * (a'' * b') = b' * (a'' * b)$ **by** *simp*
finally have $b' * (a * b'') = b' * (a'' * b)$ **.**
with *3* **show** $a * b'' = a'' * b$ **by** *simp*
qed

lemma *trans-ratrel*: *trans ratrel*
by (*auto simp add*: *trans-def elim*: *trans-ratrel-lemma*)

lemma *equiv-ratrel*: *equiv fraction ratrel*
by (*rule equiv.intro* [*OF refl-ratrel sym-ratrel trans-ratrel*])

lemmas *equiv-ratrel-iff* [*iff*] = *eq-equiv-class-iff* [*OF equiv-ratrel*]

lemma *equiv-ratrel-iff2*:
⟦*snd x* ≠ *0*; *snd y* ≠ *0*⟧
⟹ (*ratrel `` {x} = ratrel `` {y}*) = ((*x*,*y*) ∈ *ratrel*)
by (*rule eq-equiv-class-iff* [*OF equiv-ratrel*], *simp-all*)

9.1.2 The type of rational numbers

typedef (*Rat*) *rat* = *fraction//ratrel*
proof
have (*0*,*1*) ∈ *fraction* **by** (*simp add*: *fraction-def*)
thus *ratrel``{(0,1)}* ∈ *fraction//ratrel* **by** (*rule quotientI*)
qed

lemma *ratrel-in-Rat* [*simp*]: *snd x* ≠ *0* ⟹ *ratrel``{x}* ∈ *Rat*
by (*simp add*: *Rat-def quotientI*)

declare *Abs-Rat-inject* [*simp*] *Abs-Rat-inverse* [*simp*]

definition
Fract :: *int* ⇒ *int* ⇒ *rat* **where**
[*code func del*]: *Fract a b* = *Abs-Rat* (*ratrel``{(a,b)}*)

lemma *Fract-zero*:
Fract k 0 = *Fract l 0*
by (*simp add*: *Fract-def ratrel-def*)

theorem *Rat-cases* [*case-names Fract*, *cases type*: *rat*]:
(!!*a b*. *q* = *Fract a b* ==> *b* ≠ *0* ==> *C*) ==> *C*
by (*cases q*) (*clarsimp simp add*: *Fract-def Rat-def fraction-def quotient-def*)

theorem *Rat-induct* [*case-names Fract*, *induct type*: *rat*]:
(!!*a b*. *b* ≠ *0* ==> *P* (*Fract a b*)) ==> *P q*
by (*cases q*) *simp*

9.1.3 Congruence lemmas

lemma *add-congruent2*:
(λx y. ratrel``{(fst x * snd y + fst y * snd x, snd x * snd y)})
respects2 ratrel
apply (*rule equiv-ratrel* [*THEN congruent2-commuteI*])
apply (*simp-all add*: *left-distrib*)
done

lemma *minus-congruent*:
(λx. ratrel``{(− fst x, snd x)}) *respects ratrel*
by (*simp add*: *congruent-def*)

lemma *mult-congruent2*:
(λx y. ratrel``{(fst x * fst y, snd x * snd y)}) *respects2 ratrel*
by (*rule equiv-ratrel* [*THEN congruent2-commuteI*], *simp-all*)

lemma *inverse-congruent*:
(λx. ratrel``{*if fst x=0 then* (0,1) *else* (snd x, fst x)}) *respects ratrel*
by (*auto simp add*: *congruent-def mult-commute*)

lemma *le-congruent2*:
(λx y. {(fst x * snd y)*(snd x * snd y) ≤ (fst y * snd x)*(snd x * snd y)})
respects2 ratrel
proof (*clarsimp simp add*: *congruent2-def*)
fix a b a′ b′ c d c′ d′::*int*
assume *neq*: b ≠ 0 b′ ≠ 0 d ≠ 0 d′ ≠ 0
assume *eq1*: a * b′ = a′ * b
assume *eq2*: c * d′ = c′ * d

let *?le* = λa b c d. ((a * d) * (b * d) ≤ (c * b) * (b * d))
{
fix a b c d x :: *int* **assume** *x*: x ≠ 0
have *?le* a b c d = *?le* (a * x) (b * x) c d
proof −
from x **have** 0 < x * x **by** (*auto simp add*: *zero-less-mult-iff*)
hence *?le* a b c d =
((a * d) * (b * d) * (x * x) ≤ (c * b) * (b * d) * (x * x))
by (*simp add*: *mult-le-cancel-right*)
also have ... = *?le* (a * x) (b * x) c d
by (*simp add*: *mult-ac*)
finally show *?thesis* **.**
qed
} note *le-factor* = *this*

let *?D* = b * d **and** *?D′* = b′ * d′
from *neq* **have** *D*: *?D* ≠ 0 **by** *simp*
from *neq* **have** *?D′* ≠ 0 **by** *simp*
hence *?le* a b c d = *?le* (a * *?D′*) (b * *?D′*) c d
by (*rule le-factor*)

also have ... = $((a * b') * ?D * ?D' * d * d' \leq (c * d') * ?D * ?D' * b * b')$
by (*simp add*: *mult-ac*)
also have ... = $((a' * b) * ?D * ?D' * d * d' \leq (c' * d) * ?D * ?D' * b * b')$
by (*simp only*: *eq1 eq2*)
also have ... = *?le* $(a' * ?D)$ $(b' * ?D)$ c' d'
by (*simp add*: *mult-ac*)
also from *D* **have** ... = *?le* a' b' c' d'
by (*rule le-factor* [*symmetric*])
finally show *?le a b c d* = *?le* a' b' c' d' **.**
qed

lemmas *UN-ratrel* = *UN-equiv-class* [*OF equiv-ratrel*]
lemmas *UN-ratrel2* = *UN-equiv-class2* [*OF equiv-ratrel equiv-ratrel*]

9.1.4 Standard operations on rational numbers

instance *rat* :: *zero*
Zero-rat-def: *0* == *Fract 0 1* **..**
lemmas [*code func del*] = *Zero-rat-def*

instance *rat* :: *one*
One-rat-def: *1* == *Fract 1 1* **..**
lemmas [*code func del*] = *One-rat-def*

instance *rat* :: *plus*
add-rat-def:
q + *r* ==
Abs-Rat ($\bigcup x \in$ *Rep-Rat q*. $\bigcup y \in$ *Rep-Rat r*.
ratrel``{(*fst x* * *snd y* + *fst y* * *snd x*, *snd x* * *snd y*)}) **..**
lemmas [*code func del*] = *add-rat-def*

instance *rat* :: *minus*
minus-rat-def:
− *q* == *Abs-Rat* ($\bigcup x \in$ *Rep-Rat q*. *ratrel*``{(− *fst x*, *snd x*)})
diff-rat-def: *q* − *r* == *q* + − (*r*::*rat*) **..**
lemmas [*code func del*] = *minus-rat-def diff-rat-def*

instance *rat* :: *times*
mult-rat-def:
q * *r* ==
Abs-Rat ($\bigcup x \in$ *Rep-Rat q*. $\bigcup y \in$ *Rep-Rat r*.
ratrel``{(*fst x* * *fst y*, *snd x* * *snd y*)}) **..**
lemmas [*code func del*] = *mult-rat-def*

instance *rat* :: *inverse*
inverse-rat-def:
inverse q ==
Abs-Rat ($\bigcup x \in$ *Rep-Rat q*.
ratrel``{*if fst x*=*0 then* (*0*,*1*) *else* (*snd x*, *fst x*)})

divide-rat-def: $q\ /\ r == q * inverse\ (r::rat)$ **..**
lemmas [*code func del*] = *inverse-rat-def divide-rat-def*

instance *rat* :: *ord*
le-rat-def:
$q \leq r == contents\ (\bigcup x \in Rep\text{-}Rat\ q.\ \bigcup y \in Rep\text{-}Rat\ r.$
$\{(fst\ x * snd\ y)*(snd\ x * snd\ y) \leq (fst\ y * snd\ x)*(snd\ x * snd\ y)\})$
less-rat-def: $(z < (w::rat)) == (z \leq w\ \&\ z \neq w)$ **..**
lemmas [*code func del*] = *le-rat-def less-rat-def*

instance *rat* :: *abs*
abs-rat-def: $|q| == if\ q < 0\ then\ -q\ else\ (q::rat)$ **..**

instance *rat* :: *sgn*
sgn-rat-def: $sgn(q::rat) == (if\ q=0\ then\ 0\ else\ if\ 0<q\ then\ 1\ else\ -\ 1)$ **..**

instance *rat* :: *power* **..**

primrec (*rat*)
rat-power-0: $q\ \hat{}\ 0 = 1$
rat-power-Suc: $q\ \hat{}\ (Suc\ n) = (q::rat) * (q\ \hat{}\ n)$

theorem *eq-rat*: $b \neq 0 ==> d \neq 0 ==>$
$(Fract\ a\ b = Fract\ c\ d) = (a * d = c * b)$
by (*simp add*: *Fract-def*)

theorem *add-rat*: $b \neq 0 ==> d \neq 0 ==>$
$Fract\ a\ b + Fract\ c\ d = Fract\ (a * d + c * b)\ (b * d)$
by (*simp add*: *Fract-def add-rat-def add-congruent2 UN-ratrel2*)

theorem *minus-rat*: $b \neq 0 ==> -(Fract\ a\ b) = Fract\ (-a)\ b$
by (*simp add*: *Fract-def minus-rat-def minus-congruent UN-ratrel*)

theorem *diff-rat*: $b \neq 0 ==> d \neq 0 ==>$
$Fract\ a\ b - Fract\ c\ d = Fract\ (a * d - c * b)\ (b * d)$
by (*simp add*: *diff-rat-def add-rat minus-rat*)

theorem *mult-rat*: $b \neq 0 ==> d \neq 0 ==>$
$Fract\ a\ b * Fract\ c\ d = Fract\ (a * c)\ (b * d)$
by (*simp add*: *Fract-def mult-rat-def mult-congruent2 UN-ratrel2*)

theorem *inverse-rat*: $a \neq 0 ==> b \neq 0 ==>$
$inverse\ (Fract\ a\ b) = Fract\ b\ a$
by (*simp add*: *Fract-def inverse-rat-def inverse-congruent UN-ratrel*)

theorem *divide-rat*: $c \neq 0 ==> b \neq 0 ==> d \neq 0 ==>$
$Fract\ a\ b\ /\ Fract\ c\ d = Fract\ (a * d)\ (b * c)$
by (*simp add*: *divide-rat-def inverse-rat mult-rat*)

theorem *le-rat*: $b \neq 0 \Longrightarrow d \neq 0 \Longrightarrow$
$(Fract\ a\ b \leq Fract\ c\ d) = ((a * d) * (b * d) \leq (c * b) * (b * d))$
by (*simp add*: *Fract-def le-rat-def le-congruent2 UN-ratrel2*)

theorem *less-rat*: $b \neq 0 \Longrightarrow d \neq 0 \Longrightarrow$
$(Fract\ a\ b < Fract\ c\ d) = ((a * d) * (b * d) < (c * b) * (b * d))$
by (*simp add*: *less-rat-def le-rat eq-rat order-less-le*)

theorem *abs-rat*: $b \neq 0 \Longrightarrow |Fract\ a\ b| = Fract\ |a|\ |b|$
by (*simp add*: *abs-rat-def minus-rat Zero-rat-def less-rat eq-rat*)
(*auto simp add*: *mult-less-0-iff zero-less-mult-iff order-le-less*
split: *abs-split*)

9.1.5 The ordered field of rational numbers

instance *rat* :: *field*
proof
fix *q r s* :: *rat*
show $(q + r) + s = q + (r + s)$
by (*induct q*, *induct r*, *induct s*)
(*simp add*: *add-rat add-ac mult-ac int-distrib*)
show $q + r = r + q$
by (*induct q*, *induct r*) (*simp add*: *add-rat add-ac mult-ac*)
show $0 + q = q$
by (*induct q*) (*simp add*: *Zero-rat-def add-rat*)
show $(-q) + q = 0$
by (*induct q*) (*simp add*: *Zero-rat-def minus-rat add-rat eq-rat*)
show $q - r = q + (-r)$
by (*induct q*, *induct r*) (*simp add*: *add-rat minus-rat diff-rat*)
show $(q * r) * s = q * (r * s)$
by (*induct q*, *induct r*, *induct s*) (*simp add*: *mult-rat mult-ac*)
show $q * r = r * q$
by (*induct q*, *induct r*) (*simp add*: *mult-rat mult-ac*)
show $1 * q = q$
by (*induct q*) (*simp add*: *One-rat-def mult-rat*)
show $(q + r) * s = q * s + r * s$
by (*induct q*, *induct r*, *induct s*)
(*simp add*: *add-rat mult-rat eq-rat int-distrib*)
show $q \neq 0 \Longrightarrow inverse\ q * q = 1$
by (*induct q*) (*simp add*: *inverse-rat mult-rat One-rat-def Zero-rat-def eq-rat*)
show $q\ /\ r = q * inverse\ r$
by (*simp add*: *divide-rat-def*)
show $0 \neq (1::rat)$
by (*simp add*: *Zero-rat-def One-rat-def eq-rat*)
qed

instance *rat* :: *linorder*
proof
fix *q r s* :: *rat*

```
{
  assume q ≤ r and r ≤ s
  show q ≤ s
  proof (insert prems, induct q, induct r, induct s)
    fix a b c d e f :: int
    assume neq: b ≠ 0  d ≠ 0  f ≠ 0
    assume 1: Fract a b ≤ Fract c d and 2: Fract c d ≤ Fract e f
    show Fract a b ≤ Fract e f
    proof −
      from neq obtain bb: 0 < b * b and dd: 0 < d * d and ff: 0 < f * f
        by (auto simp add: zero-less-mult-iff linorder-neq-iff)
      have (a * d) * (b * d) * (f * f) ≤ (c * b) * (b * d) * (f * f)
      proof −
        from neq 1 have (a * d) * (b * d) ≤ (c * b) * (b * d)
          by (simp add: le-rat)
        with ff show ?thesis by (simp add: mult-le-cancel-right)
      qed
      also have ... = (c * f) * (d * f) * (b * b)
        by (simp only: mult-ac)
      also have ... ≤ (e * d) * (d * f) * (b * b)
      proof −
        from neq 2 have (c * f) * (d * f) ≤ (e * d) * (d * f)
          by (simp add: le-rat)
        with bb show ?thesis by (simp add: mult-le-cancel-right)
      qed
      finally have (a * f) * (b * f) * (d * d) ≤ e * b * (b * f) * (d * d)
        by (simp only: mult-ac)
      with dd have (a * f) * (b * f) ≤ (e * b) * (b * f)
        by (simp add: mult-le-cancel-right)
      with neq show ?thesis by (simp add: le-rat)
    qed
  qed
next
  assume q ≤ r and r ≤ q
  show q = r
  proof (insert prems, induct q, induct r)
    fix a b c d :: int
    assume neq: b ≠ 0  d ≠ 0
    assume 1: Fract a b ≤ Fract c d and 2: Fract c d ≤ Fract a b
    show Fract a b = Fract c d
    proof −
      from neq 1 have (a * d) * (b * d) ≤ (c * b) * (b * d)
        by (simp add: le-rat)
      also have ... ≤ (a * d) * (b * d)
      proof −
        from neq 2 have (c * b) * (d * b) ≤ (a * d) * (d * b)
          by (simp add: le-rat)
        thus ?thesis by (simp only: mult-ac)
      qed
```

```
      finally have (a * d) * (b * d) = (c * b) * (b * d) .
      moreover from neq have b * d ≠ 0 by simp
      ultimately have a * d = c * b by simp
      with neq show ?thesis by (simp add: eq-rat)
    qed
  qed
 next
  show q ≤ q
    by (induct q) (simp add: le-rat)
  show (q < r) = (q ≤ r ∧ q ≠ r)
    by (simp only: less-rat-def)
  show q ≤ r ∨ r ≤ q
    by (induct q, induct r)
      (simp add: le-rat mult-commute, rule linorder-linear)
 }
qed

instance rat :: distrib-lattice
  inf r s ≡ min r s
  sup r s ≡ max r s
  by default (auto simp add: min-max.sup-inf-distrib1 inf-rat-def sup-rat-def)

instance rat :: ordered-field
proof
  fix q r s :: rat
  show q ≤ r ==> s + q ≤ s + r
  proof (induct q, induct r, induct s)
    fix a b c d e f :: int
    assume neq: b ≠ 0  d ≠ 0  f ≠ 0
    assume le: Fract a b ≤ Fract c d
    show Fract e f + Fract a b ≤ Fract e f + Fract c d
    proof -
      let ?F = f * f from neq have F: 0 < ?F
        by (auto simp add: zero-less-mult-iff)
      from neq le have (a * d) * (b * d) ≤ (c * b) * (b * d)
        by (simp add: le-rat)
      with F have (a * d) * (b * d) * ?F * ?F ≤ (c * b) * (b * d) * ?F * ?F
        by (simp add: mult-le-cancel-right)
      with neq show ?thesis by (simp add: add-rat le-rat mult-ac int-distrib)
    qed
  qed
  show q < r ==> 0 < s ==> s * q < s * r
  proof (induct q, induct r, induct s)
    fix a b c d e f :: int
    assume neq: b ≠ 0  d ≠ 0  f ≠ 0
    assume le: Fract a b < Fract c d
    assume gt: 0 < Fract e f
    show Fract e f * Fract a b < Fract e f * Fract c d
    proof -
```

```
    let ?E = e * f and ?F = f * f
    from neq gt have 0 < ?E
      by (auto simp add: Zero-rat-def less-rat le-rat order-less-le eq-rat)
    moreover from neq have 0 < ?F
      by (auto simp add: zero-less-mult-iff)
    moreover from neq le have (a * d) * (b * d) < (c * b) * (b * d)
      by (simp add: less-rat)
   ultimately have (a * d) * (b * d) * ?E * ?F < (c * b) * (b * d) * ?E * ?F
      by (simp add: mult-less-cancel-right)
    with neq show ?thesis
      by (simp add: less-rat mult-rat mult-ac)
  qed
 qed
 show |q| = (if q < 0 then -q else q)
  by (simp only: abs-rat-def)
qed (auto simp: sgn-rat-def)

instance rat :: division-by-zero
proof
 show inverse 0 = (0::rat)
  by (simp add: Zero-rat-def Fract-def inverse-rat-def
                inverse-congruent UN-ratrel)
qed

instance rat :: recpower
proof
 fix q :: rat
 fix n :: nat
 show q ^ 0 = 1 by simp
 show q ^ (Suc n) = q * (q ^ n) by simp
qed
```

9.2 Various Other Results

```
lemma minus-rat-cancel [simp]: b ≠ 0 ==> Fract (-a) (-b) = Fract a b
by (simp add: eq-rat)

theorem Rat-induct-pos [case-names Fract, induct type: rat]:
 assumes step: !!a b. 0 < b ==> P (Fract a b)
  shows P q
proof (cases q)
 have step': !!a b. b < 0 ==> P (Fract a b)
 proof -
  fix a::int and b::int
  assume b: b < 0
  hence 0 < -b by simp
  hence P (Fract (-a) (-b)) by (rule step)
  thus P (Fract a b) by (simp add: order-less-imp-not-eq [OF b])
 qed
```

case (*Fract a b*)
thus *P q* **by** (*force simp add*: *linorder-neq-iff step step'*)
qed

lemma *zero-less-Fract-iff*:
0 < b ==> (0 < Fract a b) = (0 < a)
by (*simp add*: *Zero-rat-def less-rat order-less-imp-not-eq2 zero-less-mult-iff*)

lemma *Fract-add-one*: *n ≠ 0 ==> Fract (m + n) n = Fract m n + 1*
apply (*insert add-rat* [*of* **concl**: *m n 1 1*])
apply (*simp add*: *One-rat-def* [*symmetric*])
done

lemma *of-nat-rat*: *of-nat k = Fract (of-nat k) 1*
by (*induct k*) (*simp-all add*: *Zero-rat-def One-rat-def add-rat*)

lemma *of-int-rat*: *of-int k = Fract k 1*
by (*cases k rule*: *int-diff-cases*, *simp add*: *of-nat-rat diff-rat*)

lemma *Fract-of-nat-eq*: *Fract (of-nat k) 1 = of-nat k*
by (*rule of-nat-rat* [*symmetric*])

lemma *Fract-of-int-eq*: *Fract k 1 = of-int k*
by (*rule of-int-rat* [*symmetric*])

lemma *Fract-of-int-quotient*: *Fract k l = (if l = 0 then Fract 1 0 else of-int k / of-int l)*
by (*auto simp add*: *Fract-zero Fract-of-int-eq* [*symmetric*] *divide-rat*)

9.3 Numerals and Arithmetic

instance *rat* :: *number*
rat-number-of-def: (*number-of w* :: *rat*) ≡ *of-int w* **..**

instance *rat* :: *number-ring*
by *default* (*simp add*: *rat-number-of-def*)

use *rat-arith.ML*
declaration ⟪ *K rat-arith-setup* ⟫

9.4 Embedding from Rationals to other Fields

class *field-char-0 = field + ring-char-0*

instance *ordered-field < field-char-0* **..**

definition
of-rat :: *rat* ⇒ *'a*::*field-char-0*
where
[*code func del*]: *of-rat q = contents* (⋃(*a*,*b*) ∈ *Rep-Rat q*. {*of-int a / of-int b*})

lemma *of-rat-congruent*:
(λ(*a*, *b*). {*of-int a* / *of-int b*::'*a*::*field-char-0*}) *respects ratrel*
apply (*rule congruent.intro*)
apply (*clarsimp simp add*: *nonzero-divide-eq-eq nonzero-eq-divide-eq*)
apply (*simp only*: *of-int-mult* [*symmetric*])
done

lemma *of-rat-rat*:
b ≠ *0* ⟹ *of-rat* (*Fract a b*) = *of-int a* / *of-int b*
unfolding *Fract-def of-rat-def*
by (*simp add*: *UN-ratrel of-rat-congruent*)

lemma *of-rat-0* [*simp*]: *of-rat 0* = *0*
by (*simp add*: *Zero-rat-def of-rat-rat*)

lemma *of-rat-1* [*simp*]: *of-rat 1* = *1*
by (*simp add*: *One-rat-def of-rat-rat*)

lemma *of-rat-add*: *of-rat* (*a* + *b*) = *of-rat a* + *of-rat b*
by (*induct a*, *induct b*, *simp add*: *add-rat of-rat-rat add-frac-eq*)

lemma *of-rat-minus*: *of-rat* (− *a*) = − *of-rat a*
by (*induct a*, *simp add*: *minus-rat of-rat-rat*)

lemma *of-rat-diff*: *of-rat* (*a* − *b*) = *of-rat a* − *of-rat b*
by (*simp only*: *diff-minus of-rat-add of-rat-minus*)

lemma *of-rat-mult*: *of-rat* (*a* ∗ *b*) = *of-rat a* ∗ *of-rat b*
apply (*induct a*, *induct b*, *simp add*: *mult-rat of-rat-rat*)
apply (*simp add*: *divide-inverse nonzero-inverse-mult-distrib mult-ac*)
done

lemma *nonzero-of-rat-inverse*:
a ≠ *0* ⟹ *of-rat* (*inverse a*) = *inverse* (*of-rat a*)
apply (*rule inverse-unique* [*symmetric*])
apply (*simp add*: *of-rat-mult* [*symmetric*])
done

lemma *of-rat-inverse*:
(*of-rat* (*inverse a*)::'*a*::{*field-char-0*,*division-by-zero*}) =
inverse (*of-rat a*)
by (*cases a* = *0*, *simp-all add*: *nonzero-of-rat-inverse*)

lemma *nonzero-of-rat-divide*:
b ≠ *0* ⟹ *of-rat* (*a* / *b*) = *of-rat a* / *of-rat b*
by (*simp add*: *divide-inverse of-rat-mult nonzero-of-rat-inverse*)

lemma *of-rat-divide*:

(*of-rat* (*a* / *b*)::'*a*::{*field-char-0*,*division-by-zero*})
= *of-rat a* / *of-rat b*
by (*cases b* = *0*, *simp-all add*: *nonzero-of-rat-divide*)

lemma *of-rat-power*:
(*of-rat* (*a* ^ *n*)::'*a*::{*field-char-0*,*recpower*}) = *of-rat a* ^ *n*
by (*induct n*) (*simp-all add*: *of-rat-mult power-Suc*)

lemma *of-rat-eq-iff* [*simp*]: (*of-rat a* = *of-rat b*) = (*a* = *b*)
apply (*induct a*, *induct b*)
apply (*simp add*: *of-rat-rat eq-rat*)
apply (*simp add*: *nonzero-divide-eq-eq nonzero-eq-divide-eq*)
apply (*simp only*: *of-int-mult* [*symmetric*] *of-int-eq-iff*)
done

lemmas *of-rat-eq-0-iff* [*simp*] = *of-rat-eq-iff* [*of - 0*, *simplified*]

lemma *of-rat-eq-id* [*simp*]: *of-rat* = (*id* :: *rat* ⇒ *rat*)
proof
fix *a*
show *of-rat a* = *id a*
by (*induct a*)
(*simp add*: *of-rat-rat divide-rat Fract-of-int-eq* [*symmetric*])
qed

Collapse nested embeddings

lemma *of-rat-of-nat-eq* [*simp*]: *of-rat* (*of-nat n*) = *of-nat n*
by (*induct n*) (*simp-all add*: *of-rat-add*)

lemma *of-rat-of-int-eq* [*simp*]: *of-rat* (*of-int z*) = *of-int z*
by (*cases z rule*: *int-diff-cases*, *simp add*: *of-rat-diff*)

lemma *of-rat-number-of-eq* [*simp*]:
of-rat (*number-of w*) = (*number-of w* :: '*a*::{*number-ring*,*field-char-0*})
by (*simp add*: *number-of-eq*)

lemmas *zero-rat* = *Zero-rat-def*
lemmas *one-rat* = *One-rat-def*

abbreviation
rat-of-nat :: *nat* ⇒ *rat*
where
rat-of-nat ≡ *of-nat*

abbreviation
rat-of-int :: *int* ⇒ *rat*
where
rat-of-int ≡ *of-int*

9.5 Implementation of rational numbers as pairs of integers

definition
Rational :: *int* × *int* ⇒ *rat*
where
Rational = *INum*

code-datatype *Rational*

lemma *Rational-simp*:
Rational (*k*, *l*) = *rat-of-int k* / *rat-of-int l*
unfolding *Rational-def INum-def* **by** *simp*

lemma *Rational-zero* [*simp*]: *Rational* 0_N = *0*
by (*simp add*: *Rational-simp*)

lemma *Rational-lit* [*simp*]: *Rational* i_N = *rat-of-int i*
by (*simp add*: *Rational-simp*)

lemma *zero-rat-code* [*code*, *code unfold*]:
0 = *Rational* 0_N **by** *simp*

lemma *zero-rat-code* [*code*, *code unfold*]:
1 = *Rational* 1_N **by** *simp*

lemma [*code*, *code unfold*]:
number-of k = *rat-of-int* (*number-of k*)
by (*simp add*: *number-of-is-id rat-number-of-def*)

definition
[*code func del*]: *Fract'* (*b*::*bool*) *k l* = *Fract k l*

lemma [*code*]:
Fract k l = *Fract'* (*l* ≠ *0*) *k l*
unfolding *Fract'-def* **..**

lemma [*code*]:
Fract' True k l = (*if l* ≠ *0 then Rational* (*k*, *l*) *else Fract 1 0*)
by (*simp add*: *Fract'-def Rational-simp Fract-of-int-quotient* [*of k l*])

lemma [*code*]:
of-rat (*Rational* (*k*, *l*)) = (*if l* ≠ *0 then of-int k* / *of-int l else 0*)
by (*cases l* = *0*)
(*auto simp add*: *Rational-simp of-rat-rat* [*simplified Fract-of-int-quotient* [*of k l*], *symmetric*])

instance *rat* :: *eq* **..**

lemma *rat-eq-code* [*code*]: *Rational x* = *Rational y* ⟷ *normNum x* = *normNum y*

unfolding *Rational-def INum-normNum-iff* **..**

lemma *rat-less-eq-code* [*code*]: *Rational x* $\leq$ *Rational y* $\longleftrightarrow$ *normNum x* $\leq_N$ *normNum y*
proof −
have *normNum x* $\leq_N$ *normNum y* $\longleftrightarrow$ *Rational* (*normNum x*) $\leq$ *Rational* (*normNum y*)
by (*simp add*: *Rational-def del*: *normNum*)
also have ... = (*Rational x* $\leq$ *Rational y*) **by** (*simp add*: *Rational-def*)
finally show *?thesis* **by** *simp*
qed

lemma *rat-less-code* [*code*]: *Rational x* $<$ *Rational y* $\longleftrightarrow$ *normNum x* $<_N$ *normNum y*
proof −
have *normNum x* $<_N$ *normNum y* $\longleftrightarrow$ *Rational* (*normNum x*) $<$ *Rational* (*normNum y*)
by (*simp add*: *Rational-def del*: *normNum*)
also have ... = (*Rational x* $<$ *Rational y*) **by** (*simp add*: *Rational-def*)
finally show *?thesis* **by** *simp*
qed

lemma *rat-add-code* [*code*]: *Rational x* + *Rational y* = *Rational* (*x* $+_N$ *y*)
unfolding *Rational-def* **by** *simp*

lemma *rat-mul-code* [*code*]: *Rational x* $*$ *Rational y* = *Rational* (*x* $*_N$ *y*)
unfolding *Rational-def* **by** *simp*

lemma *rat-neg-code* [*code*]: − *Rational x* = *Rational* ($^\sim_N$ *x*)
unfolding *Rational-def* **by** *simp*

lemma *rat-sub-code* [*code*]: *Rational x* − *Rational y* = *Rational* (*x* $-_N$ *y*)
unfolding *Rational-def* **by** *simp*

lemma *rat-inv-code* [*code*]: *inverse* (*Rational x*) = *Rational* (*Ninv x*)
unfolding *Rational-def Ninv divide-rat-def* **by** *simp*

lemma *rat-div-code* [*code*]: *Rational x* / *Rational y* = *Rational* (*x* $\div_N$ *y*)
unfolding *Rational-def* **by** *simp*

Setup for SML code generator

types-code
rat ((*int* */ int*))
attach (*term-of*) ⟪
fun term-of-rat (*p*, *q*) =
let
val rT = *Type* (*Rational.rat*, [])
in
if q = *1 orelse p* = *0 then HOLogic.mk-number rT p*

```
    else Const (HOL.inverse-class.divide, rT --> rT --> rT) $
      HOLogic.mk-number rT p $ HOLogic.mk-number rT q
  end;
⟩⟩
attach (test) ⟨⟨
fun gen-rat i =
  let
    val p = random-range 0 i;
    val q = random-range 1 (i + 1);
    val g = Integer.gcd p q;
    val p' = p div g;
    val q' = q div g;
  in
    (if one-of [true, false] then p' else ~ p',
     if p' = 0 then 0 else q')
  end;
⟩⟩

consts-code
  Rational ((-))

consts-code
  of-int :: int ⇒ rat (⟨module⟩rat'-of'-int)
attach ⟨⟨
fun rat-of-int 0 = (0, 0)
  | rat-of-int i = (i, 1);
⟩⟩

end
```

10 Positive real numbers

```
theory PReal
imports Rational
begin
```

Could be generalized and moved to *Ring-and-Field*

```
lemma add-eq-exists: ∃x. a+x = (b::rat)
by (rule-tac x=b−a in exI, simp)

definition
  cut :: rat set => bool where
  cut A = ({} ⊂ A &
            A < {r. 0 < r} &
            (∀y ∈ A. ((∀z. 0<z & z < y --> z ∈ A) & (∃u ∈ A. y < u))))

lemma cut-of-rat:
  assumes q: 0 < q shows cut {r::rat. 0 < r & r < q} (is cut ?A)
```

proof −
from *q* **have** *pos*: *?A* < {*r*. *0* < *r*} **by** *force*
have *nonempty*: {} ⊂ *?A*
proof
show {} ⊆ *?A* **by** *simp*
show {} ≠ *?A*
by (*force simp only*: *q eq-commute* [*of* {}] *interval-empty-iff*)
qed
show *?thesis*
by (*simp add*: *cut-def pos nonempty*,
blast dest: *dense intro*: *order-less-trans*)
qed

typedef *preal* = {*A*. *cut A*}
by (*blast intro*: *cut-of-rat* [*OF zero-less-one*])

instance *preal* :: {*ord*, *plus*, *minus*, *times*, *inverse*, *one*} **..**

definition
preal-of-rat :: *rat* => *preal* **where**
preal-of-rat q = *Abs-preal* {*x*::*rat*. *0* < *x* & *x* < *q*}

definition
psup :: *preal set* => *preal* **where**
psup P = *Abs-preal* (⋃ *X* ∈ *P*. *Rep-preal X*)

definition
add-set :: [*rat set*,*rat set*] => *rat set* **where**
add-set A B = {*w*. ∃ *x* ∈ *A*. ∃ *y* ∈ *B*. *w* = *x* + *y*}

definition
diff-set :: [*rat set*,*rat set*] => *rat set* **where**
diff-set A B = {*w*. ∃ *x*. *0* < *w* & *0* < *x* & *x* ∉ *B* & *x* + *w* ∈ *A*}

definition
mult-set :: [*rat set*,*rat set*] => *rat set* **where**
mult-set A B = {*w*. ∃ *x* ∈ *A*. ∃ *y* ∈ *B*. *w* = *x* ∗ *y*}

definition
inverse-set :: *rat set* => *rat set* **where**
inverse-set A = {*x*. ∃ *y*. *0* < *x* & *x* < *y* & *inverse y* ∉ *A*}

defs (**overloaded**)

preal-less-def:
R < *S* == *Rep-preal R* < *Rep-preal S*

preal-le-def:
$R \leq S == \mathit{Rep\text{-}preal}\ R \subseteq \mathit{Rep\text{-}preal}\ S$

preal-add-def:
$R + S == \mathit{Abs\text{-}preal}\ (\mathit{add\text{-}set}\ (\mathit{Rep\text{-}preal}\ R)\ (\mathit{Rep\text{-}preal}\ S))$

preal-diff-def:
$R - S == \mathit{Abs\text{-}preal}\ (\mathit{diff\text{-}set}\ (\mathit{Rep\text{-}preal}\ R)\ (\mathit{Rep\text{-}preal}\ S))$

preal-mult-def:
$R * S == \mathit{Abs\text{-}preal}\ (\mathit{mult\text{-}set}\ (\mathit{Rep\text{-}preal}\ R)\ (\mathit{Rep\text{-}preal}\ S))$

preal-inverse-def:
$\mathit{inverse}\ R == \mathit{Abs\text{-}preal}\ (\mathit{inverse\text{-}set}\ (\mathit{Rep\text{-}preal}\ R))$

preal-one-def:
$1 == \mathit{preal\text{-}of\text{-}rat}\ 1$

Reduces equality on abstractions to equality on representatives

declare *Abs-preal-inject* [*simp*]
declare *Abs-preal-inverse* [*simp*]

lemma *rat-mem-preal*: $0 < q \Longrightarrow \{r{::}\mathit{rat}.\ 0 < r \ \&\ r < q\} \in \mathit{preal}$
by (*simp add*: *preal-def cut-of-rat*)

lemma *preal-nonempty*: $A \in \mathit{preal} \Longrightarrow \exists x{\in}A.\ 0 < x$
by (*unfold preal-def cut-def*, *blast*)

lemma *preal-Ex-mem*: $A \in \mathit{preal} \Longrightarrow \exists x.\ x \in A$
by (*drule preal-nonempty*, *fast*)

lemma *preal-imp-psubset-positives*: $A \in \mathit{preal} \Longrightarrow A < \{r.\ 0 < r\}$
by (*force simp add*: *preal-def cut-def*)

lemma *preal-exists-bound*: $A \in \mathit{preal} \Longrightarrow \exists x.\ 0 < x \ \&\ x \notin A$
by (*drule preal-imp-psubset-positives*, *auto*)

lemma *preal-exists-greater*: $[\!|\ A \in \mathit{preal};\ y \in A\ |\!] \Longrightarrow \exists u \in A.\ y < u$
by (*unfold preal-def cut-def*, *blast*)

lemma *preal-downwards-closed*: $[\!|\ A \in \mathit{preal};\ y \in A;\ 0 < z;\ z < y\ |\!] \Longrightarrow z \in A$
by (*unfold preal-def cut-def*, *blast*)

Relaxing the final premise

lemma *preal-downwards-closed′*:
$[\!|\ A \in \mathit{preal};\ y \in A;\ 0 < z;\ z \leq y\ |\!] \Longrightarrow z \in A$
apply (*simp add*: *order-le-less*)
apply (*blast intro*: *preal-downwards-closed*)
done

A positive fraction not in a positive real is an upper bound. Gleason p. 122 - Remark (1)

lemma *not-in-preal-ub*:
  assumes *A*: $A \in preal$
    and *notx*: $x \notin A$
    and *y*: $y \in A$
    and *pos*: $0 < x$
  shows $y < x$
proof (*cases rule*: *linorder-cases*)
  assume $x<y$
  with *notx* **show** *?thesis*
    by (*simp add*: *preal-downwards-closed* [*OF A y*] *pos*)
next
  assume $x=y$
  with *notx* **and** *y* **show** *?thesis* **by** *simp*
next
  assume $y<x$
  thus *?thesis* **.**
qed

preal lemmas instantiated to *Rep-preal X*

lemma *mem-Rep-preal-Ex*: $\exists x.\ x \in Rep\text{-}preal\ X$
by (*rule preal-Ex-mem* [*OF Rep-preal*])

lemma *Rep-preal-exists-bound*: $\exists x>0.\ x \notin Rep\text{-}preal\ X$
by (*rule preal-exists-bound* [*OF Rep-preal*])

lemmas *not-in-Rep-preal-ub* = *not-in-preal-ub* [*OF Rep-preal*]

10.1 *preal-of-prat*: the Injection from prat to preal

lemma *rat-less-set-mem-preal*: $0 < y \Longrightarrow \{u::rat.\ 0 < u \ \&\ u < y\} \in preal$
by (*simp add*: *preal-def cut-of-rat*)

lemma *rat-subset-imp-le*:
    $[|\{u::rat.\ 0 < u \ \&\ u < x\} \subseteq \{u.\ 0 < u \ \&\ u < y\};\ 0<x|] \Longrightarrow x \leq y$
apply (*simp add*: *linorder-not-less* [*symmetric*])
apply (*blast dest*: *dense intro*: *order-less-trans*)
done

lemma *rat-set-eq-imp-eq*:
    $[|\{u::rat.\ 0 < u \ \&\ u < x\} = \{u.\ 0 < u \ \&\ u < y\};$
      $0 < x;\ 0 < y|] \Longrightarrow x = y$
by (*blast intro*: *rat-subset-imp-le order-antisym*)

10.2 Properties of Ordering

lemma *preal-le-refl*: $w \leq (w::preal)$
by (*simp add*: *preal-le-def*)

lemma *preal-le-trans*: [| $i \leq j$; $j \leq k$ |] ==> $i \leq (k::preal)$
by (*force simp add*: *preal-le-def*)

lemma *preal-le-anti-sym*: [| $z \leq w$; $w \leq z$ |] ==> $z = (w::preal)$
apply (*simp add*: *preal-le-def*)
apply (*rule Rep-preal-inject* [*THEN iffD1*], *blast*)
done

lemma *preal-less-le*: $((w::preal) < z) = (w \leq z \;\&\; w \neq z)$
by (*simp add*: *preal-le-def preal-less-def Rep-preal-inject psubset-def*)

instance *preal* :: *order*
  by *intro-classes*
    (*assumption* |
      rule preal-le-refl preal-le-trans preal-le-anti-sym preal-less-le)+

lemma *preal-imp-pos*: [|$A \in preal$; $r \in A$|] ==> $0 < r$
by (*insert preal-imp-psubset-positives*, *blast*)

lemma *preal-le-linear*: $x <= y \mid y <= (x::preal)$
apply (*auto simp add*: *preal-le-def*)
apply (*rule ccontr*)
apply (*blast dest*: *not-in-Rep-preal-ub intro*: *preal-imp-pos* [*OF Rep-preal*]
        elim: *order-less-asym*)
done

instance *preal* :: *linorder*
  by *intro-classes* (*rule preal-le-linear*)

instance *preal* :: *distrib-lattice*
  $inf \equiv min$
  $sup \equiv max$
  by *intro-classes*
    (*auto simp add*: *inf-preal-def sup-preal-def min-max.sup-inf-distrib1*)

10.3 Properties of Addition

lemma *preal-add-commute*: $(x::preal) + y = y + x$
apply (*unfold preal-add-def add-set-def*)
apply (*rule-tac f = Abs-preal* **in** *arg-cong*)
apply (*force simp add*: *add-commute*)
done

Lemmas for proving that addition of two positive reals gives a positive real

lemma *empty-psubset-nonempty*: $a \in A$ ==> $\{\} \subset A$
by *blast*

Part 1 of Dedekind sections definition

lemma *add-set-not-empty*:
[|$A \in$ *preal*; $B \in$ *preal*|] ==> $\{\} \subset$ *add-set* A B
apply (*drule preal-nonempty*)+
apply (*auto simp add*: *add-set-def*)
done

Part 2 of Dedekind sections definition. A structured version of this proof is *preal-not-mem-mult-set-Ex* below.

lemma *preal-not-mem-add-set-Ex*:
[|$A \in$ *preal*; $B \in$ *preal*|] ==> $\exists\, q{>}0.\ q \notin$ *add-set* A B
apply (*insert preal-exists-bound* [*of A*] *preal-exists-bound* [*of B*], *auto*)
apply (*rule-tac* $x = x{+}xa$ **in** *exI*)
apply (*simp add*: *add-set-def*, *clarify*)
apply (*drule* (*3*) *not-in-preal-ub*)+
apply (*force dest*: *add-strict-mono*)
done

lemma *add-set-not-rat-set*:
assumes *A*: $A \in$ *preal*
and *B*: $B \in$ *preal*
shows *add-set* A $B < \{r.\ 0 < r\}$
proof
from *preal-imp-pos* [*OF A*] *preal-imp-pos* [*OF B*]
show *add-set* A $B \subseteq \{r.\ 0 < r\}$ **by** (*force simp add*: *add-set-def*)
next
show *add-set* A $B \neq \{r.\ 0 < r\}$
by (*insert preal-not-mem-add-set-Ex* [*OF A B*], *blast*)
qed

Part 3 of Dedekind sections definition

lemma *add-set-lemma3*:
[|$A \in$ *preal*; $B \in$ *preal*; $u \in$ *add-set* A B; $0 < z$; $z < u$|]
==> $z \in$ *add-set* A B
proof (*unfold add-set-def*, *clarify*)
fix x::*rat* **and** y::*rat*
assume *A*: $A \in$ *preal*
and *B*: $B \in$ *preal*
and [*simp*]: $0 < z$
and *zless*: $z < x + y$
and x: $x \in A$
and y: $y \in B$
have *xpos* [*simp*]: $0{<}x$ **by** (*rule preal-imp-pos* [*OF A x*])
have *ypos* [*simp*]: $0{<}y$ **by** (*rule preal-imp-pos* [*OF B y*])
have *xypos* [*simp*]: $0 < x{+}y$ **by** (*simp add*: *pos-add-strict*)
let *?f* $= z/(x{+}y)$
have *fless*: *?f* < 1 **by** (*simp add*: *zless pos-divide-less-eq*)
show $\exists\, x' \in A.\ \exists\, y'{\in}B.\ z = x' + y'$
proof (*intro bexI*)
show $z = x{*}$*?f* $+\ y{*}$*?f*

```
    by (simp add: left-distrib [symmetric] divide-inverse mult-ac
        order-less-imp-not-eq2)
  next
    show y * ?f ∈ B
    proof (rule preal-downwards-closed [OF B y])
      show 0 < y * ?f
        by (simp add: divide-inverse zero-less-mult-iff)
    next
      show y * ?f < y
        by (insert mult-strict-left-mono [OF fless ypos], simp)
    qed
  next
    show x * ?f ∈ A
    proof (rule preal-downwards-closed [OF A x])
      show 0 < x * ?f
        by (simp add: divide-inverse zero-less-mult-iff)
    next
      show x * ?f < x
        by (insert mult-strict-left-mono [OF fless xpos], simp)
    qed
  qed
qed
```

Part 4 of Dedekind sections definition

```
lemma add-set-lemma4:
    [|A ∈ preal; B ∈ preal; y ∈ add-set A B|] ==> ∃u ∈ add-set A B. y < u
apply (auto simp add: add-set-def)
apply (frule preal-exists-greater [of A], auto)
apply (rule-tac x=u + y in exI)
apply (auto intro: add-strict-left-mono)
done

lemma mem-add-set:
    [|A ∈ preal; B ∈ preal|] ==> add-set A B ∈ preal
apply (simp (no-asm-simp) add: preal-def cut-def)
apply (blast intro!: add-set-not-empty add-set-not-rat-set
                     add-set-lemma3 add-set-lemma4)
done

lemma preal-add-assoc: ((x::preal) + y) + z = x + (y + z)
apply (simp add: preal-add-def mem-add-set Rep-preal)
apply (force simp add: add-set-def add-ac)
done

instance preal :: ab-semigroup-add
proof
  fix a b c :: preal
  show (a + b) + c = a + (b + c) by (rule preal-add-assoc)
  show a + b = b + a by (rule preal-add-commute)
```

qed

lemma *preal-add-left-commute*: $x + (y + z) = y + ((x + z)::preal)$
by (*rule add-left-commute*)

Positive Real addition is an AC operator

lemmas *preal-add-ac = preal-add-assoc preal-add-commute preal-add-left-commute*

10.4 Properties of Multiplication

Proofs essentially same as for addition

lemma *preal-mult-commute*: $(x::preal) * y = y * x$
apply (*unfold preal-mult-def mult-set-def*)
apply (*rule-tac f = Abs-preal* **in** *arg-cong*)
apply (*force simp add*: *mult-commute*)
done

Multiplication of two positive reals gives a positive real.

Lemmas for proving positive reals multiplication set in *preal*

Part 1 of Dedekind sections definition

lemma *mult-set-not-empty*:
    $[|A \in preal; B \in preal|] ==> \{\} \subset mult\text{-}set\ A\ B$
apply (*insert preal-nonempty* [*of A*] *preal-nonempty* [*of B*])
apply (*auto simp add*: *mult-set-def*)
done

Part 2 of Dedekind sections definition

lemma *preal-not-mem-mult-set-Ex*:
  assumes *A*: $A \in preal$
    and *B*: $B \in preal$
  shows $\exists q.\ 0 < q\ \&\ q \notin mult\text{-}set\ A\ B$
proof −
 from *preal-exists-bound* [*OF A*]
 obtain *x* **where** [*simp*]: $0 < x\ x \notin A$ **by** *blast*
 from *preal-exists-bound* [*OF B*]
 obtain *y* **where** [*simp*]: $0 < y\ y \notin B$ **by** *blast*
 show *?thesis*
 proof (*intro exI conjI*)
  show $0 < x*y$ **by** (*simp add*: *mult-pos-pos*)
  show $x * y \notin mult\text{-}set\ A\ B$
  proof −
   { fix *u*::*rat* **and** *v*::*rat*
      assume $u \in A$ **and** $v \in B$ **and** $x*y = u*v$
      moreover
      with *prems* **have** $u<x$ **and** $v<y$ **by** (*blast dest*: *not-in-preal-ub*)+
      moreover

```
        with prems have 0≤v
          by (blast intro: preal-imp-pos [OF B]  order-less-imp-le prems)
        moreover
      from calculation
        have u*v < x*y by (blast intro: mult-strict-mono prems)
        ultimately have False by force }
    thus ?thesis by (auto simp add: mult-set-def)
  qed
 qed
qed

lemma mult-set-not-rat-set:
 assumes A: A ∈ preal
   and B: B ∈ preal
 shows mult-set A B < {r. 0 < r}
proof
 show mult-set A B ⊆ {r. 0 < r}
   by (force simp add: mult-set-def
     intro: preal-imp-pos [OF A] preal-imp-pos [OF B] mult-pos-pos)
 show mult-set A B ≠ {r. 0 < r}
   using preal-not-mem-mult-set-Ex [OF A B] by blast
qed
```

Part 3 of Dedekind sections definition

```
lemma mult-set-lemma3:
    [|A ∈ preal; B ∈ preal; u ∈ mult-set A B; 0 < z; z < u|]
     ==> z ∈ mult-set A B
proof (unfold mult-set-def, clarify)
 fix x::rat and y::rat
 assume A: A ∈ preal
   and B: B ∈ preal
   and [simp]: 0 < z
   and zless: z < x * y
   and x:  x ∈ A
   and y:  y ∈ B
 have [simp]: 0<y by (rule preal-imp-pos [OF B y])
 show ∃x' ∈ A. ∃y' ∈ B. z = x' * y'
 proof
   show ∃y'∈B. z = (z/y) * y'
   proof
     show z = (z/y)*y
       by (simp add: divide-inverse mult-commute [of y] mult-assoc
                order-less-imp-not-eq2)
     show y ∈ B by fact
   qed
 next
   show z/y ∈ A
   proof (rule preal-downwards-closed [OF A x])
     show 0 < z/y
```

```
      by (simp add: zero-less-divide-iff)
    show z/y < x by (simp add: pos-divide-less-eq zless)
  qed
 qed
qed
```

Part 4 of Dedekind sections definition

```
lemma mult-set-lemma4:
    [|A ∈ preal; B ∈ preal; y ∈ mult-set A B|] ==> ∃ u ∈ mult-set A B. y < u
apply (auto simp add: mult-set-def)
apply (frule preal-exists-greater [of A], auto)
apply (rule-tac x=u * y in exI)
apply (auto intro: preal-imp-pos [of A] preal-imp-pos [of B]
                   mult-strict-right-mono)
done


lemma mem-mult-set:
    [|A ∈ preal; B ∈ preal|] ==> mult-set A B ∈ preal
apply (simp (no-asm-simp) add: preal-def cut-def)
apply (blast intro!: mult-set-not-empty mult-set-not-rat-set
                     mult-set-lemma3 mult-set-lemma4)
done

lemma preal-mult-assoc: ((x::preal) * y) * z = x * (y * z)
apply (simp add: preal-mult-def mem-mult-set Rep-preal)
apply (force simp add: mult-set-def mult-ac)
done

instance preal :: ab-semigroup-mult
proof
  fix a b c :: preal
  show (a * b) * c = a * (b * c) by (rule preal-mult-assoc)
  show a * b = b * a by (rule preal-mult-commute)
qed

lemma preal-mult-left-commute: x * (y * z) = y * ((x * z)::preal)
by (rule mult-left-commute)
```

Positive Real multiplication is an AC operator

```
lemmas preal-mult-ac =
       preal-mult-assoc preal-mult-commute preal-mult-left-commute
```

Positive real 1 is the multiplicative identity element

```
lemma preal-mult-1: (1::preal) * z = z
unfolding preal-one-def
proof (induct z)
  fix A :: rat set
  assume A: A ∈ preal
```

```
  have {w. ∃ u. 0 < u ∧ u < 1 & (∃ v ∈ A. w = u * v)} = A (is ?lhs = A)
  proof
    show ?lhs ⊆ A
    proof clarify
      fix x::rat and u::rat and v::rat
      assume upos: 0<u and u<1 and v: v ∈ A
      have vpos: 0<v by (rule preal-imp-pos [OF A v])
      hence u*v < 1*v by (simp only: mult-strict-right-mono prems)
      thus u * v ∈ A
        by (force intro: preal-downwards-closed [OF A v] mult-pos-pos
          upos vpos)
    qed
  next
    show A ⊆ ?lhs
    proof clarify
      fix x::rat
      assume x: x ∈ A
      have xpos: 0<x by (rule preal-imp-pos [OF A x])
      from preal-exists-greater [OF A x]
      obtain v where v: v ∈ A and xlessv: x < v ..
      have vpos: 0<v by (rule preal-imp-pos [OF A v])
      show ∃ u. 0 < u ∧ u < 1 ∧ (∃ v∈A. x = u * v)
      proof (intro exI conjI)
        show 0 < x/v
          by (simp add: zero-less-divide-iff xpos vpos)
        show x / v < 1
          by (simp add: pos-divide-less-eq vpos xlessv)
        show ∃ v'∈A. x = (x / v) * v'
        proof
          show x = (x/v)*v
            by (simp add: divide-inverse mult-assoc vpos
                          order-less-imp-not-eq2)
          show v ∈ A by fact
        qed
      qed
    qed
  qed
  thus preal-of-rat 1 * Abs-preal A = Abs-preal A
    by (simp add: preal-of-rat-def preal-mult-def mult-set-def
                  rat-mem-preal A)
qed

instance preal :: comm-monoid-mult
by intro-classes (rule preal-mult-1)

lemma preal-mult-1-right: z * (1::preal) = z
by (rule mult-1-right)
```

10.5 Distribution of Multiplication across Addition

lemma *mem-Rep-preal-add-iff*:
  $(z \in \textit{Rep-preal}(R+S)) = (\exists\, x \in \textit{Rep-preal}\ R.\ \exists\, y \in \textit{Rep-preal}\ S.\ z = x + y)$
apply (*simp add*: *preal-add-def mem-add-set Rep-preal*)
apply (*simp add*: *add-set-def*)
done

lemma *mem-Rep-preal-mult-iff*:
  $(z \in \textit{Rep-preal}(R*S)) = (\exists\, x \in \textit{Rep-preal}\ R.\ \exists\, y \in \textit{Rep-preal}\ S.\ z = x * y)$
apply (*simp add*: *preal-mult-def mem-mult-set Rep-preal*)
apply (*simp add*: *mult-set-def*)
done

lemma *distrib-subset1*:
  $\textit{Rep-preal}\ (w * (x + y)) \subseteq \textit{Rep-preal}\ (w * x + w * y)$
apply (*auto simp add*: *Bex-def mem-Rep-preal-add-iff mem-Rep-preal-mult-iff*)
apply (*force simp add*: *right-distrib*)
done

lemma *preal-add-mult-distrib-mean*:
 assumes *a*: $a \in \textit{Rep-preal}\ w$
  and *b*: $b \in \textit{Rep-preal}\ w$
  and *d*: $d \in \textit{Rep-preal}\ x$
  and *e*: $e \in \textit{Rep-preal}\ y$
 shows $\exists\, c \in \textit{Rep-preal}\ w.\ a * d + b * e = c * (d + e)$
proof
 let *?c* = $(a*d + b*e)/(d+e)$
 have [*simp*]: $0<a\ 0<b\ 0<d\ 0<e\ 0<d+e$
  by (*blast intro*: *preal-imp-pos* [*OF Rep-preal*] *a b d e pos-add-strict*)+
 have *cpos*: $0 < ?c$
  by (*simp add*: *zero-less-divide-iff zero-less-mult-iff pos-add-strict*)
 show $a * d + b * e = ?c * (d + e)$
  by (*simp add*: *divide-inverse mult-assoc order-less-imp-not-eq2*)
 show *?c* $\in \textit{Rep-preal}\ w$
 proof (*cases rule*: *linorder-le-cases*)
  assume $a \leq b$
  hence $?c \leq b$
   by (*simp add*: *pos-divide-le-eq right-distrib mult-right-mono*
       order-less-imp-le)
  thus *?thesis* **by** (*rule preal-downwards-closed'* [*OF Rep-preal b cpos*])
 next
  assume $b \leq a$
  hence $?c \leq a$
   by (*simp add*: *pos-divide-le-eq right-distrib mult-right-mono*
       order-less-imp-le)
  thus *?thesis* **by** (*rule preal-downwards-closed'* [*OF Rep-preal a cpos*])
 qed
qed

lemma *distrib-subset2*:
    Rep-preal $(w * x + w * y) \subseteq$ *Rep-preal* $(w * (x + y))$
apply (*auto simp add*: *Bex-def mem-Rep-preal-add-iff mem-Rep-preal-mult-iff*)
apply (*drule-tac* w=w **and** x=x **and** y=y **in** *preal-add-mult-distrib-mean*, *auto*)
done

lemma *preal-add-mult-distrib2*: $(w * ((x::preal) + y)) = (w * x) + (w * y)$
apply (*rule Rep-preal-inject* [*THEN iffD1*])
apply (*rule equalityI* [*OF distrib-subset1 distrib-subset2*])
done

lemma *preal-add-mult-distrib*: $(((x::preal) + y) * w) = (x * w) + (y * w)$
by (*simp add*: *preal-mult-commute preal-add-mult-distrib2*)

instance *preal* :: *comm-semiring*
by *intro-classes* (*rule preal-add-mult-distrib*)

10.6 Existence of Inverse, a Positive Real

lemma *mem-inv-set-ex*:
  assumes A: $A \in$ *preal* **shows** $\exists x\ y.\ 0 < x\ \&\ x < y\ \&$ *inverse* $y \notin A$
proof −
  from *preal-exists-bound* [*OF A*]
  obtain x **where** [*simp*]: $0<x$ $x \notin A$ **by** *blast*
  show *?thesis*
  proof (*intro exI conjI*)
    show $0 <$ *inverse* $(x+1)$
      by (*simp add*: *order-less-trans* [*OF - less-add-one*])
    show *inverse*$(x+1) <$ *inverse* x
      by (*simp add*: *less-imp-inverse-less less-add-one*)
    show *inverse* (*inverse* x) $\notin A$
      by (*simp add*: *order-less-imp-not-eq2*)
  qed
qed

Part 1 of Dedekind sections definition

lemma *inverse-set-not-empty*:
    $A \in$ *preal* ==> $\{\} \subset$ *inverse-set* A
apply (*insert mem-inv-set-ex* [*of A*])
apply (*auto simp add*: *inverse-set-def*)
done

Part 2 of Dedekind sections definition

lemma *preal-not-mem-inverse-set-Ex*:
   assumes A: $A \in$ *preal* **shows** $\exists q.\ 0 < q\ \&\ q \notin$ *inverse-set* A
proof −
  from *preal-nonempty* [*OF A*]
  obtain x **where** x: $x \in A$ **and** *xpos* [*simp*]: $0<x$ **..**
  show *?thesis*

```
  proof (intro exI conjI)
    show 0 < inverse x by simp
    show inverse x ∉ inverse-set A
    proof −
      { fix y::rat
        assume ygt: inverse x < y
        have [simp]: 0 < y by (simp add: order-less-trans [OF - ygt])
        have iyless: inverse y < x
          by (simp add: inverse-less-imp-less [of x] ygt)
        have inverse y ∈ A
          by (simp add: preal-downwards-closed [OF A x] iyless)}
     thus ?thesis by (auto simp add: inverse-set-def)
    qed
  qed
qed

lemma inverse-set-not-rat-set:
   assumes A: A ∈ preal  shows inverse-set A < {r. 0 < r}
proof
  show inverse-set A ⊆ {r. 0 < r}  by (force simp add: inverse-set-def)
next
  show inverse-set A ≠ {r. 0 < r}
    by (insert preal-not-mem-inverse-set-Ex [OF A], blast)
qed
```

Part 3 of Dedekind sections definition

```
lemma inverse-set-lemma3:
     [|A ∈ preal; u ∈ inverse-set A; 0 < z; z < u|]
      ==> z ∈ inverse-set A
apply (auto simp add: inverse-set-def)
apply (auto intro: order-less-trans)
done
```

Part 4 of Dedekind sections definition

```
lemma inverse-set-lemma4:
     [|A ∈ preal; y ∈ inverse-set A|] ==> ∃ u ∈ inverse-set A. y < u
apply (auto simp add: inverse-set-def)
apply (drule dense [of y])
apply (blast intro: order-less-trans)
done


lemma mem-inverse-set:
     A ∈ preal ==> inverse-set A ∈ preal
apply (simp (no-asm-simp) add: preal-def cut-def)
apply (blast intro!: inverse-set-not-empty inverse-set-not-rat-set
                     inverse-set-lemma3 inverse-set-lemma4)
done
```

10.7 Gleason's Lemma 9-3.4, page 122

lemma *Gleason9-34-exists*:
assumes *A*: $A \in preal$
and $\forall x{\in}A.\ x + u \in A$
and $0 \leq z$
shows $\exists b{\in}A.\ b + (of\text{-}int\ z) * u \in A$
proof (*cases z rule*: *int-cases*)
case (*nonneg n*)
show *?thesis*
proof (*simp add*: *prems*, *induct n*)
case *0*
from *preal-nonempty* [*OF A*]
show *?case* **by** *force*
case (*Suc k*)
from *this* **obtain** *b* **where** $b \in A\ b + of\text{-}nat\ k * u \in A$ **..**
hence $b + of\text{-}int\ (int\ k) * u + u \in A$ **by** (*simp add*: *prems*)
thus *?case* **by** (*force simp add*: *left-distrib add-ac prems*)
qed
next
case (*neg n*)
with *prems* **show** *?thesis* **by** *simp*
qed

lemma *Gleason9-34-contra*:
assumes *A*: $A \in preal$
shows $[|\forall x{\in}A.\ x + u \in A;\ 0 < u;\ 0 < y;\ y \notin A|] ==> False$
proof (*induct u*, *induct y*)
fix *a*::*int* **and** *b*::*int*
fix *c*::*int* **and** *d*::*int*
assume *bpos* [*simp*]: $0 < b$
and *dpos* [*simp*]: $0 < d$
and *closed*: $\forall x{\in}A.\ x + (Fract\ c\ d) \in A$
and *upos*: $0 < Fract\ c\ d$
and *ypos*: $0 < Fract\ a\ b$
and *notin*: $Fract\ a\ b \notin A$
have *cpos* [*simp*]: $0 < c$
by (*simp add*: *zero-less-Fract-iff* [*OF dpos*, *symmetric*] *upos*)
have *apos* [*simp*]: $0 < a$
by (*simp add*: *zero-less-Fract-iff* [*OF bpos*, *symmetric*] *ypos*)
let *?k* = $a{*}d$
have *frle*: $Fract\ a\ b \leq Fract\ ?k\ 1 * (Fract\ c\ d)$
proof −
have *?thesis* = $((a * d * b * d) \leq c * b * (a * d * b * d))$
by (*simp add*: *mult-rat le-rat order-less-imp-not-eq2 mult-ac*)
moreover
have $(1 * (a * d * b * d)) \leq c * b * (a * d * b * d)$
by (*rule mult-mono*,
simp-all add: *int-one-le-iff-zero-less zero-less-mult-iff*
order-less-imp-le)

ultimately
show *?thesis* **by** *simp*
qed
have *k*: *0* ≤ *?k* **by** (*simp add*: *order-less-imp-le zero-less-mult-iff*)
from *Gleason9-34-exists* [*OF A closed k*]
obtain *z* **where** *z*: *z* ∈ *A*
and *mem*: *z* + *of-int ?k* ∗ *Fract c d* ∈ *A* **..**
have *less*: *z* + *of-int ?k* ∗ *Fract c d* < *Fract a b*
by (*rule not-in-preal-ub* [*OF A notin mem ypos*])
have *0*<*z* **by** (*rule preal-imp-pos* [*OF A z*])
with *frle* **and** *less* **show** *False* **by** (*simp add*: *Fract-of-int-eq*)
qed

lemma *Gleason9-34*:
assumes *A*: *A* ∈ *preal*
and *upos*: *0* < *u*
shows ∃ *r* ∈ *A*. *r* + *u* ∉ *A*
proof (*rule ccontr*, *simp*)
assume *closed*: ∀ *r*∈*A*. *r* + *u* ∈ *A*
from *preal-exists-bound* [*OF A*]
obtain *y* **where** *y*: *y* ∉ *A* **and** *ypos*: *0* < *y* **by** *blast*
show *False*
by (*rule Gleason9-34-contra* [*OF A closed upos ypos y*])
qed

10.8 Gleason's Lemma 9-3.6

lemma *lemma-gleason9-36*:
assumes *A*: *A* ∈ *preal*
and *x*: *1* < *x*
shows ∃ *r* ∈ *A*. *r*∗*x* ∉ *A*
proof −
from *preal-nonempty* [*OF A*]
obtain *y* **where** *y*: *y* ∈ *A* **and** *ypos*: *0*<*y* **..**
show *?thesis*
proof (*rule classical*)
assume ~(∃ *r*∈*A*. *r* ∗ *x* ∉ *A*)
with *y* **have** *ymem*: *y* ∗ *x* ∈ *A* **by** *blast*
from *ypos mult-strict-left-mono* [*OF x*]
have *yless*: *y* < *y*∗*x* **by** *simp*
let *?d* = *y*∗*x* − *y*
from *yless* **have** *dpos*: *0* < *?d* **and** *eq*: *y* + *?d* = *y*∗*x* **by** *auto*
from *Gleason9-34* [*OF A dpos*]
obtain *r* **where** *r*: *r*∈*A* **and** *notin*: *r* + *?d* ∉ *A* **..**
have *rpos*: *0*<*r* **by** (*rule preal-imp-pos* [*OF A r*])
with *dpos* **have** *rdpos*: *0* < *r* + *?d* **by** *arith*
have ~ (*r* + *?d* ≤ *y* + *?d*)
proof

```
    assume le: r + ?d ≤ y + ?d
    from ymem have yd: y + ?d ∈ A by (simp add: eq)
    have r + ?d ∈ A by (rule preal-downwards-closed' [OF A yd rdpos le])
    with notin show False by simp
   qed
   hence y < r by simp
   with ypos have  dless: ?d < (r * ?d)/y
    by (simp add: pos-less-divide-eq mult-commute [of ?d]
                  mult-strict-right-mono dpos)
   have r + ?d < r*x
   proof −
    have r + ?d < r + (r * ?d)/y by (simp add: dless)
    also with ypos have ... = (r/y) * (y + ?d)
     by (simp only: right-distrib divide-inverse mult-ac, simp)
    also have ... = r*x using ypos
     by (simp add: times-divide-eq-left)
    finally show r + ?d < r*x .
   qed
   with r notin rdpos
   show ∃r∈A. r * x ∉ A by (blast dest:  preal-downwards-closed [OF A])
  qed
qed
```

10.9 Existence of Inverse: Part 2

```
lemma mem-Rep-preal-inverse-iff:
     (z ∈ Rep-preal(inverse R)) =
      (0 < z ∧ (∃ y. z < y ∧ inverse y ∉ Rep-preal R))
apply (simp add: preal-inverse-def mem-inverse-set Rep-preal)
apply (simp add: inverse-set-def)
done

lemma Rep-preal-of-rat:
    0 < q ==> Rep-preal (preal-of-rat q) = {x. 0 < x ∧ x < q}
by (simp add: preal-of-rat-def rat-mem-preal)

lemma subset-inverse-mult-lemma:
  assumes xpos: 0 < x and xless: x < 1
  shows ∃ r u y. 0 < r & r < y & inverse y ∉ Rep-preal R &
    u ∈ Rep-preal R & x = r * u
proof −
  from xpos and xless have 1 < inverse x by (simp add: one-less-inverse-iff)
  from lemma-gleason9-36 [OF Rep-preal this]
  obtain r where r: r ∈ Rep-preal R
            and notin: r * (inverse x) ∉ Rep-preal R ..
  have rpos: 0<r by (rule preal-imp-pos [OF Rep-preal r])
  from preal-exists-greater [OF Rep-preal r]
  obtain u where u: u ∈ Rep-preal R and rless: r < u ..
  have upos: 0<u by (rule preal-imp-pos [OF Rep-preal u])
```

```
  show ?thesis
  proof (intro exI conjI)
    show 0 < x/u using xpos upos
      by (simp add: zero-less-divide-iff)
    show x/u < x/r using xpos upos rpos
      by (simp add: divide-inverse mult-less-cancel-left rless)
    show inverse (x / r) ∉ Rep-preal R using notin
      by (simp add: divide-inverse mult-commute)
    show u ∈ Rep-preal R by (rule u)
    show x = x / u * u using upos
      by (simp add: divide-inverse mult-commute)
  qed
qed

lemma subset-inverse-mult:
    Rep-preal(preal-of-rat 1) ⊆ Rep-preal(inverse R * R)
apply (auto simp add: Bex-def Rep-preal-of-rat mem-Rep-preal-inverse-iff
                      mem-Rep-preal-mult-iff)
apply (blast dest: subset-inverse-mult-lemma)
done

lemma inverse-mult-subset-lemma:
  assumes rpos: 0 < r
    and rless: r < y
    and notin: inverse y ∉ Rep-preal R
    and q: q ∈ Rep-preal R
  shows r*q < 1
proof −
  have q < inverse y using rpos rless
    by (simp add: not-in-preal-ub [OF Rep-preal notin] q)
  hence r * q < r/y using rpos
    by (simp add: divide-inverse mult-less-cancel-left)
  also have ... ≤ 1 using rpos rless
    by (simp add: pos-divide-le-eq)
  finally show ?thesis .
qed

lemma inverse-mult-subset:
    Rep-preal(inverse R * R) ⊆ Rep-preal(preal-of-rat 1)
apply (auto simp add: Bex-def Rep-preal-of-rat mem-Rep-preal-inverse-iff
                      mem-Rep-preal-mult-iff)
apply (simp add: zero-less-mult-iff preal-imp-pos [OF Rep-preal])
apply (blast intro: inverse-mult-subset-lemma)
done

lemma preal-mult-inverse: inverse R * R = (1::preal)
unfolding preal-one-def
apply (rule Rep-preal-inject [THEN iffD1])
apply (rule equalityI [OF inverse-mult-subset subset-inverse-mult])
```

done

lemma *preal-mult-inverse-right*: $R * inverse\ R = (1::preal)$
apply (*rule preal-mult-commute* [*THEN subst*])
apply (*rule preal-mult-inverse*)
done

Theorems needing *Gleason9-34*

lemma *Rep-preal-self-subset*: *Rep-preal* $(R) \subseteq$ *Rep-preal*$(R + S)$
proof
  fix r
  assume r: $r \in$ *Rep-preal* R
  have *rpos*: $0<r$ **by** (*rule preal-imp-pos* [*OF Rep-preal r*])
  from *mem-Rep-preal-Ex*
  obtain y **where** y: $y \in$ *Rep-preal* S **..**
  have *ypos*: $0<y$ **by** (*rule preal-imp-pos* [*OF Rep-preal y*])
  have *ry*: $r+y \in$ *Rep-preal*$(R + S)$ **using** r y
    by (*auto simp add*: *mem-Rep-preal-add-iff*)
  show $r \in$ *Rep-preal*$(R + S)$ **using** r *ypos rpos*
    by (*simp add*: *preal-downwards-closed* [*OF Rep-preal ry*])
qed

lemma *Rep-preal-sum-not-subset*: $\sim$ *Rep-preal* $(R + S) \subseteq$ *Rep-preal*(R)
proof −
  from *mem-Rep-preal-Ex*
  obtain y **where** y: $y \in$ *Rep-preal* S **..**
  have *ypos*: $0<y$ **by** (*rule preal-imp-pos* [*OF Rep-preal y*])
  from *Gleason9-34* [*OF Rep-preal ypos*]
  obtain r **where** r: $r \in$ *Rep-preal* R **and** *notin*: $r + y \notin$ *Rep-preal* R **..**
  have $r + y \in$ *Rep-preal* $(R + S)$ **using** r y
    by (*auto simp add*: *mem-Rep-preal-add-iff*)
  thus *?thesis* **using** *notin* **by** *blast*
qed

lemma *Rep-preal-sum-not-eq*: *Rep-preal* $(R + S) \neq$ *Rep-preal*(R)
by (*insert Rep-preal-sum-not-subset*, *blast*)

at last, Gleason prop. 9-3.5(iii) page 123

lemma *preal-self-less-add-left*: $(R::preal) < R + S$
apply (*unfold preal-less-def psubset-def*)
apply (*simp add*: *Rep-preal-self-subset Rep-preal-sum-not-eq* [*THEN not-sym*])
done

lemma *preal-self-less-add-right*: $(R::preal) < S + R$
by (*simp add*: *preal-add-commute preal-self-less-add-left*)

lemma *preal-not-eq-self*: $x \neq x + (y::preal)$
by (*insert preal-self-less-add-left* [*of x y*], *auto*)

10.10 Subtraction for Positive Reals

Gleason prop. 9-3.5(iv), page 123: proving $A < B \Longrightarrow \exists\, D.\ A + D = B$. We define the claimed D and show that it is a positive real

Part 1 of Dedekind sections definition

lemma *diff-set-not-empty*:
  $R < S$ ==> $\{\} \subset$ *diff-set* (*Rep-preal S*) (*Rep-preal R*)
apply (*auto simp add*: *preal-less-def diff-set-def elim*!: *equalityE*)
apply (*frule-tac x1* = *S* **in** *Rep-preal* [*THEN preal-exists-greater*])
apply (*drule preal-imp-pos* [*OF Rep-preal*], *clarify*)
apply (*cut-tac a*=*x* **and** *b*=*u* **in** *add-eq-exists*, *force*)
done

Part 2 of Dedekind sections definition

lemma *diff-set-nonempty*:
  $\exists\, q.\ 0 < q$ & $q \notin$ *diff-set* (*Rep-preal S*) (*Rep-preal R*)
apply (*cut-tac X* = *S* **in** *Rep-preal-exists-bound*)
apply (*erule exE*)
apply (*rule-tac x* = *x* **in** *exI*, *auto*)
apply (*simp add*: *diff-set-def*)
apply (*auto dest*: *Rep-preal* [*THEN preal-downwards-closed*])
done

lemma *diff-set-not-rat-set*:
  diff-set (*Rep-preal S*) (*Rep-preal R*) < {*r*. *0* < *r*} (**is** *?lhs* < *?rhs*)
proof
  show *?lhs* $\subseteq$ *?rhs* **by** (*auto simp add*: *diff-set-def*)
  show *?lhs* $\neq$ *?rhs* **using** *diff-set-nonempty* **by** *blast*
qed

Part 3 of Dedekind sections definition

lemma *diff-set-lemma3*:
  [|*R* < *S*; *u* $\in$ *diff-set* (*Rep-preal S*) (*Rep-preal R*); *0* < *z*; *z* < *u*|]
   ==> *z* $\in$ *diff-set* (*Rep-preal S*) (*Rep-preal R*)
apply (*auto simp add*: *diff-set-def*)
apply (*rule-tac x*=*x* **in** *exI*)
apply (*drule Rep-preal* [*THEN preal-downwards-closed*], *auto*)
done

Part 4 of Dedekind sections definition

lemma *diff-set-lemma4*:
  [|*R* < *S*; *y* $\in$ *diff-set* (*Rep-preal S*) (*Rep-preal R*)|]
   ==> $\exists\, u \in$ *diff-set* (*Rep-preal S*) (*Rep-preal R*). *y* < *u*
apply (*auto simp add*: *diff-set-def*)
apply (*drule Rep-preal* [*THEN preal-exists-greater*], *clarify*)
apply (*cut-tac a*=*x*+*y* **and** *b*=*u* **in** *add-eq-exists*, *clarify*)
apply (*rule-tac x*=*y*+*xa* **in** *exI*)

apply (*auto simp add*: *add-ac*)
done

lemma *mem-diff-set*:
$R < S \Longrightarrow$ *diff-set* (*Rep-preal S*) (*Rep-preal R*) $\in$ *preal*
apply (*unfold preal-def cut-def*)
apply (*blast intro*!: *diff-set-not-empty diff-set-not-rat-set*
diff-set-lemma3 diff-set-lemma4)
done

lemma *mem-Rep-preal-diff-iff*:
$R < S \Longrightarrow$
($z \in$ *Rep-preal*$(S-R)) =$
($\exists x.\ 0 < x\ \&\ 0 < z\ \&\ x \notin$ *Rep-preal R* $\&\ x + z \in$ *Rep-preal S*)
apply (*simp add*: *preal-diff-def mem-diff-set Rep-preal*)
apply (*force simp add*: *diff-set-def*)
done

proving that $R + D \leq S$

lemma *less-add-left-lemma*:
assumes *Rless*: $R < S$
and *a*: $a \in$ *Rep-preal R*
and *cb*: $c + b \in$ *Rep-preal S*
and $c \notin$ *Rep-preal R*
and $0 < b$
and $0 < c$
shows $a + b \in$ *Rep-preal S*
proof −
have $0<a$ **by** (*rule preal-imp-pos* [*OF Rep-preal a*])
moreover
have $a < c$ **using** *prems*
by (*blast intro*: *not-in-Rep-preal-ub*)
ultimately show *?thesis* **using** *prems*
by (*simp add*: *preal-downwards-closed* [*OF Rep-preal cb*])
qed

lemma *less-add-left-le1*:
$R < (S$::*preal*$) \Longrightarrow R + (S-R) \leq S$
apply (*auto simp add*: *Bex-def preal-le-def mem-Rep-preal-add-iff*
mem-Rep-preal-diff-iff)
apply (*blast intro*: *less-add-left-lemma*)
done

10.11 proving that $S \leq R + D$ — trickier

lemma *lemma-sum-mem-Rep-preal-ex*:
$x \in$ *Rep-preal S* $\Longrightarrow \exists e.\ 0 < e\ \&\ x + e \in$ *Rep-preal S*
apply (*drule Rep-preal* [*THEN preal-exists-greater*], *clarify*)
apply (*cut-tac a*=*x* **and** *b*=*u* **in** *add-eq-exists*, *auto*)

done

lemma *less-add-left-lemma2*:
  assumes *Rless*: $R < S$
    and x: $x \in$ *Rep-preal* S
    and *xnot*: $x \notin$ *Rep-preal* R
  shows $\exists u\ v\ z.\ 0 < v\ \&\ 0 < z\ \&\ u \in$ *Rep-preal* $R\ \&\ z \notin$ *Rep-preal* $R\ \&$
                $z + v \in$ *Rep-preal* $S\ \&\ x = u + v$
proof −
  have *xpos*: $0<x$ **by** (*rule preal-imp-pos* [*OF Rep-preal* x])
  from *lemma-sum-mem-Rep-preal-ex* [*OF* x]
  obtain e **where** *epos*: $0 < e$ **and** *xe*: $x + e \in$ *Rep-preal* S **by** *blast*
  from *Gleason9-34* [*OF Rep-preal epos*]
  obtain r **where** r: $r \in$ *Rep-preal* R **and** *notin*: $r + e \notin$ *Rep-preal* R **..**
  with x *xnot xpos* **have** *rless*: $r < x$ **by** (*blast intro*: *not-in-Rep-preal-ub*)
  from *add-eq-exists* [*of* r x]
  obtain y **where** *eq*: $x = r+y$ **by** *auto*
  show *?thesis*
  proof (*intro exI conjI*)
    show $r \in$ *Rep-preal* R **by** (*rule* r)
    show $r + e \notin$ *Rep-preal* R **by** (*rule notin*)
    show $r + e + y \in$ *Rep-preal* S **using** *xe eq* **by** (*simp add*: *add-ac*)
    show $x = r + y$ **by** (*simp add*: *eq*)
    show $0 < r + e$ **using** *epos preal-imp-pos* [*OF Rep-preal* r]
      by *simp*
    show $0 < y$ **using** *rless eq* **by** *arith*
  qed
qed

lemma *less-add-left-le2*: $R < (S::preal) ==> S \leq R + (S-R)$
apply (*auto simp add*: *preal-le-def*)
apply (*case-tac* $x \in$ *Rep-preal* R)
apply (*cut-tac Rep-preal-self-subset* [*of* R], *force*)
apply (*auto simp add*: *Bex-def mem-Rep-preal-add-iff mem-Rep-preal-diff-iff*)
apply (*blast dest*: *less-add-left-lemma2*)
done

lemma *less-add-left*: $R < (S::preal) ==> R + (S-R) = S$
by (*blast intro*: *preal-le-anti-sym* [*OF less-add-left-le1 less-add-left-le2*])

lemma *less-add-left-Ex*: $R < (S::preal) ==> \exists D.\ R + D = S$
by (*fast dest*: *less-add-left*)

lemma *preal-add-less2-mono1*: $R < (S::preal) ==> R + T < S + T$
apply (*auto dest*!: *less-add-left-Ex simp add*: *preal-add-assoc*)
apply (*rule-tac y1 = D* **in** *preal-add-commute* [*THEN subst*])
apply (*auto intro*: *preal-self-less-add-left simp add*: *preal-add-assoc* [*symmetric*])
done

lemma *preal-add-less2-mono2*: $R < (S::preal) \Longrightarrow T + R < T + S$
by (*auto intro*: *preal-add-less2-mono1 simp add*: *preal-add-commute* [*of T*])

lemma *preal-add-right-less-cancel*: $R + T < S + T \Longrightarrow R < (S::preal)$
apply (*insert linorder-less-linear* [*of R S*], *auto*)
apply (*drule-tac* $R = S$ **and** $T = T$ **in** *preal-add-less2-mono1*)
apply (*blast dest*: *order-less-trans*)
done

lemma *preal-add-left-less-cancel*: $T + R < T + S \Longrightarrow R < (S::preal)$
by (*auto elim*: *preal-add-right-less-cancel simp add*: *preal-add-commute* [*of T*])

lemma *preal-add-less-cancel-right*: $((R::preal) + T < S + T) = (R < S)$
by (*blast intro*: *preal-add-less2-mono1 preal-add-right-less-cancel*)

lemma *preal-add-less-cancel-left*: $(T + (R::preal) < T + S) = (R < S)$
by (*blast intro*: *preal-add-less2-mono2 preal-add-left-less-cancel*)

lemma *preal-add-le-cancel-right*: $((R::preal) + T \leq S + T) = (R \leq S)$
by (*simp add*: *linorder-not-less* [*symmetric*] *preal-add-less-cancel-right*)

lemma *preal-add-le-cancel-left*: $(T + (R::preal) \leq T + S) = (R \leq S)$
by (*simp add*: *linorder-not-less* [*symmetric*] *preal-add-less-cancel-left*)

lemma *preal-add-less-mono*:
$[\!|\ x1 < y1; x2 < y2\ |\!] \Longrightarrow x1 + x2 < y1 + (y2::preal)$
apply (*auto dest*!: *less-add-left-Ex simp add*: *preal-add-ac*)
apply (*rule preal-add-assoc* [*THEN subst*])
apply (*rule preal-self-less-add-right*)
done

lemma *preal-add-right-cancel*: $(R::preal) + T = S + T \Longrightarrow R = S$
apply (*insert linorder-less-linear* [*of R S*], *safe*)
apply (*drule-tac* [!] $T = T$ **in** *preal-add-less2-mono1*, *auto*)
done

lemma *preal-add-left-cancel*: $C + A = C + B \Longrightarrow A = (B::preal)$
by (*auto intro*: *preal-add-right-cancel simp add*: *preal-add-commute*)

lemma *preal-add-left-cancel-iff*: $(C + A = C + B) = ((A::preal) = B)$
by (*fast intro*: *preal-add-left-cancel*)

lemma *preal-add-right-cancel-iff*: $(A + C = B + C) = ((A::preal) = B)$
by (*fast intro*: *preal-add-right-cancel*)

lemmas *preal-cancels* =
preal-add-less-cancel-right preal-add-less-cancel-left
preal-add-le-cancel-right preal-add-le-cancel-left
preal-add-left-cancel-iff preal-add-right-cancel-iff

instance *preal* :: *ordered-cancel-ab-semigroup-add*
proof
  fix *a b c* :: *preal*
  show $a + b = a + c \Longrightarrow b = c$ **by** (*rule preal-add-left-cancel*)
  show $a \leq b \Longrightarrow c + a \leq c + b$ **by** (*simp only*: *preal-add-le-cancel-left*)
qed

10.12 Completeness of type *preal*

Prove that supremum is a cut

Part 1 of Dedekind sections definition

lemma *preal-sup-set-not-empty*:
  $P \neq \{\} ==> \{\} \subset (\bigcup X \in P.\ Rep\text{-}preal(X))$
apply *auto*
apply (*cut-tac* $X = x$ **in** *mem-Rep-preal-Ex*, *auto*)
done

Part 2 of Dedekind sections definition

lemma *preal-sup-not-exists*:
  $\forall X \in P.\ X \leq Y ==> \exists q.\ 0 < q\ \&\ q \notin (\bigcup X \in P.\ Rep\text{-}preal(X))$
apply (*cut-tac* $X = Y$ **in** *Rep-preal-exists-bound*)
apply (*auto simp add*: *preal-le-def*)
done

lemma *preal-sup-set-not-rat-set*:
  $\forall X \in P.\ X \leq Y ==> (\bigcup X \in P.\ Rep\text{-}preal(X)) < \{r.\ 0 < r\}$
apply (*drule preal-sup-not-exists*)
apply (*blast intro*: *preal-imp-pos* [*OF Rep-preal*])
done

Part 3 of Dedekind sections definition

lemma *preal-sup-set-lemma3*:
  $[|P \neq \{\}; \forall X \in P.\ X \leq Y;\ u \in (\bigcup X \in P.\ Rep\text{-}preal(X));\ 0 < z;\ z < u|]$
   $==> z \in (\bigcup X \in P.\ Rep\text{-}preal(X))$
by (*auto elim*: *Rep-preal* [*THEN preal-downwards-closed*])

Part 4 of Dedekind sections definition

lemma *preal-sup-set-lemma4*:
  $[|P \neq \{\}; \forall X \in P.\ X \leq Y;\ y \in (\bigcup X \in P.\ Rep\text{-}preal(X))\ |]$
    $==> \exists u \in (\bigcup X \in P.\ Rep\text{-}preal(X)).\ y < u$
by (*blast dest*: *Rep-preal* [*THEN preal-exists-greater*])

lemma *preal-sup*:
  $[|P \neq \{\}; \forall X \in P.\ X \leq Y|] ==> (\bigcup X \in P.\ Rep\text{-}preal(X)) \in preal$
apply (*unfold preal-def cut-def*)
apply (*blast intro*!: *preal-sup-set-not-empty preal-sup-set-not-rat-set*

preal-sup-set-lemma3 preal-sup-set-lemma4)
done

lemma *preal-psup-le*:
[| ∀ *X* ∈ *P*. *X* ≤ *Y*; *x* ∈ *P* |] ==> *x* ≤ *psup P*
apply (*simp* (*no-asm-simp*) *add*: *preal-le-def*)
apply (*subgoal-tac P* ≠ {})
apply (*auto simp add*: *psup-def preal-sup*)
done

lemma *psup-le-ub*: [| *P* ≠ {}; ∀ *X* ∈ *P*. *X* ≤ *Y* |] ==> *psup P* ≤ *Y*
apply (*simp* (*no-asm-simp*) *add*: *preal-le-def*)
apply (*simp add*: *psup-def preal-sup*)
apply (*auto simp add*: *preal-le-def*)
done

Supremum property

lemma *preal-complete*:
[| *P* ≠ {}; ∀ *X* ∈ *P*. *X* ≤ *Y* |] ==> (∃ *X* ∈ *P*. *Z* < *X*) = (*Z* < *psup P*)
apply (*simp add*: *preal-less-def psup-def preal-sup*)
apply (*auto simp add*: *preal-le-def*)
apply (*rename-tac U*)
apply (*cut-tac x* = *U* **and** *y* = *Z* **in** *linorder-less-linear*)
apply (*auto simp add*: *preal-less-def*)
done

10.13 The Embedding from *rat* into *preal*

lemma *preal-of-rat-add-lemma1*:
[|*x* < *y* + *z*; 0 < *x*; 0 < *y*|] ==> *x* * *y* * *inverse* (*y* + *z*) < (*y*::*rat*)
apply (*frule-tac c* = *y* * *inverse* (*y* + *z*) **in** *mult-strict-right-mono*)
apply (*simp add*: *zero-less-mult-iff*)
apply (*simp add*: *mult-ac*)
done

lemma *preal-of-rat-add-lemma2*:
assumes *u* < *x* + *y*
and 0 < *x*
and 0 < *y*
and 0 < *u*
shows ∃ *v w*::*rat*. *w* < *y* & 0 < *v* & *v* < *x* & 0 < *w* & *u* = *v* + *w*
proof (*intro exI conjI*)
show *u* * *x* * *inverse*(*x*+*y*) < *x* **using** *prems*
by (*simp add*: *preal-of-rat-add-lemma1*)
show *u* * *y* * *inverse*(*x*+*y*) < *y* **using** *prems*
by (*simp add*: *preal-of-rat-add-lemma1 add-commute* [*of x*])
show 0 < *u* * *x* * *inverse* (*x* + *y*) **using** *prems*
by (*simp add*: *zero-less-mult-iff*)
show 0 < *u* * *y* * *inverse* (*x* + *y*) **using** *prems*

by (*simp add*: *zero-less-mult-iff*)
show $u = u * x * inverse\ (x + y) + u * y * inverse\ (x + y)$ **using** *prems*
by (*simp add*: *left-distrib* [*symmetric*] *right-distrib* [*symmetric*] *mult-ac*)
qed

lemma *preal-of-rat-add*:
$[|\ 0 < x;\ 0 < y|]$
$==>$ *preal-of-rat* $((x::rat) + y) =$ *preal-of-rat* $x +$ *preal-of-rat* y
apply (*unfold preal-of-rat-def preal-add-def*)
apply (*simp add*: *rat-mem-preal*)
apply (*rule-tac* $f =$ *Abs-preal* **in** *arg-cong*)
apply (*auto simp add*: *add-set-def*)
apply (*blast dest*: *preal-of-rat-add-lemma2*)
done

lemma *preal-of-rat-mult-lemma1*:
$[|x < y;\ 0 < x;\ 0 < z|] ==> x * z * inverse\ y < (z::rat)$
apply (*frule-tac* $c = z * inverse\ y$ **in** *mult-strict-right-mono*)
apply (*simp add*: *zero-less-mult-iff*)
apply (*subgoal-tac* $y * (z * inverse\ y) = z * (y * inverse\ y)$)
apply (*simp-all add*: *mult-ac*)
done

lemma *preal-of-rat-mult-lemma2*:
assumes *xless*: $x < y * z$
and *xpos*: $0 < x$
and *ypos*: $0 < y$
shows $x * z * inverse\ y * inverse\ z < (z::rat)$
proof −
have $0 < y * z$ **using** *prems* **by** *simp*
hence *zpos*: $0 < z$ **using** *prems* **by** (*simp add*: *zero-less-mult-iff*)
have $x * z * inverse\ y * inverse\ z = x * inverse\ y * (z * inverse\ z)$
by (*simp add*: *mult-ac*)
also have $... = x/y$ **using** *zpos*
by (*simp add*: *divide-inverse*)
also from *xless* **have** $... < z$
by (*simp add*: *pos-divide-less-eq* [*OF ypos*] *mult-commute*)
finally show *?thesis* **.**
qed

lemma *preal-of-rat-mult-lemma3*:
assumes *uless*: $u < x * y$
and $0 < x$
and $0 < y$
and $0 < u$
shows $\exists\, v\ w::rat.\ v < x\ \&\ w < y\ \&\ 0 < v\ \&\ 0 < w\ \&\ u = v * w$
proof −
from *dense* [*OF uless*]
obtain r **where** $u < r\ r < x * y$ **by** *blast*

```
  thus ?thesis
  proof (intro exI conjI)
  show u * x * inverse r < x using prems
    by (simp add: preal-of-rat-mult-lemma1)
  show r * y * inverse x * inverse y < y using prems
    by (simp add: preal-of-rat-mult-lemma2)
  show 0 < u * x * inverse r using prems
    by (simp add: zero-less-mult-iff)
  show 0 < r * y * inverse x * inverse y using prems
    by (simp add: zero-less-mult-iff)
  have u * x * inverse r * (r * y * inverse x * inverse y) =
        u * (r * inverse r) * (x * inverse x) * (y * inverse y)
    by (simp only: mult-ac)
  thus u = u * x * inverse r * (r * y * inverse x * inverse y) using prems
    by simp
  qed
qed

lemma preal-of-rat-mult:
     [| 0 < x; 0 < y|]
      ==> preal-of-rat ((x::rat) * y) = preal-of-rat x * preal-of-rat y
apply (unfold preal-of-rat-def preal-mult-def)
apply (simp add: rat-mem-preal)
apply (rule-tac f = Abs-preal in arg-cong)
apply (auto simp add: zero-less-mult-iff mult-strict-mono mult-set-def)
apply (blast dest: preal-of-rat-mult-lemma3)
done

lemma preal-of-rat-less-iff:
      [| 0 < x; 0 < y|] ==> (preal-of-rat x < preal-of-rat y) = (x < y)
by (force simp add: preal-of-rat-def preal-less-def rat-mem-preal)

lemma preal-of-rat-le-iff:
      [| 0 < x; 0 < y|] ==> (preal-of-rat x ≤ preal-of-rat y) = (x ≤ y)
by (simp add: preal-of-rat-less-iff linorder-not-less [symmetric])

lemma preal-of-rat-eq-iff:
      [| 0 < x; 0 < y|] ==> (preal-of-rat x = preal-of-rat y) = (x = y)
by (simp add: preal-of-rat-le-iff order-eq-iff)

end
```

11 Defining the Reals from the Positive Reals

```
theory RealDef
imports PReal
uses (real-arith.ML)
begin
```

definition
realrel :: ((*preal* ∗ *preal*) ∗ (*preal* ∗ *preal*)) *set* **where**
realrel = {*p*. ∃ *x1 y1 x2 y2*. *p* = ((*x1*,*y1*),(*x2*,*y2*)) & *x1*+*y2* = *x2*+*y1*}

typedef (*Real*) *real* = *UNIV*//*realrel*
by (*auto simp add*: *quotient-def*)

definition

real-of-preal :: *preal* => *real* **where**
real-of-preal m = *Abs-Real*(*realrel*``{(*m* + *1*, *1*)})

instance *real* :: *zero*
real-zero-def: *0* == *Abs-Real*(*realrel*``{(*1*, *1*)}) **..**
lemmas [*code func del*] = *real-zero-def*

instance *real* :: *one*
real-one-def: *1* == *Abs-Real*(*realrel*``{(*1* + *1*, *1*)}) **..**
lemmas [*code func del*] = *real-one-def*

instance *real* :: *plus*
real-add-def: *z* + *w* ==
contents (⋃(*x*,*y*) ∈ *Rep-Real*(*z*). ⋃(*u*,*v*) ∈ *Rep-Real*(*w*).
{ *Abs-Real*(*realrel*``{(*x*+*u*, *y*+*v*)}) }) **..**
lemmas [*code func del*] = *real-add-def*

instance *real* :: *minus*
real-minus-def: − *r* == *contents* (⋃(*x*,*y*) ∈ *Rep-Real*(*r*). { *Abs-Real*(*realrel*``{(*y*,*x*)})
})
real-diff-def: *r* − (*s*::*real*) == *r* + − *s* **..**
lemmas [*code func del*] = *real-minus-def real-diff-def*

instance *real* :: *times*
real-mult-def:
z ∗ *w* ==
contents (⋃(*x*,*y*) ∈ *Rep-Real*(*z*). ⋃(*u*,*v*) ∈ *Rep-Real*(*w*).
{ *Abs-Real*(*realrel*``{(*x*∗*u* + *y*∗*v*, *x*∗*v* + *y*∗*u*)}) }) **..**
lemmas [*code func del*] = *real-mult-def*

instance *real* :: *inverse*
real-inverse-def: *inverse* (*R*::*real*) == (*THE S*. (*R* = *0* & *S* = *0*) | *S* ∗ *R* = *1*)
real-divide-def: *R* / (*S*::*real*) == *R* ∗ *inverse S* **..**
lemmas [*code func del*] = *real-inverse-def real-divide-def*

instance *real* :: *ord*
real-le-def: *z* ≤ (*w*::*real*) ==
∃ *x y u v*. *x*+*v* ≤ *u*+*y* & (*x*,*y*) ∈ *Rep-Real z* & (*u*,*v*) ∈ *Rep-Real w*
real-less-def: (*x* < (*y*::*real*)) == (*x* ≤ *y* & *x* ≠ *y*) **..**

lemmas [*code func del*] = *real-le-def real-less-def*

instance *real* :: *abs*
real-abs-def: *abs* (*r*::*real*) == (*if r* < *0 then* − *r else r*) **..**

instance *real* :: *sgn*
real-sgn-def: *sgn x* == (*if x*=*0 then 0 else if 0*<*x then 1 else* − *1*) **..**

11.1 Equivalence relation over positive reals

lemma *preal-trans-lemma*:
assumes *x* + *y1* = *x1* + *y*
and *x* + *y2* = *x2* + *y*
shows *x1* + *y2* = *x2* + (*y1*::*preal*)
proof −
have (*x1* + *y2*) + *x* = (*x* + *y2*) + *x1* **by** (*simp add*: *add-ac*)
also have ... = (*x2* + *y*) + *x1* **by** (*simp add*: *prems*)
also have ... = *x2* + (*x1* + *y*) **by** (*simp add*: *add-ac*)
also have ... = *x2* + (*x* + *y1*) **by** (*simp add*: *prems*)
also have ... = (*x2* + *y1*) + *x* **by** (*simp add*: *add-ac*)
finally have (*x1* + *y2*) + *x* = (*x2* + *y1*) + *x* **.**
thus *?thesis* **by** (*rule add-right-imp-eq*)
qed

lemma *realrel-iff* [*simp*]: (((*x1*,*y1*),(*x2*,*y2*)) ∈ *realrel*) = (*x1* + *y2* = *x2* + *y1*)
by (*simp add*: *realrel-def*)

lemma *equiv-realrel*: *equiv UNIV realrel*
apply (*auto simp add*: *equiv-def refl-def sym-def trans-def realrel-def*)
apply (*blast dest*: *preal-trans-lemma*)
done

Reduces equality of equivalence classes to the *realrel* relation: (*realrel* `` {*x*} = *realrel* `` {*y*}) = ((*x*, *y*) ∈ *realrel*)

lemmas *equiv-realrel-iff* =
eq-equiv-class-iff [*OF equiv-realrel UNIV-I UNIV-I*]

declare *equiv-realrel-iff* [*simp*]

lemma *realrel-in-real* [*simp*]: *realrel*``{(*x*,*y*)}: *Real*
by (*simp add*: *Real-def realrel-def quotient-def*, *blast*)

declare *Abs-Real-inject* [*simp*]
declare *Abs-Real-inverse* [*simp*]

Case analysis on the representation of a real number as an equivalence class of pairs of positive reals.

lemma *eq-Abs-Real* [*case-names Abs-Real*, *cases type*: *real*]:
  (!!*x y*. *z* = *Abs-Real*(*realrel*``{(*x*,*y*)}) ==> *P*) ==> *P*
apply (*rule Rep-Real* [*of z*, *unfolded Real-def*, *THEN quotientE*])
apply (*drule arg-cong* [**where** *f*=*Abs-Real*])
apply (*auto simp add*: *Rep-Real-inverse*)
done

11.2 Addition and Subtraction

lemma *real-add-congruent2-lemma*:
  [|*a* + *ba* = *aa* + *b*; *ab* + *bc* = *ac* + *bb*|]
  ==> *a* + *ab* + (*ba* + *bc*) = *aa* + *ac* + (*b* + (*bb*::*preal*))
apply (*simp add*: *add-assoc*)
apply (*rule add-left-commute* [*of ab*, *THEN ssubst*])
apply (*simp add*: *add-assoc* [*symmetric*])
apply (*simp add*: *add-ac*)
done

lemma *real-add*:
  Abs-Real (*realrel*``{(*x*,*y*)}) + *Abs-Real* (*realrel*``{(*u*,*v*)}) =
  Abs-Real (*realrel*``{(*x*+*u*, *y*+*v*)})
proof −
  have (λ*z w*. (λ(*x*,*y*). (λ(*u*,*v*). {*Abs-Real* (*realrel* `` {(*x*+*u*, *y*+*v*)})}) *w*) *z*)
      respects2 realrel
    by (*simp add*: *congruent2-def*, *blast intro*: *real-add-congruent2-lemma*)
  thus *?thesis*
    by (*simp add*: *real-add-def UN-UN-split-split-eq*
          UN-equiv-class2 [*OF equiv-realrel equiv-realrel*])
qed

lemma *real-minus*: − *Abs-Real*(*realrel*``{(*x*,*y*)}) = *Abs-Real*(*realrel* `` {(*y*,*x*)})
proof −
  have (λ(*x*,*y*). {*Abs-Real* (*realrel*``{(*y*,*x*)})}) *respects realrel*
    by (*simp add*: *congruent-def add-commute*)
  thus *?thesis*
    by (*simp add*: *real-minus-def UN-equiv-class* [*OF equiv-realrel*])
qed

instance *real* :: *ab-group-add*
proof
  fix *x y z* :: *real*
  show (*x* + *y*) + *z* = *x* + (*y* + *z*)
    by (*cases x*, *cases y*, *cases z*, *simp add*: *real-add add-assoc*)
  show *x* + *y* = *y* + *x*
    by (*cases x*, *cases y*, *simp add*: *real-add add-commute*)
  show *0* + *x* = *x*
    by (*cases x*, *simp add*: *real-add real-zero-def add-ac*)
  show − *x* + *x* = *0*
    by (*cases x*, *simp add*: *real-minus real-add real-zero-def add-commute*)

show $x - y = x + - y$
by (*simp add*: *real-diff-def*)
qed

11.3 Multiplication

lemma *real-mult-congruent2-lemma*:
!!(*x1*::*preal*). [| *x1* + *y2* = *x2* + *y1* |] ==>
x * *x1* + *y* * *y1* + (*x* * *y2* + *y* * *x2*) =
x * *x2* + *y* * *y2* + (*x* * *y1* + *y* * *x1*)
apply (*simp add*: *add-left-commute add-assoc* [*symmetric*])
apply (*simp add*: *add-assoc right-distrib* [*symmetric*])
apply (*simp add*: *add-commute*)
done

lemma *real-mult-congruent2*:
(%*p1 p2*.
(%(*x1*,*y1*). (%(*x2*,*y2*).
{ *Abs-Real* (*realrel*``{(*x1**x2* + *y1**y2*, *x1**y2*+*y1**x2*)}) }) *p2*) *p1*)
respects2 realrel
apply (*rule congruent2-commuteI* [*OF equiv-realrel*], *clarify*)
apply (*simp add*: *mult-commute add-commute*)
apply (*auto simp add*: *real-mult-congruent2-lemma*)
done

lemma *real-mult*:
Abs-Real((*realrel*``{(*x1*,*y1*)})) * *Abs-Real*((*realrel*``{(*x2*,*y2*)})) =
Abs-Real(*realrel* `` {(*x1**x2*+*y1**y2*,*x1**y2*+*y1**x2*)})
by (*simp add*: *real-mult-def UN-UN-split-split-eq*
UN-equiv-class2 [*OF equiv-realrel equiv-realrel real-mult-congruent2*])

lemma *real-mult-commute*: (*z*::*real*) * *w* = *w* * *z*
by (*cases z*, *cases w*, *simp add*: *real-mult add-ac mult-ac*)

lemma *real-mult-assoc*: ((*z1*::*real*) * *z2*) * *z3* = *z1* * (*z2* * *z3*)
apply (*cases z1*, *cases z2*, *cases z3*)
apply (*simp add*: *real-mult right-distrib add-ac mult-ac*)
done

lemma *real-mult-1*: (*1*::*real*) * *z* = *z*
apply (*cases z*)
apply (*simp add*: *real-mult real-one-def right-distrib*
mult-1-right mult-ac add-ac)
done

lemma *real-add-mult-distrib*: ((*z1*::*real*) + *z2*) * *w* = (*z1* * *w*) + (*z2* * *w*)
apply (*cases z1*, *cases z2*, *cases w*)
apply (*simp add*: *real-add real-mult right-distrib add-ac mult-ac*)
done

one and zero are distinct

lemma *real-zero-not-eq-one*: $0 \neq (1::real)$
proof −
  have $(1::preal) < 1 + 1$
    by (*simp add*: *preal-self-less-add-left*)
  thus *?thesis*
    by (*simp add*: *real-zero-def real-one-def*)
qed

instance *real* :: *comm-ring-1*
proof
  fix $x\ y\ z$ :: *real*
  show $(x * y) * z = x * (y * z)$ **by** (*rule real-mult-assoc*)
  show $x * y = y * x$ **by** (*rule real-mult-commute*)
  show $1 * x = x$ **by** (*rule real-mult-1*)
  show $(x + y) * z = x * z + y * z$ **by** (*rule real-add-mult-distrib*)
  show $0 \neq (1::real)$ **by** (*rule real-zero-not-eq-one*)
qed

11.4 Inverse and Division

lemma *real-zero-iff*: *Abs-Real* (*realrel* `` $\{(x, x)\}$) = *0*
by (*simp add*: *real-zero-def add-commute*)

Instead of using an existential quantifier and constructing the inverse within the proof, we could define the inverse explicitly.

lemma *real-mult-inverse-left-ex*: $x \neq 0 \Longrightarrow \exists y.\ y*x = (1::real)$
apply (*simp add*: *real-zero-def real-one-def*, *cases x*)
apply (*cut-tac x = xa* **and** *y = y* **in** *linorder-less-linear*)
apply (*auto dest*!: *less-add-left-Ex simp add*: *real-zero-iff*)
apply (*rule-tac*
      x = Abs-Real (*realrel*``$\{(1, inverse\ (D) + 1)\}$)
    in *exI*)
apply (*rule-tac* [*2*]
      x = Abs-Real (*realrel*``$\{(inverse\ (D) + 1, 1)\}$)
    in *exI*)
apply (*auto simp add*: *real-mult preal-mult-inverse-right ring-simps*)
done

lemma *real-mult-inverse-left*: $x \neq 0 \Longrightarrow inverse(x)*x = (1::real)$
apply (*simp add*: *real-inverse-def*)
apply (*drule real-mult-inverse-left-ex*, *safe*)
apply (*rule theI*, *assumption*, *rename-tac z*)
apply (*subgoal-tac* $(z * x) * y = z * (x * y)$)
apply (*simp add*: *mult-commute*)
apply (*rule mult-assoc*)
done

11.5 The Real Numbers form a Field

instance *real* :: *field*
proof
  fix *x y z* :: *real*
  show *x* ≠ *0* ==> *inverse x* ∗ *x* = *1* **by** (*rule real-mult-inverse-left*)
  show *x* / *y* = *x* ∗ *inverse y* **by** (*simp add*: *real-divide-def*)
qed

Inverse of zero! Useful to simplify certain equations

lemma *INVERSE-ZERO*: *inverse 0* = (*0*::*real*)
by (*simp add*: *real-inverse-def*)

instance *real* :: *division-by-zero*
proof
  show *inverse 0* = (*0*::*real*) **by** (*rule INVERSE-ZERO*)
qed

11.6 The ≤ Ordering

lemma *real-le-refl*: *w* ≤ (*w*::*real*)
by (*cases w*, *force simp add*: *real-le-def*)

The arithmetic decision procedure is not set up for type preal. This lemma is currently unused, but it could simplify the proofs of the following two lemmas.

lemma *preal-eq-le-imp-le*:
  assumes *eq*: *a*+*b* = *c*+*d* **and** *le*: *c* ≤ *a*
  shows *b* ≤ (*d*::*preal*)
proof −
  have *c*+*d* ≤ *a*+*d* **by** (*simp add*: *prems*)
  hence *a*+*b* ≤ *a*+*d* **by** (*simp add*: *prems*)
  thus *b* ≤ *d* **by** *simp*
qed

lemma *real-le-lemma*:
  assumes *l*: *u1* + *v2* ≤ *u2* + *v1*
      and *x1* + *v1* = *u1* + *y1*
      and *x2* + *v2* = *u2* + *y2*
  shows *x1* + *y2* ≤ *x2* + (*y1*::*preal*)
proof −
  have (*x1*+*v1*) + (*u2*+*y2*) = (*u1*+*y1*) + (*x2*+*v2*) **by** (*simp add*: *prems*)
  hence (*x1*+*y2*) + (*u2*+*v1*) = (*x2*+*y1*) + (*u1*+*v2*) **by** (*simp add*: *add-ac*)
  also have ... ≤ (*x2*+*y1*) + (*u2*+*v1*) **by** (*simp add*: *prems*)
  finally show *?thesis* **by** *simp*
qed

lemma *real-le*:
    (*Abs-Real*(*realrel*``{(*x1*,*y1*)}) ≤ *Abs-Real*(*realrel*``{(*x2*,*y2*)})) =

```
    (x1 + y2 ≤ x2 + y1)
apply (simp add: real-le-def)
apply (auto intro: real-le-lemma)
done

lemma real-le-anti-sym: [| z ≤ w; w ≤ z |] ==> z = (w::real)
by (cases z, cases w, simp add: real-le)

lemma real-trans-lemma:
  assumes x + v ≤ u + y
      and u + v' ≤ u' + v
      and x2 + v2 = u2 + y2
  shows x + v' ≤ u' + (y::preal)
proof −
  have (x+v') + (u+v) = (x+v) + (u+v') by (simp add: add-ac)
  also have ... ≤ (u+y) + (u+v') by (simp add: prems)
  also have ... ≤ (u+y) + (u'+v) by (simp add: prems)
  also have ... = (u'+y) + (u+v)  by (simp add: add-ac)
  finally show ?thesis by simp
qed

lemma real-le-trans: [| i ≤ j; j ≤ k |] ==> i ≤ (k::real)
apply (cases i, cases j, cases k)
apply (simp add: real-le)
apply (blast intro: real-trans-lemma)
done


lemma real-less-le: ((w::real) < z) = (w ≤ z & w ≠ z)
by (simp add: real-less-def)

instance real :: order
proof qed
 (assumption |
  rule real-le-refl real-le-trans real-le-anti-sym real-less-le)+


lemma real-le-linear: (z::real) ≤ w | w ≤ z
apply (cases z, cases w)
apply (auto simp add: real-le real-zero-def add-ac)
done


instance real :: linorder
  by (intro-classes, rule real-le-linear)


lemma real-le-eq-diff: (x ≤ y) = (x−y ≤ (0::real))
apply (cases x, cases y)
```

apply (*auto simp add*: *real-le real-zero-def real-diff-def real-add real-minus add-ac*)
apply (*simp-all add*: *add-assoc* [*symmetric*])
done

lemma *real-add-left-mono*:
  assumes *le*: $x \leq y$ **shows** $z + x \leq z + (y::real)$
proof −
  have $z + x - (z + y) = (z + -z) + (x - y)$
    by (*simp add*: *diff-minus add-ac*)
  with *le* **show** *?thesis*
    by (*simp add*: *real-le-eq-diff* [*of x*] *real-le-eq-diff* [*of z+x*] *diff-minus*)
qed

lemma *real-sum-gt-zero-less*: $(0 < S + (-W::real)) ==> (W < S)$
by (*simp add*: *linorder-not-le* [*symmetric*] *real-le-eq-diff* [*of S*] *diff-minus*)

lemma *real-less-sum-gt-zero*: $(W < S) ==> (0 < S + (-W::real))$
by (*simp add*: *linorder-not-le* [*symmetric*] *real-le-eq-diff* [*of S*] *diff-minus*)

lemma *real-mult-order*: [| $0 < x$; $0 < y$ |] ==> $(0::real) < x * y$
apply (*cases x*, *cases y*)
apply (*simp add*: *linorder-not-le* [**where** *'a* = *real*, *symmetric*]
            linorder-not-le [**where** *'a* = *preal*]
             real-zero-def real-le real-mult)
  — Reduce to the (simpler) ≤ relation
apply (*auto dest*!: *less-add-left-Ex*
    simp add: *add-ac mult-ac*
        right-distrib preal-self-less-add-left)
done

lemma *real-mult-less-mono2*: [| $(0::real) < z$; $x < y$ |] ==> $z * x < z * y$
apply (*rule real-sum-gt-zero-less*)
apply (*drule real-less-sum-gt-zero* [*of x y*])
apply (*drule real-mult-order*, *assumption*)
apply (*simp add*: *right-distrib*)
done

instance *real* :: *distrib-lattice*
  inf x y ≡ *min x y*
  sup x y ≡ *max x y*
  by *default* (*auto simp add*: *inf-real-def sup-real-def min-max.sup-inf-distrib1*)

11.7 The Reals Form an Ordered Field

instance *real* :: *ordered-field*
proof
  fix *x y z* :: *real*
  show $x \leq y ==> z + x \leq z + y$ **by** (*rule real-add-left-mono*)

show $x < y ==> 0 < z ==> z * x < z * y$ **by** (*rule real-mult-less-mono2*)
show $|x| = (if\ x < 0\ then\ -x\ else\ x)$ **by** (*simp only*: *real-abs-def*)
show $sgn\ x = (if\ x=0\ then\ 0\ else\ if\ 0<x\ then\ 1\ else\ -\ 1)$
by (*simp only*: *real-sgn-def*)
qed

instance *real* :: *lordered-ab-group-add* **..**

The function *real-of-preal* requires many proofs, but it seems to be essential for proving completeness of the reals from that of the positive reals.

lemma *real-of-preal-add*:
$real\text{-}of\text{-}preal\ ((x::preal) + y) = real\text{-}of\text{-}preal\ x + real\text{-}of\text{-}preal\ y$
by (*simp add*: *real-of-preal-def real-add left-distrib add-ac*)

lemma *real-of-preal-mult*:
$real\text{-}of\text{-}preal\ ((x::preal) * y) = real\text{-}of\text{-}preal\ x* real\text{-}of\text{-}preal\ y$
by (*simp add*: *real-of-preal-def real-mult right-distrib add-ac mult-ac*)

Gleason prop 9-4.4 p 127

lemma *real-of-preal-trichotomy*:
$\exists m.\ (x::real) = real\text{-}of\text{-}preal\ m \mid x = 0 \mid x = -(real\text{-}of\text{-}preal\ m)$
apply (*simp add*: *real-of-preal-def real-zero-def*, *cases x*)
apply (*auto simp add*: *real-minus add-ac*)
apply (*cut-tac x = x* **and** *y = y* **in** *linorder-less-linear*)
apply (*auto dest*!: *less-add-left-Ex simp add*: *add-assoc* [*symmetric*])
done

lemma *real-of-preal-leD*:
$real\text{-}of\text{-}preal\ m1 \leq real\text{-}of\text{-}preal\ m2 ==> m1 \leq m2$
by (*simp add*: *real-of-preal-def real-le*)

lemma *real-of-preal-lessI*: $m1 < m2 ==> real\text{-}of\text{-}preal\ m1 < real\text{-}of\text{-}preal\ m2$
by (*auto simp add*: *real-of-preal-leD linorder-not-le* [*symmetric*])

lemma *real-of-preal-lessD*:
$real\text{-}of\text{-}preal\ m1 < real\text{-}of\text{-}preal\ m2 ==> m1 < m2$
by (*simp add*: *real-of-preal-def real-le linorder-not-le* [*symmetric*])

lemma *real-of-preal-less-iff* [*simp*]:
$(real\text{-}of\text{-}preal\ m1 < real\text{-}of\text{-}preal\ m2) = (m1 < m2)$
by (*blast intro*: *real-of-preal-lessI real-of-preal-lessD*)

lemma *real-of-preal-le-iff*:
$(real\text{-}of\text{-}preal\ m1 \leq real\text{-}of\text{-}preal\ m2) = (m1 \leq m2)$
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *real-of-preal-zero-less*: $0 < real\text{-}of\text{-}preal\ m$
apply (*insert preal-self-less-add-left* [*of 1 m*])
apply (*auto simp add*: *real-zero-def real-of-preal-def*

real-less-def real-le-def add-ac)
apply (*rule-tac x=m + 1* **in** *exI*, *rule-tac x=1* **in** *exI*)
apply (*simp add*: *add-ac*)
done

lemma *real-of-preal-minus-less-zero*: $- \text{real-of-preal } m < 0$
by (*simp add*: *real-of-preal-zero-less*)

lemma *real-of-preal-not-minus-gt-zero*: $\sim\ 0 < - \text{real-of-preal } m$
proof −
  from *real-of-preal-minus-less-zero*
  show *?thesis* **by** (*blast dest*: *order-less-trans*)
qed

11.8 Theorems About the Ordering

lemma *real-gt-zero-preal-Ex*: $(0 < x) = (\exists y.\ x = \text{real-of-preal } y)$
apply (*auto simp add*: *real-of-preal-zero-less*)
apply (*cut-tac x = x* **in** *real-of-preal-trichotomy*)
apply (*blast elim*!: *real-of-preal-not-minus-gt-zero* [*THEN notE*])
done

lemma *real-gt-preal-preal-Ex*:
    $\text{real-of-preal } z < x \Longrightarrow \exists y.\ x = \text{real-of-preal } y$
by (*blast dest*!: *real-of-preal-zero-less* [*THEN order-less-trans*]
        intro: *real-gt-zero-preal-Ex* [*THEN iffD1*])

lemma *real-ge-preal-preal-Ex*:
    $\text{real-of-preal } z \leq x \Longrightarrow \exists y.\ x = \text{real-of-preal } y$
by (*blast dest*: *order-le-imp-less-or-eq real-gt-preal-preal-Ex*)

lemma *real-less-all-preal*: $y \leq 0 \Longrightarrow \forall x.\ y < \text{real-of-preal } x$
by (*auto elim*: *order-le-imp-less-or-eq* [*THEN disjE*]
        intro: *real-of-preal-zero-less* [*THEN* [*2*] *order-less-trans*]
        simp add: *real-of-preal-zero-less*)

lemma *real-less-all-real2*: $\sim\ 0 < y \Longrightarrow \forall x.\ y < \text{real-of-preal } x$
by (*blast intro*!: *real-less-all-preal linorder-not-less* [*THEN iffD1*])

11.9 More Lemmas

lemma *real-mult-left-cancel*: $(c::real) \neq 0 \Longrightarrow (c*a=c*b) = (a=b)$
by *auto*

lemma *real-mult-right-cancel*: $(c::real) \neq 0 \Longrightarrow (a*c=b*c) = (a=b)$
by *auto*

lemma *real-mult-less-iff1* [*simp*]: $(0::real) < z \Longrightarrow (x*z < y*z) = (x < y)$
  by (*force elim*: *order-less-asym*
        simp add: *Ring-and-Field.mult-less-cancel-right*)

lemma *real-mult-le-cancel-iff1* [*simp*]: (*0*::*real*) < *z* ==> (*x*∗*z* ≤ *y*∗*z*) = (*x*≤*y*)
apply (*simp add*: *mult-le-cancel-right*)
apply (*blast intro*: *elim*: *order-less-asym*)
done

lemma *real-mult-le-cancel-iff2* [*simp*]: (*0*::*real*) < *z* ==> (*z*∗*x* ≤ *z*∗*y*) = (*x*≤*y*)
by(*simp add*:*mult-commute*)

lemma *real-inverse-gt-one*: [| (*0*::*real*) < *x*; *x* < *1* |] ==> *1* < *inverse x*
by (*simp add*: *one-less-inverse-iff*)

11.10 Embedding numbers into the Reals

abbreviation
  real-of-nat :: *nat* ⇒ *real*
where
  real-of-nat ≡ *of-nat*

abbreviation
  real-of-int :: *int* ⇒ *real*
where
  real-of-int ≡ *of-int*

abbreviation
  real-of-rat :: *rat* ⇒ *real*
where
  real-of-rat ≡ *of-rat*

consts

  real :: *'a* => *real*

defs (**overloaded**)
  real-of-nat-def [*code inline*]: *real* == *real-of-nat*
  real-of-int-def [*code inline*]: *real* == *real-of-int*

lemma *real-eq-of-nat*: *real* = *of-nat*
  unfolding *real-of-nat-def* **..**

lemma *real-eq-of-int*: *real* = *of-int*
  unfolding *real-of-int-def* **..**

lemma *real-of-int-zero* [*simp*]: *real* (*0*::*int*) = *0*
by (*simp add*: *real-of-int-def*)

lemma *real-of-one* [*simp*]: *real* (*1*::*int*) = (*1*::*real*)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-add* [*simp*]: *real*(x + y) = *real* (x::*int*) + *real* y
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-minus* [*simp*]: *real*(−x) = −*real* (x::*int*)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-diff* [*simp*]: *real*(x − y) = *real* (x::*int*) − *real* y
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-mult* [*simp*]: *real*(x ∗ y) = *real* (x::*int*) ∗ *real* y
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-setsum* [*simp*]: *real* ((*SUM* x:*A*. f x)::*int*) = (*SUM* x:*A*. *real*(f x))
  apply (*subst real-eq-of-int*)+
  apply (*rule of-int-setsum*)
done

lemma *real-of-int-setprod* [*simp*]: *real* ((*PROD* x:*A*. f x)::*int*) =
    (*PROD* x:*A*. *real*(f x))
  apply (*subst real-eq-of-int*)+
  apply (*rule of-int-setprod*)
done

lemma *real-of-int-zero-cancel* [*simp*]: (*real* x = 0) = (x = (0::*int*))
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-inject* [*iff*]: (*real* (x::*int*) = *real* y) = (x = y)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-less-iff* [*iff*]: (*real* (x::*int*) < *real* y) = (x < y)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-le-iff* [*simp*]: (*real* (x::*int*) ≤ *real* y) = (x ≤ y)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-gt-zero-cancel-iff* [*simp*]: (0 < *real* (n::*int*)) = (0 < n)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-ge-zero-cancel-iff* [*simp*]: (0 <= *real* (n::*int*)) = (0 <= n)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-lt-zero-cancel-iff* [*simp*]: (*real* (n::*int*) < 0) = (n < 0)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-le-zero-cancel-iff* [*simp*]: (*real* (n::*int*) <= 0) = (n <= 0)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-abs* [*simp*]: *real* (*abs* x) = *abs*(*real* (x::*int*))

by (*auto simp add*: *abs-if*)

lemma *int-less-real-le*: ((*n*::*int*) < *m*) = (*real n* + *1* <= *real m*)
  apply (*subgoal-tac real n* + *1* = *real* (*n* + *1*))
  apply (*simp del*: *real-of-int-add*)
  apply *auto*
done

lemma *int-le-real-less*: ((*n*::*int*) <= *m*) = (*real n* < *real m* + *1*)
  apply (*subgoal-tac real m* + *1* = *real* (*m* + *1*))
  apply (*simp del*: *real-of-int-add*)
  apply *simp*
done

lemma *real-of-int-div-aux*: *d* ~= *0* ==> (*real* (*x*::*int*)) / (*real d*) =
    real (*x div d*) + (*real* (*x mod d*)) / (*real d*)
proof −
  assume *d* ~= *0*
  have *x* = (*x div d*) ∗ *d* + *x mod d*
    by *auto*
  then have *real x* = *real* (*x div d*) ∗ *real d* + *real*(*x mod d*)
    by (*simp only*: *real-of-int-mult* [*THEN sym*] *real-of-int-add* [*THEN sym*])
  then have *real x* / *real d* = ... / *real d*
    by *simp*
  then show *?thesis*
    by (*auto simp add*: *add-divide-distrib ring-simps prems*)
qed

lemma *real-of-int-div*: (*d*::*int*) ~= *0* ==> *d dvd n* ==>
    real(*n div d*) = *real n* / *real d*
  apply (*frule real-of-int-div-aux* [*of d n*])
  apply *simp*
  apply (*simp add*: *zdvd-iff-zmod-eq-0*)
done

lemma *real-of-int-div2*:
  0 <= *real* (*n*::*int*) / *real* (*x*) − *real* (*n div x*)
  apply (*case-tac x* = *0*)
  apply *simp*
  apply (*case-tac* 0 < *x*)
  apply (*simp add*: *compare-rls*)
  apply (*subst real-of-int-div-aux*)
  apply *simp*
  apply *simp*
  apply (*subst zero-le-divide-iff*)
  apply *auto*
  apply (*simp add*: *compare-rls*)
  apply (*subst real-of-int-div-aux*)
  apply *simp*

apply *simp*
apply (*subst zero-le-divide-iff*)
apply *auto*
done

lemma *real-of-int-div3*:
real (*n*::*int*) / *real* (*x*) − *real* (*n div x*) <= *1*
apply(*case-tac x = 0*)
apply *simp*
apply (*simp add*: *compare-rls*)
apply (*subst real-of-int-div-aux*)
apply *assumption*
apply *simp*
apply (*subst divide-le-eq*)
apply *clarsimp*
apply (*rule conjI*)
apply (*rule impI*)
apply (*rule order-less-imp-le*)
apply *simp*
apply (*rule impI*)
apply (*rule order-less-imp-le*)
apply *simp*
done

lemma *real-of-int-div4*: *real* (*n div x*) <= *real* (*n*::*int*) / *real x*
by (*insert real-of-int-div2* [*of n x*], *simp*)

11.11 Embedding the Naturals into the Reals

lemma *real-of-nat-zero* [*simp*]: *real* (*0*::*nat*) = *0*
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-one* [*simp*]: *real* (*Suc 0*) = (*1*::*real*)
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-add* [*simp*]: *real* (*m* + *n*) = *real* (*m*::*nat*) + *real n*
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-Suc*: *real* (*Suc n*) = *real n* + (*1*::*real*)
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-less-iff* [*iff*]:
(*real* (*n*::*nat*) < *real m*) = (*n* < *m*)
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-le-iff* [*iff*]: (*real* (*n*::*nat*) ≤ *real m*) = (*n* ≤ *m*)
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-ge-zero* [*iff*]: *0* ≤ *real* (*n*::*nat*)
by (*simp add*: *real-of-nat-def zero-le-imp-of-nat*)

lemma *real-of-nat-Suc-gt-zero*: *0* < *real* (*Suc n*)
by (*simp add*: *real-of-nat-def del*: *of-nat-Suc*)

lemma *real-of-nat-mult* [*simp*]: *real* (*m* * *n*) = *real* (*m*::*nat*) * *real n*
by (*simp add*: *real-of-nat-def of-nat-mult*)

lemma *real-of-nat-setsum* [*simp*]: *real* ((*SUM x*:*A*. *f x*)::*nat*) =
(*SUM x*:*A*. *real*(*f x*))
apply (*subst real-eq-of-nat*)+
apply (*rule of-nat-setsum*)
done

lemma *real-of-nat-setprod* [*simp*]: *real* ((*PROD x*:*A*. *f x*)::*nat*) =
(*PROD x*:*A*. *real*(*f x*))
apply (*subst real-eq-of-nat*)+
apply (*rule of-nat-setprod*)
done

lemma *real-of-card*: *real* (*card A*) = *setsum* (%*x*.*1*) *A*
apply (*subst card-eq-setsum*)
apply (*subst real-of-nat-setsum*)
apply *simp*
done

lemma *real-of-nat-inject* [*iff*]: (*real* (*n*::*nat*) = *real m*) = (*n* = *m*)
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-zero-iff* [*iff*]: (*real* (*n*::*nat*) = *0*) = (*n* = *0*)
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-diff*: *n* ≤ *m* ==> *real* (*m* − *n*) = *real* (*m*::*nat*) − *real n*
by (*simp add*: *add*: *real-of-nat-def of-nat-diff*)

lemma *real-of-nat-gt-zero-cancel-iff* [*simp*]: (*0* < *real* (*n*::*nat*)) = (*0* < *n*)
by (*auto simp*: *real-of-nat-def*)

lemma *real-of-nat-le-zero-cancel-iff* [*simp*]: (*real* (*n*::*nat*) ≤ *0*) = (*n* = *0*)
by (*simp add*: *add*: *real-of-nat-def*)

lemma *not-real-of-nat-less-zero* [*simp*]: ~ *real* (*n*::*nat*) < *0*
by (*simp add*: *add*: *real-of-nat-def*)

lemma *real-of-nat-ge-zero-cancel-iff* [*simp*]: (*0* ≤ *real* (*n*::*nat*))
by (*simp add*: *add*: *real-of-nat-def*)

lemma *nat-less-real-le*: ((*n*::*nat*) < *m*) = (*real n* + *1* <= *real m*)

```
  apply (subgoal-tac real n + 1 = real (Suc n))
  apply simp
  apply (auto simp add: real-of-nat-Suc)
done

lemma nat-le-real-less: ((n::nat) <= m) = (real n < real m + 1)
  apply (subgoal-tac real m + 1 = real (Suc m))
  apply (simp add: less-Suc-eq-le)
  apply (simp add: real-of-nat-Suc)
done

lemma real-of-nat-div-aux: 0 < d ==> (real (x::nat)) / (real d) =
    real (x div d) + (real (x mod d)) / (real d)
proof -
  assume 0 < d
  have x = (x div d) * d + x mod d
    by auto
  then have real x = real (x div d) * real d + real(x mod d)
    by (simp only: real-of-nat-mult [THEN sym] real-of-nat-add [THEN sym])
  then have real x / real d = ... / real d
    by simp
  then show ?thesis
    by (auto simp add: add-divide-distrib ring-simps prems)
qed

lemma real-of-nat-div: 0 < (d::nat) ==> d dvd n ==>
    real(n div d) = real n / real d
  apply (frule real-of-nat-div-aux [of d n])
  apply simp
  apply (subst dvd-eq-mod-eq-0 [THEN sym])
  apply assumption
done

lemma real-of-nat-div2:
  0 <= real (n::nat) / real (x) - real (n div x)
apply(case-tac x = 0)
 apply (simp)
apply (simp add: compare-rls)
apply (subst real-of-nat-div-aux)
 apply simp
apply simp
apply (subst zero-le-divide-iff)
apply simp
done

lemma real-of-nat-div3:
  real (n::nat) / real (x) - real (n div x) <= 1
apply(case-tac x = 0)
apply (simp)
```

apply (*simp add*: *compare-rls*)
apply (*subst real-of-nat-div-aux*)
apply *simp*
apply *simp*
done

lemma *real-of-nat-div4*: *real* (*n div x*) <= *real* (*n*::*nat*) / *real x*
by (*insert real-of-nat-div2* [*of n x*], *simp*)

lemma *real-of-int-real-of-nat*: *real* (*int n*) = *real n*
by (*simp add*: *real-of-nat-def real-of-int-def int-eq-of-nat*)

lemma *real-of-int-of-nat-eq* [*simp*]: *real* (*of-nat n* :: *int*) = *real n*
by (*simp add*: *real-of-int-def real-of-nat-def*)

lemma *real-nat-eq-real* [*simp*]: *0* <= *x* ==> *real*(*nat x*) = *real x*
apply (*subgoal-tac real*(*int*(*nat x*)) = *real*(*nat x*))
apply *force*
apply (*simp only*: *real-of-int-real-of-nat*)
done

11.12 Numerals and Arithmetic

instance *real* :: *number-ring*
real-number-of-def: *number-of w* ≡ *real-of-int w*
by *intro-classes* (*simp add*: *real-number-of-def*)

lemma [*code*, *code unfold*]:
number-of k = *real-of-int* (*number-of k*)
unfolding *number-of-is-id real-number-of-def* **..**

Collapse applications of *real* to *number-of*

lemma *real-number-of* [*simp*]: *real* (*number-of v* :: *int*) = *number-of v*
by (*simp add*: *real-of-int-def of-int-number-of-eq*)

lemma *real-of-nat-number-of* [*simp*]:
real (*number-of v* :: *nat*) =
(*if neg* (*number-of v* :: *int*) *then 0*
else (*number-of v* :: *real*))
by (*simp add*: *real-of-int-real-of-nat* [*symmetric*] *int-nat-number-of*)

use *real-arith.ML*
declaration ⟪ *K real-arith-setup* ⟫

11.13 Simprules combining x+y and 0: ARE THEY NEEDED?

Needed in this non-standard form by Hyperreal/Transcendental

lemma *real-0-le-divide-iff*:

$((0::real) \leq x/y) = ((x \leq 0 \mid 0 \leq y) \;\&\; (0 \leq x \mid y \leq 0))$
by (*simp add*: *real-divide-def zero-le-mult-iff*, *auto*)

lemma *real-add-minus-iff* [*simp*]: $(x + -a = (0::real)) = (x=a)$
by *arith*

lemma *real-add-eq-0-iff*: $(x+y = (0::real)) = (y = -x)$
by *auto*

lemma *real-add-less-0-iff*: $(x+y < (0::real)) = (y < -x)$
by *auto*

lemma *real-0-less-add-iff*: $((0::real) < x+y) = (-x < y)$
by *auto*

lemma *real-add-le-0-iff*: $(x+y \leq (0::real)) = (y \leq -x)$
by *auto*

lemma *real-0-le-add-iff*: $((0::real) \leq x+y) = (-x \leq y)$
by *auto*

11.13.1 Density of the Reals

lemma *real-lbound-gt-zero*:
$[\!| (0::real) < d1; 0 < d2 |\!] ==> \exists e.\ 0 < e \;\&\; e < d1 \;\&\; e < d2$
apply (*rule-tac x* = $(min\ d1\ d2)\ /2$ **in** *exI*)
apply (*simp add*: *min-def*)
done

Similar results are proved in *Ring-and-Field*

lemma *real-less-half-sum*: $x < y ==> x < (x+y) / (2::real)$
by *auto*

lemma *real-gt-half-sum*: $x < y ==> (x+y)/(2::real) < y$
by *auto*

11.14 Absolute Value Function for the Reals

lemma *abs-minus-add-cancel*: $abs(x + (-y)) = abs\ (y + (-(x::real)))$
by (*simp add*: *abs-if*)

lemma *abs-le-interval-iff*: $(abs\ x \leq r) = (-r \leq x \;\&\; x \leq (r::real))$
by (*force simp add*: *OrderedGroup.abs-le-iff*)

lemma *abs-add-one-gt-zero* [*simp*]: $(0::real) < 1 + abs(x)$
by (*simp add*: *abs-if*)

lemma *abs-real-of-nat-cancel* [*simp*]: $abs\ (real\ x) = real\ (x::nat)$

by (*rule abs-of-nonneg* [*OF real-of-nat-ge-zero*])

lemma *abs-add-one-not-less-self* [*simp*]: ~ *abs*(*x*) + (*1*::*real*) < *x*
by *simp*

lemma *abs-sum-triangle-ineq*: *abs* ((*x*::*real*) + *y* + (−*l* + −*m*)) ≤ *abs*(*x* + −*l*) + *abs*(*y* + −*m*)
by *simp*

11.15 Implementation of rational real numbers as pairs of integers

definition
Ratreal :: *int* × *int* ⇒ *real*
where
Ratreal = *INum*

code-datatype *Ratreal*

lemma *Ratreal-simp*:
Ratreal (*k*, *l*) = *real-of-int k* / *real-of-int l*
unfolding *Ratreal-def INum-def* **by** *simp*

lemma *Ratreal-zero* [*simp*]: *Ratreal* 0_N = *0*
by (*simp add*: *Ratreal-simp*)

lemma *Ratreal-lit* [*simp*]: *Ratreal* i_N = *real-of-int i*
by (*simp add*: *Ratreal-simp*)

lemma *zero-real-code* [*code*, *code unfold*]:
0 = *Ratreal* 0_N **by** *simp*

lemma *one-real-code* [*code*, *code unfold*]:
1 = *Ratreal* 1_N **by** *simp*

instance *real* :: *eq* **..**

lemma *real-eq-code* [*code*]: *Ratreal x* = *Ratreal y* ⟷ *normNum x* = *normNum y*
unfolding *Ratreal-def INum-normNum-iff* **..**

lemma *real-less-eq-code* [*code*]: *Ratreal x* ≤ *Ratreal y* ⟷ *normNum x* $\leq_N$ *normNum y*
proof −
have *normNum x* $\leq_N$ *normNum y* ⟷ *Ratreal* (*normNum x*) ≤ *Ratreal* (*normNum y*)
by (*simp add*: *Ratreal-def del*: *normNum*)
also have ... = (*Ratreal x* ≤ *Ratreal y*) **by** (*simp add*: *Ratreal-def*)
finally show *?thesis* **by** *simp*
qed

lemma *real-less-code* [*code*]: *Ratreal x* < *Ratreal y* ⟷ *normNum x* <$_N$ *normNum y*
proof −
have *normNum x* <$_N$ *normNum y* ⟷ *Ratreal* (*normNum x*) < *Ratreal* (*normNum y*)
by (*simp add*: *Ratreal-def del*: *normNum*)
also have ... = (*Ratreal x* < *Ratreal y*) **by** (*simp add*: *Ratreal-def*)
finally show *?thesis* **by** *simp*
qed

lemma *real-add-code* [*code*]: *Ratreal x* + *Ratreal y* = *Ratreal* (*x* +$_N$ *y*)
unfolding *Ratreal-def* **by** *simp*

lemma *real-mul-code* [*code*]: *Ratreal x* ∗ *Ratreal y* = *Ratreal* (*x* ∗$_N$ *y*)
unfolding *Ratreal-def* **by** *simp*

lemma *real-neg-code* [*code*]: − *Ratreal x* = *Ratreal* (~$_N$ *x*)
unfolding *Ratreal-def* **by** *simp*

lemma *real-sub-code* [*code*]: *Ratreal x* − *Ratreal y* = *Ratreal* (*x* −$_N$ *y*)
unfolding *Ratreal-def* **by** *simp*

lemma *real-inv-code* [*code*]: *inverse* (*Ratreal x*) = *Ratreal* (*Ninv x*)
unfolding *Ratreal-def Ninv real-divide-def* **by** *simp*

lemma *real-div-code* [*code*]: *Ratreal x* / *Ratreal y* = *Ratreal* (*x* ÷$_N$ *y*)
unfolding *Ratreal-def* **by** *simp*

Setup for SML code generator

types-code
real ((*int* ∗/ *int*))
attach (*term-of*) ⟪
fun term-of-real (*p*, *q*) =
let
val rT = *HOLogic.realT*
in
if q = *1 orelse p* = *0 then HOLogic.mk-number rT p*
else @{*term op* / :: *real* ⇒ *real* ⇒ *real*} $
HOLogic.mk-number rT p $ *HOLogic.mk-number rT q*
end;
⟫
attach (*test*) ⟪
fun gen-real i =
let
val p = *random-range 0 i*;
val q = *random-range 1* (*i* + *1*);
val g = *Integer.gcd p q*;
val p′ = *p div g*;

```
    val q' = q div g;
  in
    (if one-of [true, false] then p' else ~ p',
     if p' = 0 then 0 else q')
  end;
⟩⟩

consts-code
  Ratreal ((-))

consts-code
  of-int :: int ⇒ real (⟨module⟩real'-of'-int)
attach ⟪
fun real-of-int 0 = (0, 0)
  | real-of-int i = (i, 1);
⟫

declare real-of-int-of-nat-eq [symmetric, code]

end
```

12 Completeness of the Reals; Floor and Ceiling Functions

```
theory RComplete
imports Lubs RealDef
begin

lemma real-sum-of-halves: x/2 + x/2 = (x::real)
  by simp
```

12.1 Completeness of Positive Reals

Supremum property for the set of positive reals

Let P be a non-empty set of positive reals, with an upper bound y. Then P has a least upper bound (written S).

FIXME: Can the premise be weakened to $\forall x \in P.\ x \leq y$?

```
lemma posreal-complete:
  assumes positive-P: ∀ x ∈ P. (0::real) < x
    and not-empty-P: ∃ x. x ∈ P
    and upper-bound-Ex: ∃ y. ∀ x ∈ P. x<y
  shows ∃ S. ∀ y. (∃ x ∈ P. y < x) = (y < S)
proof (rule exI, rule allI)
  fix y
  let ?pP = {w. real-of-preal w ∈ P}
```

```
show (∃ x∈P. y < x) = (y < real-of-preal (psup ?pP))
proof (cases 0 < y)
  assume neg-y: ¬ 0 < y
  show ?thesis
  proof
    assume ∃ x∈P. y < x
    have ∀ x. y < real-of-preal x
      using neg-y by (rule real-less-all-real2)
    thus y < real-of-preal (psup ?pP) ..
  next
    assume y < real-of-preal (psup ?pP)
    obtain x where x-in-P: x ∈ P using not-empty-P ..
    hence 0 < x using positive-P by simp
    hence y < x using neg-y by simp
    thus ∃ x ∈ P. y < x using x-in-P ..
  qed
next
  assume pos-y: 0 < y

  then obtain py where y-is-py: y = real-of-preal py
    by (auto simp add: real-gt-zero-preal-Ex)

  obtain a where a ∈ P using not-empty-P ..
  with positive-P have a-pos: 0 < a ..
  then obtain pa where a = real-of-preal pa
    by (auto simp add: real-gt-zero-preal-Ex)
  hence pa ∈ ?pP using ‹a ∈ P› by auto
  hence pP-not-empty: ?pP ≠ {} by auto

  obtain sup where sup: ∀ x ∈ P. x < sup
    using upper-bound-Ex ..
  from this and ‹a ∈ P› have a < sup ..
  hence 0 < sup using a-pos by arith
  then obtain possup where sup = real-of-preal possup
    by (auto simp add: real-gt-zero-preal-Ex)
  hence ∀ X ∈ ?pP. X ≤ possup
    using sup by (auto simp add: real-of-preal-lessI)
  with pP-not-empty have psup: ⋀Z. (∃ X ∈ ?pP. Z < X) = (Z < psup ?pP)
    by (rule preal-complete)

  show ?thesis
  proof
    assume ∃ x ∈ P. y < x
    then obtain x where x-in-P: x ∈ P and y-less-x: y < x ..
    hence 0 < x using pos-y by arith
    then obtain px where x-is-px: x = real-of-preal px
      by (auto simp add: real-gt-zero-preal-Ex)

    have py-less-X: ∃ X ∈ ?pP. py < X
```

```
    proof
      show py < px using y-is-py and x-is-px and y-less-x
        by (simp add: real-of-preal-lessI)
      show px ∈ ?pP using x-in-P and x-is-px by simp
    qed

    have (∃ X ∈ ?pP. py < X) ==> (py < psup ?pP)
      using psup by simp
    hence py < psup ?pP using py-less-X by simp
    thus y < real-of-preal (psup {w. real-of-preal w ∈ P})
      using y-is-py and pos-y by (simp add: real-of-preal-lessI)
  next
    assume y-less-psup: y < real-of-preal (psup ?pP)

    hence py < psup ?pP using y-is-py
      by (simp add: real-of-preal-lessI)
    then obtain X where py-less-X: py < X and X-in-pP: X ∈ ?pP
      using psup by auto
    then obtain x where x-is-X: x = real-of-preal X
      by (simp add: real-gt-zero-preal-Ex)
    hence y < x using py-less-X and y-is-py
      by (simp add: real-of-preal-lessI)

    moreover have x ∈ P using x-is-X and X-in-pP by simp

    ultimately show ∃ x ∈ P. y < x ..
  qed
 qed
qed
```

Completeness properties using *isUb*, *isLub* etc.

```
lemma real-isLub-unique: [| isLub R S x; isLub R S y |] ==> x = (y::real)
 apply (frule isLub-isUb)
 apply (frule-tac x = y in isLub-isUb)
 apply (blast intro!: order-antisym dest!: isLub-le-isUb)
 done
```

Completeness theorem for the positive reals (again).

```
lemma posreals-complete:
 assumes positive-S: ∀ x ∈ S. 0 < x
   and not-empty-S: ∃ x. x ∈ S
   and upper-bound-Ex: ∃ u. isUb (UNIV::real set) S u
 shows ∃ t. isLub (UNIV::real set) S t
proof
 let ?pS = {w. real-of-preal w ∈ S}

 obtain u where isUb UNIV S u using upper-bound-Ex ..
 hence sup: ∀ x ∈ S. x ≤ u by (simp add: isUb-def setle-def)
```

obtain x **where** *x-in-S*: $x \in S$ **using** *not-empty-S* **..**
hence *x-gt-zero*: $0 < x$ **using** *positive-S* **by** *simp*
have $x \leq u$ **using** *sup* **and** *x-in-S* **..**
hence $0 < u$ **using** *x-gt-zero* **by** *arith*

then obtain *pu* **where** *u-is-pu*: $u = real\text{-}of\text{-}preal\ pu$
by (*auto simp add*: *real-gt-zero-preal-Ex*)

have *pS-less-pu*: $\forall\, pa \in ?pS.\ pa \leq pu$
proof
fix *pa*
assume $pa \in ?pS$
then obtain a **where** $a \in S$ **and** $a = real\text{-}of\text{-}preal\ pa$
by *simp*
moreover hence $a \leq u$ **using** *sup* **by** *simp*
ultimately show $pa \leq pu$
using *sup* **and** *u-is-pu* **by** (*simp add*: *real-of-preal-le-iff*)
qed

have $\forall\, y \in S.\ y \leq real\text{-}of\text{-}preal\ (psup\ ?pS)$
proof
fix y
assume *y-in-S*: $y \in S$
hence $0 < y$ **using** *positive-S* **by** *simp*
then obtain *py* **where** *y-is-py*: $y = real\text{-}of\text{-}preal\ py$
by (*auto simp add*: *real-gt-zero-preal-Ex*)
hence *py-in-pS*: $py \in ?pS$ **using** *y-in-S* **by** *simp*
with *pS-less-pu* **have** $py \leq psup\ ?pS$
by (*rule preal-psup-le*)
thus $y \leq real\text{-}of\text{-}preal\ (psup\ ?pS)$
using *y-is-py* **by** (*simp add*: *real-of-preal-le-iff*)
qed

moreover {
fix x
assume *x-ub-S*: $\forall\, y{\in}S.\ y \leq x$
have $real\text{-}of\text{-}preal\ (psup\ ?pS) \leq x$
proof −
obtain s **where** *s-in-S*: $s \in S$ **using** *not-empty-S* **..**
hence *s-pos*: $0 < s$ **using** *positive-S* **by** *simp*

hence $\exists\ ps.\ s = real\text{-}of\text{-}preal\ ps$ **by** (*simp add*: *real-gt-zero-preal-Ex*)
then obtain *ps* **where** *s-is-ps*: $s = real\text{-}of\text{-}preal\ ps$ **..**
hence *ps-in-pS*: $ps \in \{w.\ real\text{-}of\text{-}preal\ w \in S\}$ **using** *s-in-S* **by** *simp*

from *x-ub-S* **have** $s \leq x$ **using** *s-in-S* **..**
hence $0 < x$ **using** *s-pos* **by** *simp*
hence $\exists\ px.\ x = real\text{-}of\text{-}preal\ px$ **by** (*simp add*: *real-gt-zero-preal-Ex*)

```
    then obtain px where x-is-px: x = real-of-preal px ..

    have ∀ pe ∈ ?pS. pe ≤ px
    proof
      fix pe
      assume pe ∈ ?pS
      hence real-of-preal pe ∈ S by simp
      hence real-of-preal pe ≤ x using x-ub-S by simp
      thus pe ≤ px using x-is-px by (simp add: real-of-preal-le-iff)
    qed

    moreover have ?pS ≠ {} using ps-in-pS by auto
    ultimately have (psup ?pS) ≤ px by (simp add: psup-le-ub)
   thus real-of-preal (psup ?pS) ≤ x using x-is-px by (simp add: real-of-preal-le-iff)
   qed
  }
  ultimately show isLub UNIV S (real-of-preal (psup ?pS))
    by (simp add: isLub-def leastP-def isUb-def setle-def setge-def)
qed
```

reals Completeness (again!)

```
lemma reals-complete:
  assumes notempty-S: ∃ X. X ∈ S
    and exists-Ub: ∃ Y. isUb (UNIV::real set) S Y
  shows ∃ t. isLub (UNIV :: real set) S t
proof −
  obtain X where X-in-S: X ∈ S using notempty-S ..
  obtain Y where Y-isUb: isUb (UNIV::real set) S Y
    using exists-Ub ..
  let ?SHIFT = {z. ∃ x ∈S. z = x + (−X) + 1} ∩ {x. 0 < x}

  {
    fix x
    assume isUb (UNIV::real set) S x
    hence S-le-x: ∀ y ∈ S. y <= x
      by (simp add: isUb-def setle-def)
    {
      fix s
      assume s ∈ {z. ∃ x∈S. z = x + − X + 1}
      hence ∃ x ∈ S. s = x + −X + 1 ..
      then obtain x1 where x1 ∈ S and s = x1 + (−X) + 1 ..
      moreover hence x1 ≤ x using S-le-x by simp
      ultimately have s ≤ x + − X + 1 by arith
    }
    then have isUb (UNIV::real set) ?SHIFT (x + (−X) + 1)
      by (auto simp add: isUb-def setle-def)
  } note S-Ub-is-SHIFT-Ub = this

  hence isUb UNIV ?SHIFT (Y + (−X) + 1) using Y-isUb by simp
```

```
hence ∃ Z. isUb UNIV ?SHIFT Z ..
moreover have ∀ y ∈ ?SHIFT. 0 < y by auto
moreover have shifted-not-empty: ∃ u. u ∈ ?SHIFT
  using X-in-S and Y-isUb by auto
ultimately obtain t where t-is-Lub: isLub UNIV ?SHIFT t
  using posreals-complete [of ?SHIFT] by blast

show ?thesis
proof
  show isLub UNIV S (t + X + (−1))
  proof (rule isLubI2)
    {
      fix x
      assume isUb (UNIV::real set) S x
      hence isUb (UNIV::real set) (?SHIFT) (x + (−X) + 1)
        using S-Ub-is-SHIFT-Ub by simp
      hence t ≤ (x + (−X) + 1)
        using t-is-Lub by (simp add: isLub-le-isUb)
      hence t + X + −1 ≤ x by arith
    }
    then show (t + X + −1) <=* Collect (isUb UNIV S)
      by (simp add: setgeI)
  next
    show isUb UNIV S (t + X + −1)
    proof −
      {
        fix y
        assume y-in-S: y ∈ S
        have y ≤ t + X + −1
        proof −
          obtain u where u-in-shift: u ∈ ?SHIFT using shifted-not-empty ..
          hence ∃ x ∈ S. u = x + − X + 1 by simp
          then obtain x where x-and-u: u = x + − X + 1 ..
         have u-le-t: u ≤ t using u-in-shift and t-is-Lub by (simp add: isLubD2)

          show ?thesis
          proof cases
            assume y ≤ x
            moreover have x = u + X + − 1 using x-and-u by arith
            moreover have u + X + − 1 ≤ t + X + −1 using u-le-t by arith
            ultimately show y ≤ t + X + −1 by arith
          next
            assume ~(y ≤ x)
            hence x-less-y: x < y by arith

           have x + (−X) + 1 ∈ ?SHIFT using x-and-u and u-in-shift by simp
            hence 0 < x + (−X) + 1 by simp
            hence 0 < y + (−X) + 1 using x-less-y by arith
            hence y + (−X) + 1 ∈ ?SHIFT using y-in-S by simp
```

```
          hence y + (−X) + 1 ≤ t using t-is-Lub  by (simp add: isLubD2)
          thus ?thesis by simp
        qed
      qed
    }
    then show ?thesis by (simp add: isUb-def setle-def)
  qed
 qed
qed
```

12.2 The Archimedean Property of the Reals

```
theorem reals-Archimedean:
 assumes x-pos: 0 < x
 shows ∃ n. inverse (real (Suc n)) < x
proof (rule ccontr)
 assume contr: ¬ ?thesis
 have ∀ n. x * real (Suc n) <= 1
 proof
  fix n
  from contr have x ≤ inverse (real (Suc n))
   by (simp add: linorder-not-less)
  hence x ≤ (1 / (real (Suc n)))
   by (simp add: inverse-eq-divide)
  moreover have 0 ≤ real (Suc n)
   by (rule real-of-nat-ge-zero)
  ultimately have x * real (Suc n) ≤ (1 / real (Suc n)) * real (Suc n)
   by (rule mult-right-mono)
  thus x * real (Suc n) ≤ 1 by simp
 qed
 hence {z. ∃ n. z = x * (real (Suc n))} *<= 1
  by (simp add: setle-def, safe, rule spec)
 hence isUb (UNIV::real set) {z. ∃ n. z = x * (real (Suc n))} 1
  by (simp add: isUbI)
 hence ∃ Y. isUb (UNIV::real set) {z. ∃ n. z = x* (real (Suc n))} Y ..
 moreover have ∃ X. X ∈ {z. ∃ n. z = x* (real (Suc n))} by auto
 ultimately have ∃ t. isLub UNIV {z. ∃ n. z = x * real (Suc n)} t
  by (simp add: reals-complete)
 then obtain t where
  t-is-Lub: isLub UNIV {z. ∃ n. z = x * real (Suc n)} t ..

 have ∀ n::nat. x * real n ≤ t + − x
 proof
  fix n
  from t-is-Lub have x * real (Suc n) ≤ t
   by (simp add: isLubD2)
  hence  x * (real n) + x ≤ t
   by (simp add: right-distrib real-of-nat-Suc)
```

```
    thus  x * (real n) ≤ t + - x by arith
  qed

  hence ∀ m. x * real (Suc m) ≤ t + - x by simp
  hence {z. ∃ n. z = x * (real (Suc n))}  *<= (t + - x)
    by (auto simp add: setle-def)
  hence isUb (UNIV::real set) {z. ∃ n. z = x * (real (Suc n))} (t + (-x))
    by (simp add: isUbI)
  hence t ≤ t + - x
    using t-is-Lub by (simp add: isLub-le-isUb)
  thus False using x-pos by arith
qed
```

There must be other proofs, e.g. *Suc* of the largest integer in the cut representing x.

```
lemma reals-Archimedean2: ∃ n. (x::real) < real (n::nat)
proof cases
  assume x ≤ 0
  hence x < real (1::nat) by simp
  thus ?thesis ..
next
  assume ¬ x ≤ 0
  hence x-greater-zero: 0 < x by simp
  hence 0 < inverse x by simp
  then obtain n where inverse (real (Suc n)) < inverse x
    using reals-Archimedean by blast
  hence inverse (real (Suc n)) * x < inverse x * x
    using x-greater-zero by (rule mult-strict-right-mono)
  hence inverse (real (Suc n)) * x < 1
    using x-greater-zero by simp
  hence real (Suc n) * (inverse (real (Suc n)) * x) < real (Suc n) * 1
    by (rule mult-strict-left-mono) simp
  hence x < real (Suc n)
    by (simp add: ring-simps)
  thus ∃ (n::nat). x < real n ..
qed

lemma reals-Archimedean3:
  assumes x-greater-zero: 0 < x
  shows ∀ (y::real). ∃ (n::nat). y < real n * x
proof
  fix y
  have x-not-zero: x ≠ 0 using x-greater-zero by simp
  obtain n where y * inverse x < real (n::nat)
    using reals-Archimedean2 ..
  hence y * inverse x * x < real n * x
    using x-greater-zero by (simp add: mult-strict-right-mono)
  hence x * inverse x * y < x * real n
    by (simp add: ring-simps)
```

hence $y < real\ (n::nat) * x$
using *x-not-zero* **by** (*simp add*: *ring-simps*)
thus $\exists (n::nat).\ y < real\ n * x$ **..**
qed

lemma *reals-Archimedean6*:
$0 \leq r ==> \exists (n::nat).\ real\ (n - 1) \leq r\ \&\ r < real\ (n)$
apply (*insert reals-Archimedean2* [*of r*], *safe*)
apply (*subgoal-tac* $\exists x::nat.\ r < real\ x \wedge (\forall y.\ r < real\ y \longrightarrow x \leq y)$, *auto*)
apply (*rule-tac* $x = x$ **in** *exI*)
apply (*case-tac x*, *simp*)
apply (*rename-tac* x')
apply (*drule-tac* $x = x'$ **in** *spec*, *simp*)
apply (*rule-tac x=LEAST n.* $r < real\ n$ **in** *exI*, *safe*)
apply (*erule LeastI*, *erule Least-le*)
done

lemma *reals-Archimedean6a*: $0 \leq r ==> \exists n.\ real\ (n) \leq r\ \&\ r < real\ (Suc\ n)$
by (*drule reals-Archimedean6*) *auto*

lemma *reals-Archimedean-6b-int*:
$0 \leq r ==> \exists n::int.\ real\ n \leq r\ \&\ r < real\ (n+1)$
apply (*drule reals-Archimedean6a*, *auto*)
apply (*rule-tac* $x = int\ n$ **in** *exI*)
apply (*simp add*: *real-of-int-real-of-nat real-of-nat-Suc*)
done

lemma *reals-Archimedean-6c-int*:
$r < 0 ==> \exists n::int.\ real\ n \leq r\ \&\ r < real\ (n+1)$
apply (*rule reals-Archimedean-6b-int* [*of* $-r$, *THEN exE*], *simp*, *auto*)
apply (*rename-tac n*)
apply (*drule order-le-imp-less-or-eq*, *auto*)
apply (*rule-tac* $x = -\ n - 1$ **in** *exI*)
apply (*rule-tac* [2] $x = -\ n$ **in** *exI*, *auto*)
done

12.3 Floor and Ceiling Functions from the Reals to the Integers

definition
floor :: *real* => *int* **where**
floor r = (*LEAST n::int.* $r < real\ (n+1)$)

definition
ceiling :: *real* => *int* **where**
ceiling r = $-$ *floor* $(-\ r)$

notation (*xsymbols*)
floor ($\lfloor$-$\rfloor$) **and**

ceiling (⌈-⌉)

notation (*HTML* **output**)
floor (⌊-⌋) **and**
ceiling (⌈-⌉)

lemma *number-of-less-real-of-int-iff* [*simp*]:
((*number-of n*) < *real* (*m*::*int*)) = (*number-of n* < *m*)
apply *auto*
apply (*rule real-of-int-less-iff* [*THEN iffD1*])
apply (*drule-tac* [*2*] *real-of-int-less-iff* [*THEN iffD2*], *auto*)
done

lemma *number-of-less-real-of-int-iff2* [*simp*]:
(*real* (*m*::*int*) < (*number-of n*)) = (*m* < *number-of n*)
apply *auto*
apply (*rule real-of-int-less-iff* [*THEN iffD1*])
apply (*drule-tac* [*2*] *real-of-int-less-iff* [*THEN iffD2*], *auto*)
done

lemma *number-of-le-real-of-int-iff* [*simp*]:
((*number-of n*) ≤ *real* (*m*::*int*)) = (*number-of n* ≤ *m*)
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *number-of-le-real-of-int-iff2* [*simp*]:
(*real* (*m*::*int*) ≤ (*number-of n*)) = (*m* ≤ *number-of n*)
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *floor-zero* [*simp*]: *floor 0* = *0*
apply (*simp add*: *floor-def del*: *real-of-int-add*)
apply (*rule Least-equality*)
apply *simp-all*
done

lemma *floor-real-of-nat-zero* [*simp*]: *floor* (*real* (*0*::*nat*)) = *0*
by *auto*

lemma *floor-real-of-nat* [*simp*]: *floor* (*real* (*n*::*nat*)) = *int n*
apply (*simp only*: *floor-def*)
apply (*rule Least-equality*)
apply (*drule-tac* [*2*] *real-of-int-of-nat-eq* [*THEN ssubst*])
apply (*drule-tac* [*2*] *real-of-int-less-iff* [*THEN iffD1*])
apply *simp-all*
done

lemma *floor-minus-real-of-nat* [*simp*]: *floor* (− *real* (*n*::*nat*)) = − *int n*
apply (*simp only*: *floor-def*)
apply (*rule Least-equality*)

apply (*drule-tac* [*2*] *real-of-int-of-nat-eq* [*THEN ssubst*])
apply (*drule-tac* [*2*] *real-of-int-minus* [*THEN sym*, *THEN subst*])
apply (*drule-tac* [*2*] *real-of-int-less-iff* [*THEN iffD1*])
apply *simp-all*
done

lemma *floor-real-of-int* [*simp*]: *floor* (*real* (*n*::*int*)) = *n*
apply (*simp only*: *floor-def*)
apply (*rule Least-equality*)
apply *auto*
done

lemma *floor-minus-real-of-int* [*simp*]: *floor* (− *real* (*n*::*int*)) = − *n*
apply (*simp only*: *floor-def*)
apply (*rule Least-equality*)
apply (*drule-tac* [*2*] *real-of-int-minus* [*THEN sym*, *THEN subst*])
apply *auto*
done

lemma *real-lb-ub-int*: ∃ *n*::*int*. *real n* ≤ *r* & *r* < *real* (*n*+*1*)
apply (*case-tac r* < *0*)
apply (*blast intro*: *reals-Archimedean-6c-int*)
apply (*simp only*: *linorder-not-less*)
apply (*blast intro*: *reals-Archimedean-6b-int reals-Archimedean-6c-int*)
done

lemma *lemma-floor*:
  assumes *a1*: *real m* ≤ *r* **and** *a2*: *r* < *real n* + *1*
  shows *m* ≤ (*n*::*int*)
proof −
  have *real m* < *real n* + *1* **using** *a1 a2* **by** (*rule order-le-less-trans*)
  also have ... = *real* (*n* + *1*) **by** *simp*
  finally have *m* < *n* + *1* **by** (*simp only*: *real-of-int-less-iff*)
  thus *?thesis* **by** *arith*
qed

lemma *real-of-int-floor-le* [*simp*]: *real* (*floor r*) ≤ *r*
apply (*simp add*: *floor-def Least-def*)
apply (*insert real-lb-ub-int* [*of r*], *safe*)
apply (*rule theI2*)
apply *auto*
done

lemma *floor-mono*: *x* < *y* ==> *floor x* ≤ *floor y*
apply (*simp add*: *floor-def Least-def*)
apply (*insert real-lb-ub-int* [*of x*])
apply (*insert real-lb-ub-int* [*of y*], *safe*)
apply (*rule theI2*)
apply (*rule-tac* [*3*] *theI2*)

apply *simp*
apply (*erule conjI*)
apply (*auto simp add*: *order-eq-iff int-le-real-less*)
done

lemma *floor-mono2*: $x \leq y \Longrightarrow$ *floor* $x \leq$ *floor* y
by (*auto dest*: *order-le-imp-less-or-eq simp add*: *floor-mono*)

lemma *lemma-floor2*: *real* $n <$ *real* (x::*int*) + *1* $\Longrightarrow n \leq x$
by (*auto intro*: *lemma-floor*)

lemma *real-of-int-floor-cancel* [*simp*]:
  (*real* (*floor* x) $= x$) = ($\exists\, n$::*int*. $x =$ *real* n)
apply (*simp add*: *floor-def Least-def*)
apply (*insert real-lb-ub-int* [*of* x], *erule exE*)
apply (*rule theI2*)
apply (*auto intro*: *lemma-floor*)
done

lemma *floor-eq*: [| *real* $n < x$; $x <$ *real* n + *1* |] $\Longrightarrow$ *floor* $x = n$
apply (*simp add*: *floor-def*)
apply (*rule Least-equality*)
apply (*auto intro*: *lemma-floor*)
done

lemma *floor-eq2*: [| *real* $n \leq x$; $x <$ *real* n + *1* |] $\Longrightarrow$ *floor* $x = n$
apply (*simp add*: *floor-def*)
apply (*rule Least-equality*)
apply (*auto intro*: *lemma-floor*)
done

lemma *floor-eq3*: [| *real* $n < x$; $x <$ *real* (*Suc* n) |] $\Longrightarrow$ *nat*(*floor* x) $= n$
apply (*rule inj-int* [*THEN injD*])
apply (*simp add*: *real-of-nat-Suc*)
apply (*simp add*: *real-of-nat-Suc floor-eq floor-eq* [**where** $n =$ *int* n])
done

lemma *floor-eq4*: [| *real* $n \leq x$; $x <$ *real* (*Suc* n) |] $\Longrightarrow$ *nat*(*floor* x) $= n$
apply (*drule order-le-imp-less-or-eq*)
apply (*auto intro*: *floor-eq3*)
done

lemma *floor-number-of-eq* [*simp*]:
  floor(*number-of* n :: *real*) = (*number-of* n :: *int*)
apply (*subst real-number-of* [*symmetric*])
apply (*rule floor-real-of-int*)
done

lemma *floor-one* [*simp*]: *floor 1 = 1*

```
  apply (rule trans)
  prefer 2
  apply (rule floor-real-of-int)
  apply simp
done

lemma real-of-int-floor-ge-diff-one [simp]: r - 1 ≤ real(floor r)
apply (simp add: floor-def Least-def)
apply (insert real-lb-ub-int [of r], safe)
apply (rule theI2)
apply (auto intro: lemma-floor)
done

lemma real-of-int-floor-gt-diff-one [simp]: r - 1 < real(floor r)
apply (simp add: floor-def Least-def)
apply (insert real-lb-ub-int [of r], safe)
apply (rule theI2)
apply (auto intro: lemma-floor)
done

lemma real-of-int-floor-add-one-ge [simp]: r ≤ real(floor r) + 1
apply (insert real-of-int-floor-ge-diff-one [of r])
apply (auto simp del: real-of-int-floor-ge-diff-one)
done

lemma real-of-int-floor-add-one-gt [simp]: r < real(floor r) + 1
apply (insert real-of-int-floor-gt-diff-one [of r])
apply (auto simp del: real-of-int-floor-gt-diff-one)
done

lemma le-floor: real a <= x ==> a <= floor x
  apply (subgoal-tac a < floor x + 1)
  apply arith
  apply (subst real-of-int-less-iff [THEN sym])
  apply simp
  apply (insert real-of-int-floor-add-one-gt [of x])
  apply arith
done

lemma real-le-floor: a <= floor x ==> real a <= x
  apply (rule order-trans)
  prefer 2
  apply (rule real-of-int-floor-le)
  apply (subst real-of-int-le-iff)
  apply assumption
done

lemma le-floor-eq: (a <= floor x) = (real a <= x)
  apply (rule iffI)
```

apply (*erule real-le-floor*)
apply (*erule le-floor*)
done

lemma *le-floor-eq-number-of* [*simp*]:
(*number-of n <= floor x*) = (*number-of n <= x*)
by (*simp add*: *le-floor-eq*)

lemma *le-floor-eq-zero* [*simp*]: (*0 <= floor x*) = (*0 <= x*)
by (*simp add*: *le-floor-eq*)

lemma *le-floor-eq-one* [*simp*]: (*1 <= floor x*) = (*1 <= x*)
by (*simp add*: *le-floor-eq*)

lemma *floor-less-eq*: (*floor x < a*) = (*x < real a*)
apply (*subst linorder-not-le* [*THEN sym*])+
apply *simp*
apply (*rule le-floor-eq*)
done

lemma *floor-less-eq-number-of* [*simp*]:
(*floor x < number-of n*) = (*x < number-of n*)
by (*simp add*: *floor-less-eq*)

lemma *floor-less-eq-zero* [*simp*]: (*floor x < 0*) = (*x < 0*)
by (*simp add*: *floor-less-eq*)

lemma *floor-less-eq-one* [*simp*]: (*floor x < 1*) = (*x < 1*)
by (*simp add*: *floor-less-eq*)

lemma *less-floor-eq*: (*a < floor x*) = (*real a + 1 <= x*)
apply (*insert le-floor-eq* [*of a + 1 x*])
apply *auto*
done

lemma *less-floor-eq-number-of* [*simp*]:
(*number-of n < floor x*) = (*number-of n + 1 <= x*)
by (*simp add*: *less-floor-eq*)

lemma *less-floor-eq-zero* [*simp*]: (*0 < floor x*) = (*1 <= x*)
by (*simp add*: *less-floor-eq*)

lemma *less-floor-eq-one* [*simp*]: (*1 < floor x*) = (*2 <= x*)
by (*simp add*: *less-floor-eq*)

lemma *floor-le-eq*: (*floor x <= a*) = (*x < real a + 1*)
apply (*insert floor-less-eq* [*of x a + 1*])
apply *auto*
done

lemma *floor-le-eq-number-of* [*simp*]:
  (*floor x <= number-of n*) = (*x < number-of n + 1*)
by (*simp add*: *floor-le-eq*)

lemma *floor-le-eq-zero* [*simp*]: (*floor x <= 0*) = (*x < 1*)
by (*simp add*: *floor-le-eq*)

lemma *floor-le-eq-one* [*simp*]: (*floor x <= 1*) = (*x < 2*)
by (*simp add*: *floor-le-eq*)

lemma *floor-add* [*simp*]: *floor* (*x + real a*) = *floor x + a*
  apply (*subst order-eq-iff*)
  apply (*rule conjI*)
  prefer *2*
  apply (*subgoal-tac floor x + a < floor* (*x + real a*) + *1*)
  apply *arith*
  apply (*subst real-of-int-less-iff* [*THEN sym*])
  apply *simp*
  apply (*subgoal-tac x + real a < real*(*floor*(*x + real a*)) + *1*)
  apply (*subgoal-tac real* (*floor x*) <= *x*)
  apply *arith*
  apply (*rule real-of-int-floor-le*)
  apply (*rule real-of-int-floor-add-one-gt*)
  apply (*subgoal-tac floor* (*x + real a*) < *floor x + a + 1*)
  apply *arith*
  apply (*subst real-of-int-less-iff* [*THEN sym*])
  apply *simp*
  apply (*subgoal-tac real*(*floor*(*x + real a*)) <= *x + real a*)
  apply (*subgoal-tac x < real*(*floor x*) + *1*)
  apply *arith*
  apply (*rule real-of-int-floor-add-one-gt*)
  apply (*rule real-of-int-floor-le*)
done

lemma *floor-add-number-of* [*simp*]:
  floor (*x + number-of n*) = *floor x + number-of n*
  apply (*subst floor-add* [*THEN sym*])
  apply *simp*
done

lemma *floor-add-one* [*simp*]: *floor* (*x + 1*) = *floor x + 1*
  apply (*subst floor-add* [*THEN sym*])
  apply *simp*
done

lemma *floor-subtract* [*simp*]: *floor* (*x − real a*) = *floor x − a*
  apply (*subst diff-minus*)+
  apply (*subst real-of-int-minus* [*THEN sym*])

apply (*rule floor-add*)
done

lemma *floor-subtract-number-of* [*simp*]: *floor* (*x* − *number-of n*) =
floor x − *number-of n*
apply (*subst floor-subtract* [*THEN sym*])
apply *simp*
done

lemma *floor-subtract-one* [*simp*]: *floor* (*x* − *1*) = *floor x* − *1*
apply (*subst floor-subtract* [*THEN sym*])
apply *simp*
done

lemma *ceiling-zero* [*simp*]: *ceiling 0* = *0*
by (*simp add*: *ceiling-def*)

lemma *ceiling-real-of-nat* [*simp*]: *ceiling* (*real* (*n*::*nat*)) = *int n*
by (*simp add*: *ceiling-def*)

lemma *ceiling-real-of-nat-zero* [*simp*]: *ceiling* (*real* (*0*::*nat*)) = *0*
by *auto*

lemma *ceiling-floor* [*simp*]: *ceiling* (*real* (*floor r*)) = *floor r*
by (*simp add*: *ceiling-def*)

lemma *floor-ceiling* [*simp*]: *floor* (*real* (*ceiling r*)) = *ceiling r*
by (*simp add*: *ceiling-def*)

lemma *real-of-int-ceiling-ge* [*simp*]: *r* ≤ *real* (*ceiling r*)
apply (*simp add*: *ceiling-def*)
apply (*subst le-minus-iff*, *simp*)
done

lemma *ceiling-mono*: *x* < *y* ==> *ceiling x* ≤ *ceiling y*
by (*simp add*: *floor-mono ceiling-def*)

lemma *ceiling-mono2*: *x* ≤ *y* ==> *ceiling x* ≤ *ceiling y*
by (*simp add*: *floor-mono2 ceiling-def*)

lemma *real-of-int-ceiling-cancel* [*simp*]:
(*real* (*ceiling x*) = *x*) = (∃ *n*::*int*. *x* = *real n*)
apply (*auto simp add*: *ceiling-def*)
apply (*drule arg-cong* [**where** *f* = *uminus*], *auto*)
apply (*rule-tac x* = −*n* **in** *exI*, *auto*)
done

lemma *ceiling-eq*: [| *real n* < *x*; *x* < *real n* + *1* |] ==> *ceiling x* = *n* + *1*
apply (*simp add*: *ceiling-def*)

apply (*rule minus-equation-iff* [*THEN iffD1*])
apply (*simp add*: *floor-eq* [**where** *n* = −(*n*+*1*)])
done

lemma *ceiling-eq2*: [| *real n* < *x*; *x* ≤ *real n* + *1* |] ==> *ceiling x* = *n* + *1*
by (*simp add*: *ceiling-def floor-eq2* [**where** *n* = −(*n*+*1*)])

lemma *ceiling-eq3*: [| *real n* − *1* < *x*; *x* ≤ *real n* |] ==> *ceiling x* = *n*
by (*simp add*: *ceiling-def floor-eq2* [**where** *n* = −*n*])

lemma *ceiling-real-of-int* [*simp*]: *ceiling* (*real* (*n*::*int*)) = *n*
by (*simp add*: *ceiling-def*)

lemma *ceiling-number-of-eq* [*simp*]:
    ceiling (*number-of n* :: *real*) = (*number-of n*)
apply (*subst real-number-of* [*symmetric*])
apply (*rule ceiling-real-of-int*)
done

lemma *ceiling-one* [*simp*]: *ceiling 1* = *1*
  by (*unfold ceiling-def*, *simp*)

lemma *real-of-int-ceiling-diff-one-le* [*simp*]: *real* (*ceiling r*) − *1* ≤ *r*
apply (*rule neg-le-iff-le* [*THEN iffD1*])
apply (*simp add*: *ceiling-def diff-minus*)
done

lemma *real-of-int-ceiling-le-add-one* [*simp*]: *real* (*ceiling r*) ≤ *r* + *1*
apply (*insert real-of-int-ceiling-diff-one-le* [*of r*])
apply (*simp del*: *real-of-int-ceiling-diff-one-le*)
done

lemma *ceiling-le*: *x* <= *real a* ==> *ceiling x* <= *a*
  apply (*unfold ceiling-def*)
  apply (*subgoal-tac* −*a* <= *floor*(− *x*))
  apply *simp*
  apply (*rule le-floor*)
  apply *simp*
done

lemma *ceiling-le-real*: *ceiling x* <= *a* ==> *x* <= *real a*
  apply (*unfold ceiling-def*)
  apply (*subgoal-tac real*(− *a*) <= − *x*)
  apply *simp*
  apply (*rule real-le-floor*)
  apply *simp*
done

lemma *ceiling-le-eq*: (*ceiling x* <= *a*) = (*x* <= *real a*)

```
  apply (rule iffI)
  apply (erule ceiling-le-real)
  apply (erule ceiling-le)
done

lemma ceiling-le-eq-number-of [simp]:
    (ceiling x <= number-of n) = (x <= number-of n)
by (simp add: ceiling-le-eq)

lemma ceiling-le-zero-eq [simp]: (ceiling x <= 0) = (x <= 0)
by (simp add: ceiling-le-eq)

lemma ceiling-le-eq-one [simp]: (ceiling x <= 1) = (x <= 1)
by (simp add: ceiling-le-eq)

lemma less-ceiling-eq: (a < ceiling x) = (real a < x)
  apply (subst linorder-not-le [THEN sym])+
  apply simp
  apply (rule ceiling-le-eq)
done

lemma less-ceiling-eq-number-of [simp]:
    (number-of n < ceiling x) = (number-of n < x)
by (simp add: less-ceiling-eq)

lemma less-ceiling-eq-zero [simp]: (0 < ceiling x) = (0 < x)
by (simp add: less-ceiling-eq)

lemma less-ceiling-eq-one [simp]: (1 < ceiling x) = (1 < x)
by (simp add: less-ceiling-eq)

lemma ceiling-less-eq: (ceiling x < a) = (x <= real a - 1)
  apply (insert ceiling-le-eq [of x a - 1])
  apply auto
done

lemma ceiling-less-eq-number-of [simp]:
    (ceiling x < number-of n) = (x <= number-of n - 1)
by (simp add: ceiling-less-eq)

lemma ceiling-less-eq-zero [simp]: (ceiling x < 0) = (x <= -1)
by (simp add: ceiling-less-eq)

lemma ceiling-less-eq-one [simp]: (ceiling x < 1) = (x <= 0)
by (simp add: ceiling-less-eq)

lemma le-ceiling-eq: (a <= ceiling x) = (real a - 1 < x)
  apply (insert less-ceiling-eq [of a - 1 x])
  apply auto
```

done

lemma *le-ceiling-eq-number-of* [*simp*]:
  (*number-of n* <= *ceiling x*) = (*number-of n* − *1* < *x*)
by (*simp add*: *le-ceiling-eq*)

lemma *le-ceiling-eq-zero* [*simp*]: (*0* <= *ceiling x*) = (−*1* < *x*)
by (*simp add*: *le-ceiling-eq*)

lemma *le-ceiling-eq-one* [*simp*]: (*1* <= *ceiling x*) = (*0* < *x*)
by (*simp add*: *le-ceiling-eq*)

lemma *ceiling-add* [*simp*]: *ceiling* (*x* + *real a*) = *ceiling x* + *a*
  apply (*unfold ceiling-def*, *simp*)
  apply (*subst real-of-int-minus* [*THEN sym*])
  apply (*subst floor-add*)
  apply *simp*
done

lemma *ceiling-add-number-of* [*simp*]: *ceiling* (*x* + *number-of n*) =
    ceiling x + *number-of n*
  apply (*subst ceiling-add* [*THEN sym*])
  apply *simp*
done

lemma *ceiling-add-one* [*simp*]: *ceiling* (*x* + *1*) = *ceiling x* + *1*
  apply (*subst ceiling-add* [*THEN sym*])
  apply *simp*
done

lemma *ceiling-subtract* [*simp*]: *ceiling* (*x* − *real a*) = *ceiling x* − *a*
  apply (*subst diff-minus*)+
  apply (*subst real-of-int-minus* [*THEN sym*])
  apply (*rule ceiling-add*)
done

lemma *ceiling-subtract-number-of* [*simp*]: *ceiling* (*x* − *number-of n*) =
    ceiling x − *number-of n*
  apply (*subst ceiling-subtract* [*THEN sym*])
  apply *simp*
done

lemma *ceiling-subtract-one* [*simp*]: *ceiling* (*x* − *1*) = *ceiling x* − *1*
  apply (*subst ceiling-subtract* [*THEN sym*])
  apply *simp*
done

12.4 Versions for the natural numbers

definition
natfloor :: *real* => *nat* **where**
natfloor x = *nat*(*floor x*)

definition
natceiling :: *real* => *nat* **where**
natceiling x = *nat*(*ceiling x*)

lemma *natfloor-zero* [*simp*]: *natfloor 0* = *0*
by (*unfold natfloor-def*, *simp*)

lemma *natfloor-one* [*simp*]: *natfloor 1* = *1*
by (*unfold natfloor-def*, *simp*)

lemma *zero-le-natfloor* [*simp*]: *0* <= *natfloor x*
by (*unfold natfloor-def*, *simp*)

lemma *natfloor-number-of-eq* [*simp*]: *natfloor* (*number-of n*) = *number-of n*
by (*unfold natfloor-def*, *simp*)

lemma *natfloor-real-of-nat* [*simp*]: *natfloor*(*real n*) = *n*
by (*unfold natfloor-def*, *simp*)

lemma *real-natfloor-le*: *0* <= *x* ==> *real*(*natfloor x*) <= *x*
by (*unfold natfloor-def*, *simp*)

lemma *natfloor-neg*: *x* <= *0* ==> *natfloor x* = *0*
apply (*unfold natfloor-def*)
apply (*subgoal-tac floor x* <= *floor 0*)
apply *simp*
apply (*erule floor-mono2*)
done

lemma *natfloor-mono*: *x* <= *y* ==> *natfloor x* <= *natfloor y*
apply (*case-tac 0* <= *x*)
apply (*subst natfloor-def*)+
apply (*subst nat-le-eq-zle*)
apply *force*
apply (*erule floor-mono2*)
apply (*subst natfloor-neg*)
apply *simp*
apply *simp*
done

lemma *le-natfloor*: *real x* <= *a* ==> *x* <= *natfloor a*
apply (*unfold natfloor-def*)
apply (*subst nat-int* [*THEN sym*])
apply (*subst nat-le-eq-zle*)

apply *simp*
apply (*rule le-floor*)
apply *simp*
done

lemma *le-natfloor-eq*: *0 <= x ==> (a <= natfloor x) = (real a <= x)*
apply (*rule iffI*)
apply (*rule order-trans*)
prefer *2*
apply (*erule real-natfloor-le*)
apply (*subst real-of-nat-le-iff*)
apply *assumption*
apply (*erule le-natfloor*)
done

lemma *le-natfloor-eq-number-of* [*simp*]:
~ *neg*((*number-of n*)::*int*) *==> 0 <= x ==>*
(*number-of n <= natfloor x*) = (*number-of n <= x*)
apply (*subst le-natfloor-eq*, *assumption*)
apply *simp*
done

lemma *le-natfloor-eq-one* [*simp*]: (*1 <= natfloor x*) = (*1 <= x*)
apply (*case-tac 0 <= x*)
apply (*subst le-natfloor-eq*, *assumption*, *simp*)
apply (*rule iffI*)
apply (*subgoal-tac natfloor x <= natfloor 0*)
apply *simp*
apply (*rule natfloor-mono*)
apply *simp*
apply *simp*
done

lemma *natfloor-eq*: *real n <= x ==> x < real n + 1 ==> natfloor x = n*
apply (*unfold natfloor-def*)
apply (*subst nat-int* [*THEN sym*])**back**
apply (*subst eq-nat-nat-iff*)
apply *simp*
apply *simp*
apply (*rule floor-eq2*)
apply *auto*
done

lemma *real-natfloor-add-one-gt*: *x < real*(*natfloor x*) + *1*
apply (*case-tac 0 <= x*)
apply (*unfold natfloor-def*)
apply *simp*
apply *simp-all*
done

lemma *real-natfloor-gt-diff-one*: $x - 1 < real(natfloor\ x)$
apply (*simp add*: *compare-rls*)
apply (*rule real-natfloor-add-one-gt*)
done

lemma *ge-natfloor-plus-one-imp-gt*: $natfloor\ z + 1 <= n ==> z < real\ n$
apply (*subgoal-tac* $z < real(natfloor\ z) + 1$)
apply *arith*
apply (*rule real-natfloor-add-one-gt*)
done

lemma *natfloor-add* [*simp*]: $0 <= x ==> natfloor\ (x + real\ a) = natfloor\ x + a$
apply (*unfold natfloor-def*)
apply (*subgoal-tac real a = real* (*int a*))
apply (*erule ssubst*)
apply (*simp add*: *nat-add-distrib del*: *real-of-int-of-nat-eq*)
apply *simp*
done

lemma *natfloor-add-number-of* [*simp*]:
$\sim neg\ ((number\text{-}of\ n)::int) ==> 0 <= x ==>$
$natfloor\ (x + number\text{-}of\ n) = natfloor\ x + number\text{-}of\ n$
apply (*subst natfloor-add* [*THEN sym*])
apply *simp-all*
done

lemma *natfloor-add-one*: $0 <= x ==> natfloor(x + 1) = natfloor\ x + 1$
apply (*subst natfloor-add* [*THEN sym*])
apply *assumption*
apply *simp*
done

lemma *natfloor-subtract* [*simp*]: $real\ a <= x ==>$
$natfloor(x - real\ a) = natfloor\ x - a$
apply (*unfold natfloor-def*)
apply (*subgoal-tac real a = real* (*int a*))
apply (*erule ssubst*)
apply (*simp del*: *real-of-int-of-nat-eq*)
apply *simp*
done

lemma *natceiling-zero* [*simp*]: $natceiling\ 0 = 0$
by (*unfold natceiling-def*, *simp*)

lemma *natceiling-one* [*simp*]: $natceiling\ 1 = 1$
by (*unfold natceiling-def*, *simp*)

lemma *zero-le-natceiling* [*simp*]: $0 <= natceiling\ x$

by (*unfold natceiling-def*, *simp*)

lemma *natceiling-number-of-eq* [*simp*]: *natceiling* (*number-of n*) = *number-of n*
by (*unfold natceiling-def*, *simp*)

lemma *natceiling-real-of-nat* [*simp*]: *natceiling*(*real n*) = *n*
by (*unfold natceiling-def*, *simp*)

lemma *real-natceiling-ge*: *x* <= *real*(*natceiling x*)
apply (*unfold natceiling-def*)
apply (*case-tac x* < *0*)
apply *simp*
apply (*subst real-nat-eq-real*)
apply (*subgoal-tac ceiling 0* <= *ceiling x*)
apply *simp*
apply (*rule ceiling-mono2*)
apply *simp*
apply *simp*
done

lemma *natceiling-neg*: *x* <= *0* ==> *natceiling x* = *0*
apply (*unfold natceiling-def*)
apply *simp*
done

lemma *natceiling-mono*: *x* <= *y* ==> *natceiling x* <= *natceiling y*
apply (*case-tac 0* <= *x*)
apply (*subst natceiling-def*)+
apply (*subst nat-le-eq-zle*)
apply (*rule disjI2*)
apply (*subgoal-tac real* (*0*::*int*) <= *real*(*ceiling y*))
apply *simp*
apply (*rule order-trans*)
apply *simp*
apply (*erule order-trans*)
apply *simp*
apply (*erule ceiling-mono2*)
apply (*subst natceiling-neg*)
apply *simp-all*
done

lemma *natceiling-le*: *x* <= *real a* ==> *natceiling x* <= *a*
apply (*unfold natceiling-def*)
apply (*case-tac x* < *0*)
apply *simp*
apply (*subst nat-int* [*THEN sym*])**back**
apply (*subst nat-le-eq-zle*)
apply *simp*
apply (*rule ceiling-le*)

```
  apply simp
done

lemma natceiling-le-eq: 0 <= x ==> (natceiling x <= a) = (x <= real a)
  apply (rule iffI)
  apply (rule order-trans)
  apply (rule real-natceiling-ge)
  apply (subst real-of-nat-le-iff)
  apply assumption
  apply (erule natceiling-le)
done

lemma natceiling-le-eq-number-of [simp]:
    ~ neg((number-of n)::int) ==> 0 <= x ==>
      (natceiling x <= number-of n) = (x <= number-of n)
  apply (subst natceiling-le-eq, assumption)
  apply simp
done

lemma natceiling-le-eq-one: (natceiling x <= 1) = (x <= 1)
  apply (case-tac 0 <= x)
  apply (subst natceiling-le-eq)
  apply assumption
  apply simp
  apply (subst natceiling-neg)
  apply simp
  apply simp
done

lemma natceiling-eq: real n < x ==> x <= real n + 1 ==> natceiling x = n +
1
  apply (unfold natceiling-def)
  apply (simplesubst nat-int [THEN sym]) back back
  apply (subgoal-tac nat(int n) + 1 = nat(int n + 1))
  apply (erule ssubst)
  apply (subst eq-nat-nat-iff)
  apply (subgoal-tac ceiling 0 <= ceiling x)
  apply simp
  apply (rule ceiling-mono2)
  apply force
  apply force
  apply (rule ceiling-eq2)
  apply (simp, simp)
  apply (subst nat-add-distrib)
  apply auto
done

lemma natceiling-add [simp]: 0 <= x ==>
    natceiling (x + real a) = natceiling x + a
```

```
  apply (unfold natceiling-def)
  apply (subgoal-tac real a = real (int a))
  apply (erule ssubst)
  apply (simp del: real-of-int-of-nat-eq)
  apply (subst nat-add-distrib)
  apply (subgoal-tac 0 = ceiling 0)
  apply (erule ssubst)
  apply (erule ceiling-mono2)
  apply simp-all
done

lemma natceiling-add-number-of [simp]:
    ~ neg ((number-of n)::int) ==> 0 <= x ==>
      natceiling (x + number-of n) = natceiling x + number-of n
  apply (subst natceiling-add [THEN sym])
  apply simp-all
done

lemma natceiling-add-one: 0 <= x ==> natceiling(x + 1) = natceiling x + 1
  apply (subst natceiling-add [THEN sym])
  apply assumption
  apply simp
done

lemma natceiling-subtract [simp]: real a <= x ==>
    natceiling(x - real a) = natceiling x - a
  apply (unfold natceiling-def)
  apply (subgoal-tac real a = real (int a))
  apply (erule ssubst)
  apply (simp del: real-of-int-of-nat-eq)
  apply simp
done

lemma natfloor-div-nat: 1 <= x ==> y > 0 ==>
  natfloor (x / real y) = natfloor x div y
proof -
  assume 1 <= (x::real) and (y::nat) > 0
  have natfloor x = (natfloor x) div y * y + (natfloor x) mod y
    by simp
  then have a: real(natfloor x) = real ((natfloor x) div y) * real y +
    real((natfloor x) mod y)
    by (simp only: real-of-nat-add [THEN sym] real-of-nat-mult [THEN sym])
  have x = real(natfloor x) + (x - real(natfloor x))
    by simp
  then have x = real ((natfloor x) div y) * real y +
      real((natfloor x) mod y) + (x - real(natfloor x))
    by (simp add: a)
  then have x / real y = ... / real y
    by simp
```

also have ... = *real*((*natfloor x*) *div y*) + *real*((*natfloor x*) *mod y*) /
real y + (*x* − *real*(*natfloor x*)) / *real y*
by (*auto simp add*: *ring-simps add-divide-distrib*
diff-divide-distrib prems)
finally have *natfloor* (*x* / *real y*) = *natfloor*(...) **by** *simp*
also have ... = *natfloor*(*real*((*natfloor x*) *mod y*) /
real y + (*x* − *real*(*natfloor x*)) / *real y* + *real*((*natfloor x*) *div y*))
by (*simp add*: *add-ac*)
also have ... = *natfloor*(*real*((*natfloor x*) *mod y*) /
real y + (*x* − *real*(*natfloor x*)) / *real y*) + (*natfloor x*) *div y*
apply (*rule natfloor-add*)
apply (*rule add-nonneg-nonneg*)
apply (*rule divide-nonneg-pos*)
apply *simp*
apply (*simp add*: *prems*)
apply (*rule divide-nonneg-pos*)
apply (*simp add*: *compare-rls*)
apply (*rule real-natfloor-le*)
apply (*insert prems*, *auto*)
done
also have *natfloor*(*real*((*natfloor x*) *mod y*) /
real y + (*x* − *real*(*natfloor x*)) / *real y*) = *0*
apply (*rule natfloor-eq*)
apply *simp*
apply (*rule add-nonneg-nonneg*)
apply (*rule divide-nonneg-pos*)
apply *force*
apply (*force simp add*: *prems*)
apply (*rule divide-nonneg-pos*)
apply (*simp add*: *compare-rls*)
apply (*rule real-natfloor-le*)
apply (*auto simp add*: *prems*)
apply (*insert prems*, *arith*)
apply (*simp add*: *add-divide-distrib* [*THEN sym*])
apply (*subgoal-tac real y* = *real y* − *1* + *1*)
apply (*erule ssubst*)
apply (*rule add-le-less-mono*)
apply (*simp add*: *compare-rls*)
apply (*subgoal-tac real*(*natfloor x mod y*) + *1* =
real(*natfloor x mod y* + *1*))
apply (*erule ssubst*)
apply (*subst real-of-nat-le-iff*)
apply (*subgoal-tac natfloor x mod y* < *y*)
apply *arith*
apply (*rule mod-less-divisor*)
apply *auto*
apply (*simp add*: *compare-rls*)
apply (*subst add-commute*)
apply (*rule real-natfloor-add-one-gt*)

done
finally show *?thesis* **by** *simp*
qed

end

13 Non-denumerability of the Continuum.

theory *ContNotDenum*
imports *RComplete*
begin

13.1 Abstract

The following document presents a proof that the Continuum is uncountable. It is formalised in the Isabelle/Isar theorem proving system.

Theorem: The Continuum $\mathbb{R}$ is not denumerable. In other words, there does not exist a function f:$\mathbb{N}\Rightarrow\mathbb{R}$ such that f is surjective.

Outline: An elegant informal proof of this result uses Cantor's Diagonalisation argument. The proof presented here is not this one. First we formalise some properties of closed intervals, then we prove the Nested Interval Property. This property relies on the completeness of the Real numbers and is the foundation for our argument. Informally it states that an intersection of countable closed intervals (where each successive interval is a subset of the last) is non-empty. We then assume a surjective function f:$\mathbb{N}\Rightarrow\mathbb{R}$ exists and find a real x such that x is not in the range of f by generating a sequence of closed intervals then using the NIP.

13.2 Closed Intervals

This section formalises some properties of closed intervals.

13.2.1 Definition

definition
closed-int :: *real* ⇒ *real* ⇒ *real set* **where**
closed-int x y = $\{z.\ x \leq z \wedge z \leq y\}$

13.2.2 Properties

lemma *closed-int-subset*:
assumes *xy*: *x1* ≥ *x0 y1* ≤ *y0*
shows *closed-int x1 y1* ⊆ *closed-int x0 y0*
proof −
{

fix x::*real*
assume $x \in$ *closed-int x1 y1*
hence $x \geq x1 \wedge x \leq y1$ **by** (*simp add*: *closed-int-def*)
with *xy* **have** $x \geq x0 \wedge x \leq y0$ **by** *auto*
hence $x \in$ *closed-int x0 y0* **by** (*simp add*: *closed-int-def*)
}
thus *?thesis* **by** *auto*
qed

lemma *closed-int-least*:
assumes *a*: $a \leq b$
shows $a \in$ *closed-int a b* $\wedge$ ($\forall x \in$ *closed-int a b*. $a \leq x$)
proof
from *a* **have** $a \in \{x.\ a \leq x \wedge x \leq b\}$ **by** *simp*
thus $a \in$ *closed-int a b* **by** (*unfold closed-int-def*)
next
have $\forall x \in \{x.\ a \leq x \wedge x \leq b\}.\ a \leq x$ **by** *simp*
thus $\forall x \in$ *closed-int a b*. $a \leq x$ **by** (*unfold closed-int-def*)
qed

lemma *closed-int-most*:
assumes *a*: $a \leq b$
shows $b \in$ *closed-int a b* $\wedge$ ($\forall x \in$ *closed-int a b*. $x \leq b$)
proof
from *a* **have** $b \in \{x.\ a \leq x \wedge x \leq b\}$ **by** *simp*
thus $b \in$ *closed-int a b* **by** (*unfold closed-int-def*)
next
have $\forall x \in \{x.\ a \leq x \wedge x \leq b\}.\ x \leq b$ **by** *simp*
thus $\forall x \in$ *closed-int a b*. $x \leq b$ **by** (*unfold closed-int-def*)
qed

lemma *closed-not-empty*:
shows $a \leq b \Longrightarrow \exists x.\ x \in$ *closed-int a b*
by (*auto dest*: *closed-int-least*)

lemma *closed-mem*:
assumes $a \leq c$ **and** $c \leq b$
shows $c \in$ *closed-int a b*
using *assms* **unfolding** *closed-int-def* **by** *auto*

lemma *closed-subset*:
assumes *ac*: $a \leq b \quad c \leq d$
assumes *closed*: *closed-int a b* $\subseteq$ *closed-int c d*
shows $b \geq c$
proof −
from *closed* **have** $\forall x \in$*closed-int a b*. $x \in$*closed-int c d* **by** *auto*
hence $\forall x.\ a \leq x \wedge x \leq b \longrightarrow c \leq x \wedge x \leq d$ **by** (*unfold closed-int-def*, *auto*)
with *ac* **have** $c \leq b \wedge b \leq d$ **by** *simp*
thus *?thesis* **by** *auto*

qed

13.3 Nested Interval Property

theorem *NIP*:
fixes f::*nat* $\Rightarrow$ *real set*
assumes *subset*: $\forall\, n.\ f\ (Suc\ n) \subseteq f\ n$
and *closed*: $\forall\, n.\ \exists\, a\ b.\ f\ n = closed\text{-}int\ a\ b \wedge a \leq b$
shows $(\bigcap n.\ f\ n) \neq \{\}$
proof −
let *?g* = $\lambda n.\ (SOME\ c.\ c \in (f\ n) \wedge (\forall\, x \in (f\ n).\ c \leq x))$
have *ne*: $\forall\, n.\ \exists\, x.\ x \in (f\ n)$
proof
fix n
from *closed* **have** $\exists\, a\ b.\ f\ n = closed\text{-}int\ a\ b \wedge a \leq b$ **by** *simp*
then obtain a **and** b **where** *fn*: $f\ n = closed\text{-}int\ a\ b \wedge a \leq b$ **by** *auto*
hence $a \leq b$ **..**
with *closed-not-empty* **have** $\exists\, x.\ x \in closed\text{-}int\ a\ b$ **by** *simp*
with *fn* **show** $\exists\, x.\ x \in (f\ n)$ **by** *simp*
qed

have *gdef*: $\forall\, n.\ (?g\ n) \in (f\ n) \wedge (\forall\, x \in (f\ n).\ (?g\ n) \leq x)$
proof
fix n
from *closed* **have** $\exists\, a\ b.\ f\ n = closed\text{-}int\ a\ b \wedge a \leq b$ **..**
then obtain a **and** b **where** *ff*: $f\ n = closed\text{-}int\ a\ b$ **and** $a \leq b$ **by** *auto*
hence $a \leq b$ **by** *simp*
hence $a \in closed\text{-}int\ a\ b \wedge (\forall\, x \in closed\text{-}int\ a\ b.\ a \leq x)$ **by** (*rule closed-int-least*)
with *ff* **have** $a \in (f\ n) \wedge (\forall\, x \in (f\ n).\ a \leq x)$ **by** *simp*
hence $\exists\, c.\ c \in (f\ n) \wedge (\forall\, x \in (f\ n).\ c \leq x)$ **..**
thus $(?g\ n) \in (f\ n) \wedge (\forall\, x \in (f\ n).\ (?g\ n) \leq x)$ **by** (*rule someI-ex*)
qed

— A denotes the set of all left-most points of all the intervals ...
moreover obtain A **where** *Adef*: $A = ?g\ `\ \mathbb{N}$ **by** *simp*
ultimately have $\exists\, x.\ x \in A$
proof −
have $(0::nat) \in \mathbb{N}$ **by** *simp*
moreover have *?g 0* = *?g 0* **by** *simp*
ultimately have *?g 0* $\in$ *?g* ` $\mathbb{N}$ **by** (*rule rev-image-eqI*)
with *Adef* **have** *?g 0* $\in A$ **by** *simp*
thus *?thesis* **..**
qed

— Now show that A is bounded above ...
moreover have $\exists\, y.\ isUb\ (UNIV::real\ set)\ A\ y$
proof −
{
fix n

```
    from ne have ex: ∃x. x∈(f n) ..
    from gdef have (?g n)∈(f n) ∧ (∀x∈(f n). (?g n)≤x) by simp
    moreover
    from closed have ∃a b. f n = closed-int a b ∧ a ≤ b ..
    then obtain a and b where f n = closed-int a b ∧ a ≤ b by auto
    hence b∈(f n) ∧ (∀x∈(f n). x ≤ b) using closed-int-most by blast
    ultimately have ∀x∈(f n). (?g n) ≤ b by simp
    with ex have (?g n) ≤ b by auto
    hence ∃b. (?g n) ≤ b by auto
  }
  hence aux: ∀n. ∃b. (?g n) ≤ b ..

  have fs: ∀n::nat. f n ⊆ f 0
  proof (rule allI, induct-tac n)
    show f 0 ⊆ f 0 by simp
  next
    fix n
    assume f n ⊆ f 0
    moreover from subset have f (Suc n) ⊆ f n ..
    ultimately show f (Suc n) ⊆ f 0 by simp
  qed
  have ∀n. (?g n)∈(f 0)
  proof
    fix n
    from gdef have (?g n)∈(f n) ∧ (∀x∈(f n). (?g n)≤x) by simp
    hence ?g n ∈ f n ..
    with fs show ?g n ∈ f 0 by auto
  qed
  moreover from closed
    obtain a and b where f 0 = closed-int a b and alb: a ≤ b by blast
  ultimately have ∀n. ?g n ∈ closed-int a b by auto
  with alb have ∀n. ?g n ≤ b using closed-int-most by blast
  with Adef have ∀y∈A. y≤b by auto
  hence A *<= b by (unfold setle-def)
  moreover have b ∈ (UNIV::real set) by simp
  ultimately have A *<= b ∧ b ∈ (UNIV::real set) by simp
  hence isUb (UNIV::real set) A b by (unfold isUb-def)
  thus ?thesis by auto
qed
— by the Axiom Of Completeness, A has a least upper bound ...
ultimately have ∃t. isLub UNIV A t by (rule reals-complete)

— denote this least upper bound as t ...
then obtain t where tdef: isLub UNIV A t ..

— and finally show that this least upper bound is in all the intervals...
have ∀n. t ∈ f n
proof
  fix n::nat
```

```
from closed obtain a and b where
  int: f n = closed-int a b and alb: a ≤ b by blast

have t ≥ a
proof −
  have a ∈ A
  proof −

    from alb int have ain: a∈f n ∧ (∀ x∈f n. a ≤ x)
      using closed-int-least by blast
    moreover have ∀ e. e∈f n ∧ (∀ x∈f n. e ≤ x) ⟶ e = a
    proof clarsimp
      fix e
      assume ein: e ∈ f n and lt: ∀ x∈f n. e ≤ x
      from lt ain have aux: ∀ x∈f n. a ≤ x ∧ e ≤ x by auto

      from ein aux have a ≤ e ∧ e ≤ e by auto
      moreover from ain aux have a ≤ a ∧ e ≤ a by auto
      ultimately show e = a by simp
    qed
    hence ⋀e.  e∈f n ∧ (∀ x∈f n. e ≤ x) ⟹ e = a by simp
    ultimately have (?g n) = a by (rule some-equality)
    moreover
    {
      have n = of-nat n by simp
      moreover have of-nat n ∈ ℕ by simp
      ultimately have n ∈ ℕ
        apply −
        apply (subst(asm) eq-sym-conv)
        apply (erule subst)
        .
    }
    with Adef have (?g n) ∈ A by auto
    ultimately show ?thesis by simp
  qed
  with tdef show a ≤ t by (rule isLubD2)
qed
moreover have t ≤ b
proof −
  have isUb UNIV A b
  proof −
    {
      from alb int have
        ain: b∈f n ∧ (∀ x∈f n. x ≤ b) using closed-int-most by blast

      have subsetd: ∀ m. ∀ n. f (n + m) ⊆ f n
      proof (rule allI, induct-tac m)
        show ∀ n. f (n + 0) ⊆ f n by simp
      next
```

```
  fix m n
  assume pp: ∀p. f (p + n) ⊆ f p
  {
    fix p
    from pp have f (p + n) ⊆ f p by simp
    moreover from subset have f (Suc (p + n)) ⊆ f (p + n) by auto
    hence f (p + (Suc n)) ⊆ f (p + n) by simp
    ultimately have f (p + (Suc n)) ⊆ f p by simp
  }
  thus ∀p. f (p + Suc n) ⊆ f p ..
qed
have subsetm: ∀α β. α ≥ β ⟶ (f α) ⊆ (f β)
proof ((rule allI)+, rule impI)
  fix α::nat and β::nat
  assume β ≤ α
  hence ∃k. α = β + k by (simp only: le-iff-add)
  then obtain k where α = β + k ..
  moreover
  from subsetd have f (β + k) ⊆ f β by simp
  ultimately show f α ⊆ f β by auto
qed

fix m
{
  assume m ≥ n
  with subsetm have f m ⊆ f n by simp
  with ain have ∀x∈f m. x ≤ b by auto
  moreover
  from gdef have ?g m ∈ f m ∧ (∀x∈f m. ?g m ≤ x) by simp
  ultimately have ?g m ≤ b by auto
}
moreover
{
  assume ¬(m ≥ n)
  hence m < n by simp
  with subsetm have sub: (f n) ⊆ (f m) by simp
  from closed obtain ma and mb where
    f m = closed-int ma mb ∧ ma ≤ mb by blast
  hence one: ma ≤ mb and fm: f m = closed-int ma mb by auto
  from one alb sub fm int have ma ≤ b using closed-subset by blast
  moreover have (?g m) = ma
  proof −
    from gdef have ?g m ∈ f m ∧ (∀x∈f m. ?g m ≤ x) ..
    moreover from one have
      ma ∈ closed-int ma mb ∧ (∀x∈closed-int ma mb. ma ≤ x)
      by (rule closed-int-least)
    with fm have ma∈f m ∧ (∀x∈f m. ma ≤ x) by simp
    ultimately have ma ≤ ?g m ∧ ?g m ≤ ma by auto
    thus ?g m = ma by auto
```

```
        qed
        ultimately have ?g m ≤ b by simp
       }
       ultimately have ?g m ≤ b by (rule case-split)
      }
      with Adef have ∀ y∈A. y≤b by auto
      hence A *<= b by (unfold setle-def)
      moreover have b ∈ (UNIV::real set) by simp
      ultimately have A *<= b ∧ b ∈ (UNIV::real set) by simp
      thus isUb (UNIV::real set) A b by (unfold isUb-def)
     qed
     with tdef show t ≤ b by (rule isLub-le-isUb)
    qed
    ultimately have t ∈ closed-int a b by (rule closed-mem)
    with int show t ∈ f n by simp
  qed
  hence t ∈ (⋂ n. f n) by auto
  thus ?thesis by auto
qed
```

13.4 Generating the intervals

13.4.1 Existence of non-singleton closed intervals

This lemma asserts that given any non-singleton closed interval (a,b) and any element c, there exists a closed interval that is a subset of (a,b) and that does not contain c and is a non-singleton itself.

```
lemma closed-subset-ex:
  fixes c::real
  assumes alb: a < b
  shows
    ∃ ka kb. ka < kb ∧ closed-int ka kb ⊆ closed-int a b ∧ c ∉ (closed-int ka kb)
proof −
  {
    assume clb: c < b
    {
      assume cla: c < a
      from alb cla clb have c ∉ closed-int a b by (unfold closed-int-def, auto)
      with alb have
        a < b ∧ closed-int a b ⊆ closed-int a b ∧ c ∉ closed-int a b
        by auto
      hence
        ∃ ka kb. ka < kb ∧ closed-int ka kb ⊆ closed-int a b ∧ c ∉ (closed-int ka kb)
        by auto
    }
    moreover
    {
      assume ncla: ¬(c < a)
      with clb have cdef: a ≤ c ∧ c < b by simp
```

obtain *ka* **where** *kadef*: $ka = (c + b)/2$ **by** *blast*

from *kadef clb* **have** *kalb*: $ka < b$ **by** *auto*
moreover from *kadef cdef* **have** *kagc*: $ka > c$ **by** *simp*
ultimately have $c \notin$(*closed-int ka b*) **by** (*unfold closed-int-def*, *auto*)
moreover from *cdef kagc* **have** $ka \geq a$ **by** *simp*
hence *closed-int ka b* $\subseteq$ *closed-int a b* **by** (*unfold closed-int-def*, *auto*)
ultimately have
$ka < b \ \wedge$ *closed-int ka b* $\subseteq$ *closed-int a b* $\wedge\ c \notin$ *closed-int ka b*
using *kalb* **by** *auto*
hence
$\exists$ *ka kb*. $ka < kb \wedge$ *closed-int ka kb* $\subseteq$ *closed-int a b* $\wedge\ c \notin$ (*closed-int ka kb*)
by *auto*

}
ultimately have
$\exists$ *ka kb*. $ka < kb \wedge$ *closed-int ka kb* $\subseteq$ *closed-int a b* $\wedge\ c \notin$ (*closed-int ka kb*)
by (*rule case-split*)
}
moreover
{
assume $\neg\ (c < b)$
hence *cgeb*: $c \geq b$ **by** *simp*

obtain *kb* **where** *kbdef*: $kb = (a + b)/2$ **by** *blast*
with *alb* **have** *kblb*: $kb < b$ **by** *auto*
with *kbdef cgeb* **have** $a < kb \wedge kb < c$ **by** *auto*
moreover hence $c \notin$ (*closed-int a kb*) **by** (*unfold closed-int-def*, *auto*)
moreover from *kblb* **have**
closed-int a kb $\subseteq$ *closed-int a b* **by** (*unfold closed-int-def*, *auto*)
ultimately have
$a < kb \wedge$ *closed-int a kb* $\subseteq$ *closed-int a b* $\wedge\ c \notin$*closed-int a kb*
by *simp*
hence
$\exists$ *ka kb*. $ka < kb \wedge$ *closed-int ka kb* $\subseteq$ *closed-int a b* $\wedge\ c \notin$ (*closed-int ka kb*)
by *auto*
}
ultimately show *?thesis* **by** (*rule case-split*)
qed

13.5 newInt: Interval generation

Given a function f:$\mathbb{N}\Rightarrow\mathbb{R}$, newInt (Suc n) f returns a closed interval such that *newInt* (*Suc n*) *f* $\subseteq$ *newInt n f* and does not contain *f* (*Suc n*). With the base case defined such that (*f 0*)$\notin$*newInt 0 f*.

13.5.1 Definition

consts *newInt* :: *nat* $\Rightarrow$ (*nat* $\Rightarrow$ *real*) $\Rightarrow$ (*real set*)

primrec
newInt 0 f = closed-int (f 0 + 1) (f 0 + 2)
newInt (Suc n) f =
(SOME e. (∃ e1 e2.
e1 < e2 ∧
e = closed-int e1 e2 ∧
e ⊆ (newInt n f) ∧
(f (Suc n)) ∉ e)
)

13.5.2 Properties

We now show that every application of newInt returns an appropriate interval.

lemma *newInt-ex*:
∃ a b. a < b ∧
newInt (Suc n) f = closed-int a b ∧
newInt (Suc n) f ⊆ newInt n f ∧
f (Suc n) ∉ newInt (Suc n) f
proof *(induct n)*
case *0*

let *?e = SOME e. ∃ e1 e2.*
e1 < e2 ∧
e = closed-int e1 e2 ∧
e ⊆ closed-int (f 0 + 1) (f 0 + 2) ∧
f (Suc 0) ∉ e

have *newInt (Suc 0) f = ?e* **by** *auto*
moreover
have *f 0 + 1 < f 0 + 2* **by** *simp*
with *closed-subset-ex* **have**
∃ ka kb. ka < kb ∧ closed-int ka kb ⊆ closed-int (f 0 + 1) (f 0 + 2) ∧
f (Suc 0) ∉ (closed-int ka kb) **.**
hence
∃ e. ∃ ka kb. ka < kb ∧ e = closed-int ka kb ∧
e ⊆ closed-int (f 0 + 1) (f 0 + 2) ∧ f (Suc 0) ∉ e **by** *simp*
hence
∃ ka kb. ka < kb ∧ ?e = closed-int ka kb ∧
?e ⊆ closed-int (f 0 + 1) (f 0 + 2) ∧ f (Suc 0) ∉ ?e
by *(rule someI-ex)*
ultimately have *∃ e1 e2. e1 < e2 ∧*
newInt (Suc 0) f = closed-int e1 e2 ∧
newInt (Suc 0) f ⊆ closed-int (f 0 + 1) (f 0 + 2) ∧
f (Suc 0) ∉ newInt (Suc 0) f **by** *simp*
thus
∃ a b. a < b ∧ newInt (Suc 0) f = closed-int a b ∧
newInt (Suc 0) f ⊆ newInt 0 f ∧ f (Suc 0) ∉ newInt (Suc 0) f
by *simp*

next
case (*Suc n*)
hence ∃ *a b*.
a < *b* ∧
newInt (*Suc n*) *f* = *closed-int a b* ∧
newInt (*Suc n*) *f* ⊆ *newInt n f* ∧
f (*Suc n*) ∉ *newInt* (*Suc n*) *f* **by** *simp*
then obtain *a* **and** *b* **where** *ab*: *a* < *b* ∧
newInt (*Suc n*) *f* = *closed-int a b* ∧
newInt (*Suc n*) *f* ⊆ *newInt n f* ∧
f (*Suc n*) ∉ *newInt* (*Suc n*) *f* **by** *auto*
hence *cab*: *closed-int a b* = *newInt* (*Suc n*) *f* **by** *simp*

let *?e* = *SOME e*. ∃ *e1 e2*.
e1 < *e2* ∧
e = *closed-int e1 e2* ∧
e ⊆ *closed-int a b* ∧
f (*Suc* (*Suc n*)) ∉ *e*
from *cab* **have** *ni*: *newInt* (*Suc* (*Suc n*)) *f* = *?e* **by** *auto*

from *ab* **have** *a* < *b* **by** *simp*
with *closed-subset-ex* **have**
∃ *ka kb*. *ka* < *kb* ∧ *closed-int ka kb* ⊆ *closed-int a b* ∧
f (*Suc* (*Suc n*)) ∉ *closed-int ka kb* **.**
hence
∃ *e*. ∃ *ka kb*. *ka* < *kb* ∧ *e* = *closed-int ka kb* ∧
closed-int ka kb ⊆ *closed-int a b* ∧ *f* (*Suc* (*Suc n*)) ∉ *closed-int ka kb*
by *simp*
hence
∃ *e*. ∃ *ka kb*. *ka* < *kb* ∧ *e* = *closed-int ka kb* ∧
e ⊆ *closed-int a b* ∧ *f* (*Suc* (*Suc n*)) ∉ *e* **by** *simp*
hence
∃ *ka kb*. *ka* < *kb* ∧ *?e* = *closed-int ka kb* ∧
?e ⊆ *closed-int a b* ∧ *f* (*Suc* (*Suc n*)) ∉ *?e* **by** (*rule someI-ex*)
with *ab ni* **show**
∃ *ka kb*. *ka* < *kb* ∧
newInt (*Suc* (*Suc n*)) *f* = *closed-int ka kb* ∧
newInt (*Suc* (*Suc n*)) *f* ⊆ *newInt* (*Suc n*) *f* ∧
f (*Suc* (*Suc n*)) ∉ *newInt* (*Suc* (*Suc n*)) *f* **by** *auto*
qed

lemma *newInt-subset*:
newInt (*Suc n*) *f* ⊆ *newInt n f*
using *newInt-ex* **by** *auto*

Another fundamental property is that no element in the range of f is in the intersection of all closed intervals generated by newInt.

lemma *newInt-inter*:
∀ *n*. *f n* ∉ (⋂ *n*. *newInt n f*)

proof
fix n::nat
{
assume $n0$: $n = 0$
moreover have $newInt\ 0\ f = closed\text{-}int\ (f\ 0 + 1)\ (f\ 0 + 2)$ **by** $simp$
ultimately have $f\ n \notin newInt\ n\ f$ **by** ($unfold\ closed\text{-}int\text{-}def$, $simp$)
}
moreover
{
assume $\neg\ n = 0$
hence $n > 0$ **by** $simp$
then obtain m **where** $ndef$: $n = Suc\ m$ **by** ($auto\ simp\ add$: $gr0\text{-}conv\text{-}Suc$)

from $newInt\text{-}ex$ **have**
$\exists\, a\ b.\ a < b \wedge (newInt\ (Suc\ m)\ f) = closed\text{-}int\ a\ b \wedge$
$newInt\ (Suc\ m)\ f \subseteq newInt\ m\ f \wedge f\ (Suc\ m) \notin newInt\ (Suc\ m)\ f$ **.**
then have $f\ (Suc\ m) \notin newInt\ (Suc\ m)\ f$ **by** $auto$
with $ndef$ **have** $f\ n \notin newInt\ n\ f$ **by** $simp$
}
ultimately have $f\ n \notin newInt\ n\ f$ **by** ($rule\ case\text{-}split$)
thus $f\ n \notin (\bigcap n.\ newInt\ n\ f)$ **by** $auto$
qed

lemma $newInt\text{-}notempty$:
$(\bigcap n.\ newInt\ n\ f) \neq \{\}$
proof −
let $?g = \lambda n.\ newInt\ n\ f$
have $\forall\, n.\ ?g\ (Suc\ n) \subseteq ?g\ n$
proof
fix n
show $?g\ (Suc\ n) \subseteq ?g\ n$ **by** ($rule\ newInt\text{-}subset$)
qed
moreover have $\forall\, n.\ \exists\, a\ b.\ ?g\ n = closed\text{-}int\ a\ b \wedge a \leq b$
proof
fix n::nat
{
assume $n = 0$
then have
$?g\ n = closed\text{-}int\ (f\ 0 + 1)\ (f\ 0 + 2) \wedge (f\ 0 + 1 \leq f\ 0 + 2)$
by $simp$
hence $\exists\, a\ b.\ ?g\ n = closed\text{-}int\ a\ b \wedge a \leq b$ **by** $blast$
}
moreover
{
assume $\neg\ n = 0$
then have $n > 0$ **by** $simp$
then obtain m **where** nd: $n = Suc\ m$ **by** ($auto\ simp\ add$: $gr0\text{-}conv\text{-}Suc$)

have
$\exists\, a\ b.\ a < b \wedge (newInt\ (Suc\ m)\ f) = closed\text{-}int\ a\ b \wedge$
$(newInt\ (Suc\ m)\ f) \subseteq (newInt\ m\ f) \wedge (f\ (Suc\ m)) \notin (newInt\ (Suc\ m)\ f)$
by (*rule newInt-ex*)
then obtain a **and** b **where**
$a < b \wedge (newInt\ (Suc\ m)\ f) = closed\text{-}int\ a\ b$ **by** *auto*
with *nd* **have** $?g\ n = closed\text{-}int\ a\ b \wedge a \leq b$ **by** *auto*
hence $\exists\, a\ b.\ ?g\ n = closed\text{-}int\ a\ b \wedge a \leq b$ **by** *blast*
}
ultimately show $\exists\, a\ b.\ ?g\ n = closed\text{-}int\ a\ b \wedge a \leq b$ **by** (*rule case-split*)
qed
ultimately show *?thesis* **by** (*rule NIP*)
qed

13.6 Final Theorem

theorem *real-non-denum*:
shows $\neg\ (\exists f{::}nat \Rightarrow real.\ surj\ f)$
proof — by contradiction
assume $\exists f{::}nat \Rightarrow real.\ surj\ f$
then obtain $f{::}nat \Rightarrow real$ **where** $surj\ f$ **by** *auto*
hence *rangeF*: $range\ f = UNIV$ **by** (*rule surj-range*)
— We now produce a real number x that is not in the range of f, using the properties of newInt.
have $\exists\, x.\ x \in (\bigcap n.\ newInt\ n\ f)$ **using** *newInt-notempty* **by** *blast*
moreover have $\forall\, n.\ f\ n \notin (\bigcap n.\ newInt\ n\ f)$ **by** (*rule newInt-inter*)
ultimately obtain x **where** $x \in (\bigcap n.\ newInt\ n\ f)$ **and** $\forall\, n.\ f\ n \neq x$ **by** *blast*
moreover from *rangeF* **have** $x \in range\ f$ **by** *simp*
ultimately show *False* **by** *blast*
qed

end

14 Natural powers theory

theory *RealPow*
imports *RealDef*
begin

declare *abs-mult-self* [*simp*]

instance *real* :: *power* **..**

primrec (*realpow*)
realpow-0: $r\ \hat{}\ 0 = 1$
realpow-Suc: $r\ \hat{}\ (Suc\ n) = (r{::}real) * (r\ \hat{}\ n)$

instance *real* :: *recpower*
proof
  fix *z* :: *real*
  fix *n* :: *nat*
  show *z^0 = 1* **by** *simp*
  show *z^(Suc n) = z * (z^n)* **by** *simp*
qed

lemma *two-realpow-ge-one* [*simp*]: (*1*::*real*) ≤ *2* ^ *n*
by (*rule power-increasing*[*of 0 n 2*::*real*, *simplified*])

lemma *two-realpow-gt* [*simp*]: *real* (*n*::*nat*) < *2* ^ *n*
apply (*induct n*)
apply (*auto simp add*: *real-of-nat-Suc*)
apply (*subst mult-2*)
apply (*rule add-less-le-mono*)
apply (*auto simp add*: *two-realpow-ge-one*)
done

lemma *realpow-Suc-le-self*: [| *0* ≤ *r*; *r* ≤ (*1*::*real*) |] ==> *r* ^ *Suc n* ≤ *r*
by (*insert power-decreasing* [*of 1 Suc n r*], *simp*)

lemma *realpow-minus-mult* [*rule-format*]:
    0 < *n* --> (*x*::*real*) ^ (*n* − *1*) * *x* = *x* ^ *n*
apply (*simp split add*: *nat-diff-split*)
done

lemma *realpow-two-mult-inverse* [*simp*]:
    r ≠ *0* ==> *r* * *inverse r* ^*Suc* (*Suc 0*) = *inverse* (*r*::*real*)
by (*simp add*: *real-mult-assoc* [*symmetric*])

lemma *realpow-two-minus* [*simp*]: (−*x*) ^*Suc* (*Suc 0*) = (*x*::*real*) ^*Suc* (*Suc 0*)
by *simp*

lemma *realpow-two-diff*:
    (*x*::*real*) ^*Suc* (*Suc 0*) − *y* ^*Suc* (*Suc 0*) = (*x* − *y*) * (*x* + *y*)
apply (*unfold real-diff-def*)
apply (*simp add*: *ring-simps*)
done

lemma *realpow-two-disj*:
    ((*x*::*real*) ^*Suc* (*Suc 0*) = *y* ^*Suc* (*Suc 0*)) = (*x* = *y* | *x* = −*y*)
apply (*cut-tac x* = *x* **and** *y* = *y* **in** *realpow-two-diff*)
apply (*auto simp del*: *realpow-Suc*)
done

lemma *realpow-real-of-nat*: *real* (*m*::*nat*) ^ *n* = *real* (*m* ^ *n*)
apply (*induct n*)

apply (*auto simp add*: *real-of-nat-one real-of-nat-mult*)
done

lemma *realpow-real-of-nat-two-pos* [*simp*] : *0 < real* (*Suc* (*Suc 0*) ^ *n*)
apply (*induct n*)
apply (*auto simp add*: *real-of-nat-mult zero-less-mult-iff*)
done

lemma *realpow-increasing*:
[|(*0*::*real*) ≤ *x*; *0* ≤ *y*; *x* ^ *Suc n* ≤ *y* ^ *Suc n*|] ==> *x* ≤ *y*
by (*rule power-le-imp-le-base*)

14.1 Literal Arithmetic Involving Powers, Type *real*

lemma *real-of-int-power*: *real* (*x*::*int*) ^ *n* = *real* (*x* ^ *n*)
apply (*induct n*)
apply (*simp-all add*: *nat-mult-distrib*)
done
declare *real-of-int-power* [*symmetric*, *simp*]

lemma *power-real-number-of*:
(*number-of v* :: *real*) ^ *n* = *real* ((*number-of v* :: *int*) ^ *n*)
by (*simp only*: *real-number-of* [*symmetric*] *real-of-int-power*)

declare *power-real-number-of* [*of* - *number-of w*, *standard*, *simp*]

14.2 Properties of Squares

lemma *sum-squares-ge-zero*:
fixes *x y* :: *'a*::*ordered-ring-strict*
shows *0* ≤ *x* * *x* + *y* * *y*
by (*intro add-nonneg-nonneg zero-le-square*)

lemma *not-sum-squares-lt-zero*:
fixes *x y* :: *'a*::*ordered-ring-strict*
shows ¬ *x* * *x* + *y* * *y* < *0*
by (*simp add*: *linorder-not-less sum-squares-ge-zero*)

lemma *sum-nonneg-eq-zero-iff*:
fixes *x y* :: *'a*::*pordered-ab-group-add*
assumes *x*: *0* ≤ *x* **and** *y*: *0* ≤ *y*
shows (*x* + *y* = *0*) = (*x* = *0* ∧ *y* = *0*)
proof (*auto*)
from *y* **have** *x* + *0* ≤ *x* + *y* **by** (*rule add-left-mono*)
also assume *x* + *y* = *0*
finally have *x* ≤ *0* **by** *simp*
thus *x* = *0* **using** *x* **by** (*rule order-antisym*)
next
from *x* **have** *0* + *y* ≤ *x* + *y* **by** (*rule add-right-mono*)

also assume $x + y = 0$
finally have $y \leq 0$ **by** *simp*
thus $y = 0$ **using** y **by** (*rule order-antisym*)
qed

lemma *sum-squares-eq-zero-iff*:
fixes $x\ y :: 'a$::*ordered-ring-strict*
shows $(x * x + y * y = 0) = (x = 0 \wedge y = 0)$
by (*simp add*: *sum-nonneg-eq-zero-iff*)

lemma *sum-squares-le-zero-iff*:
fixes $x\ y :: 'a$::*ordered-ring-strict*
shows $(x * x + y * y \leq 0) = (x = 0 \wedge y = 0)$
by (*simp add*: *order-le-less not-sum-squares-lt-zero sum-squares-eq-zero-iff*)

lemma *sum-squares-gt-zero-iff*:
fixes $x\ y :: 'a$::*ordered-ring-strict*
shows $(0 < x * x + y * y) = (x \neq 0 \vee y \neq 0)$
by (*simp add*: *order-less-le sum-squares-ge-zero sum-squares-eq-zero-iff*)

lemma *sum-power2-ge-zero*:
fixes $x\ y :: 'a$::{*ordered-idom*,*recpower*}
shows $0 \leq x^2 + y^2$
unfolding *power2-eq-square* **by** (*rule sum-squares-ge-zero*)

lemma *not-sum-power2-lt-zero*:
fixes $x\ y :: 'a$::{*ordered-idom*,*recpower*}
shows $\neg\ x^2 + y^2 < 0$
unfolding *power2-eq-square* **by** (*rule not-sum-squares-lt-zero*)

lemma *sum-power2-eq-zero-iff*:
fixes $x\ y :: 'a$::{*ordered-idom*,*recpower*}
shows $(x^2 + y^2 = 0) = (x = 0 \wedge y = 0)$
unfolding *power2-eq-square* **by** (*rule sum-squares-eq-zero-iff*)

lemma *sum-power2-le-zero-iff*:
fixes $x\ y :: 'a$::{*ordered-idom*,*recpower*}
shows $(x^2 + y^2 \leq 0) = (x = 0 \wedge y = 0)$
unfolding *power2-eq-square* **by** (*rule sum-squares-le-zero-iff*)

lemma *sum-power2-gt-zero-iff*:
fixes $x\ y :: 'a$::{*ordered-idom*,*recpower*}
shows $(0 < x^2 + y^2) = (x \neq 0 \vee y \neq 0)$
unfolding *power2-eq-square* **by** (*rule sum-squares-gt-zero-iff*)

14.3 Squares of Reals

lemma *real-two-squares-add-zero-iff* [*simp*]:
$(x * x + y * y = 0) = ((x::real) = 0 \wedge y = 0)$

by (*rule sum-squares-eq-zero-iff*)

lemma *real-sum-squares-cancel*: $x * x + y * y = 0 ==> x = (0::real)$
by *simp*

lemma *real-sum-squares-cancel2*: $x * x + y * y = 0 ==> y = (0::real)$
by *simp*

lemma *real-mult-self-sum-ge-zero*: $(0::real) \leq x*x + y*y$
by (*rule sum-squares-ge-zero*)

lemma *real-sum-squares-cancel-a*: $x * x = -(y * y) ==> x = (0::real)$ & $y=0$
by (*simp add*: *real-add-eq-0-iff* [*symmetric*])

lemma *real-squared-diff-one-factored*: $x*x - (1::real) = (x + 1)*(x - 1)$
by (*simp add*: *left-distrib right-diff-distrib*)

lemma *real-mult-is-one* [*simp*]: $(x*x = (1::real)) = (x = 1 \mid x = -1)$
apply *auto*
apply (*drule right-minus-eq* [*THEN iffD2*])
apply (*auto simp add*: *real-squared-diff-one-factored*)
done

lemma *real-sum-squares-not-zero*: $x \sim= 0 ==> x * x + y * y \sim= (0::real)$
by *simp*

lemma *real-sum-squares-not-zero2*: $y \sim= 0 ==> x * x + y * y \sim= (0::real)$
by *simp*

lemma *realpow-two-sum-zero-iff* [*simp*]:
$(x \hat{} 2 + y \hat{} 2 = (0::real)) = (x = 0 \ \& \ y = 0)$
by (*rule sum-power2-eq-zero-iff*)

lemma *realpow-two-le-add-order* [*simp*]: $(0::real) \leq u \hat{} 2 + v \hat{} 2$
by (*rule sum-power2-ge-zero*)

lemma *realpow-two-le-add-order2* [*simp*]: $(0::real) \leq u \hat{} 2 + v \hat{} 2 + w \hat{} 2$
by (*intro add-nonneg-nonneg zero-le-power2*)

lemma *real-sum-square-gt-zero*: $x \sim= 0 ==> (0::real) < x * x + y * y$
by (*simp add*: *sum-squares-gt-zero-iff*)

lemma *real-sum-square-gt-zero2*: $y \sim= 0 ==> (0::real) < x * x + y * y$
by (*simp add*: *sum-squares-gt-zero-iff*)

lemma *real-minus-mult-self-le* [*simp*]: $-(u * u) \leq (x * (x::real))$
by (*rule-tac* $j = 0$ **in** *real-le-trans*, *auto*)

lemma *realpow-square-minus-le* [*simp*]: $-(u \hat{} 2) \leq (x::real) \hat{} 2$

by (*auto simp add*: *power2-eq-square*)

lemma *real-sq-order*:
  fixes *x*::*real*
  assumes *xgt0*: *0* ≤ *x* **and** *ygt0*: *0* ≤ *y* **and** *sq*: *x^2* ≤ *y^2*
  shows *x* ≤ *y*
proof −
  from *sq* **have** *x* ^ *Suc* (*Suc 0*) ≤ *y* ^ *Suc* (*Suc 0*)
    by (*simp only*: *numeral-2-eq-2*)
  thus *x* ≤ *y* **using** *ygt0*
    by (*rule power-le-imp-le-base*)
qed

14.4 Various Other Theorems

lemma *real-le-add-half-cancel*: (*x* + *y*/*2* ≤ (*y*::*real*)) = (*x* ≤ *y* /*2*)
by *auto*

lemma *real-minus-half-eq* [*simp*]: (*x*::*real*) − *x*/*2* = *x*/*2*
by *auto*

lemma *real-mult-inverse-cancel*:
    [|(*0*::*real*) < *x*; *0* < *x1*; *x1* ∗ *y* < *x* ∗ *u* |]
     ==> *inverse x* ∗ *y* < *inverse x1* ∗ *u*
apply (*rule-tac c*=*x* **in** *mult-less-imp-less-left*)
apply (*auto simp add*: *real-mult-assoc* [*symmetric*])
apply (*simp* (*no-asm*) *add*: *mult-ac*)
apply (*rule-tac c*=*x1* **in** *mult-less-imp-less-right*)
apply (*auto simp add*: *mult-ac*)
done

lemma *real-mult-inverse-cancel2*:
    [|(*0*::*real*) < *x*;*0* < *x1*; *x1* ∗ *y* < *x* ∗ *u* |] ==> *y* ∗ *inverse x* < *u* ∗ *inverse x1*
apply (*auto dest*: *real-mult-inverse-cancel simp add*: *mult-ac*)
done

lemma *inverse-real-of-nat-gt-zero* [*simp*]: *0* < *inverse* (*real* (*Suc n*))
by *simp*

lemma *inverse-real-of-nat-ge-zero* [*simp*]: *0* ≤ *inverse* (*real* (*Suc n*))
by *simp*

lemma *realpow-num-eq-if*: (*m*::*real*) ^ *n* = (*if n=0 then 1 else m* ∗ *m* ^ (*n* − *1*))
by (*case-tac n*, *auto*)

end

15 Vector Spaces and Algebras over the Reals

theory *RealVector*
imports *RealPow*
begin

15.1 Locale for additive functions

locale *additive* =
  fixes f :: $'a$::*ab-group-add* ⇒ $'b$::*ab-group-add*
  assumes *add*: $f\ (x + y) = f\ x + f\ y$

lemma (**in** *additive*) *zero*: $f\ 0 = 0$
proof −
  have $f\ 0 = f\ (0 + 0)$ **by** *simp*
  also have $\ldots = f\ 0 + f\ 0$ **by** (*rule add*)
  finally show $f\ 0 = 0$ **by** *simp*
qed

lemma (**in** *additive*) *minus*: $f\ (-\ x) = -\ f\ x$
proof −
  have $f\ (-\ x) + f\ x = f\ (-\ x + x)$ **by** (*rule add* [*symmetric*])
  also have $\ldots = -\ f\ x + f\ x$ **by** (*simp add*: *zero*)
  finally show $f\ (-\ x) = -\ f\ x$ **by** (*rule add-right-imp-eq*)
qed

lemma (**in** *additive*) *diff*: $f\ (x - y) = f\ x - f\ y$
by (*simp add*: *diff-def add minus*)

lemma (**in** *additive*) *setsum*: $f\ (setsum\ g\ A) = (\sum x{\in}A.\ f\ (g\ x))$
apply (*cases finite A*)
apply (*induct set*: *finite*)
apply (*simp add*: *zero*)
apply (*simp add*: *add*)
apply (*simp add*: *zero*)
done

15.2 Real vector spaces

class *scaleR* = *type* +
  fixes *scaleR* :: *real* ⇒ $'a$ ⇒ $'a$ (**infixr** $*_R$ 75)
begin

abbreviation
  divideR :: $'a$ ⇒ *real* ⇒ $'a$ (**infixl** $'/_R$ 70)
where
  $x\ /_R\ r == scaleR\ (inverse\ r)\ x$

end

instance *real* :: *scaleR*
real-scaleR-def [*simp*]: *scaleR a x* ≡ *a* ∗ *x* **..**

class *real-vector* = *scaleR* + *ab-group-add* +
assumes *scaleR-right-distrib*: *scaleR a* (*x* + *y*) = *scaleR a x* + *scaleR a y*
and *scaleR-left-distrib*: *scaleR* (*a* + *b*) *x* = *scaleR a x* + *scaleR b x*
and *scaleR-scaleR* [*simp*]: *scaleR a* (*scaleR b x*) = *scaleR* (*a* ∗ *b*) *x*
and *scaleR-one* [*simp*]: *scaleR 1 x* = *x*

class *real-algebra* = *real-vector* + *ring* +
assumes *mult-scaleR-left* [*simp*]: *scaleR a x* ∗ *y* = *scaleR a* (*x* ∗ *y*)
and *mult-scaleR-right* [*simp*]: *x* ∗ *scaleR a y* = *scaleR a* (*x* ∗ *y*)

class *real-algebra-1* = *real-algebra* + *ring-1*

class *real-div-algebra* = *real-algebra-1* + *division-ring*

class *real-field* = *real-div-algebra* + *field*

instance *real* :: *real-field*
apply (*intro-classes*, *unfold real-scaleR-def*)
apply (*rule right-distrib*)
apply (*rule left-distrib*)
apply (*rule mult-assoc* [*symmetric*])
apply (*rule mult-1-left*)
apply (*rule mult-assoc*)
apply (*rule mult-left-commute*)
done

lemma *scaleR-left-commute*:
fixes *x* :: *'a*::*real-vector*
shows *scaleR a* (*scaleR b x*) = *scaleR b* (*scaleR a x*)
by (*simp add*: *mult-commute*)

interpretation *scaleR-left*: *additive* [(λ*a*. *scaleR a x*::*'a*::*real-vector*)]
by *unfold-locales* (*rule scaleR-left-distrib*)

interpretation *scaleR-right*: *additive* [(λ*x*. *scaleR a x*::*'a*::*real-vector*)]
by *unfold-locales* (*rule scaleR-right-distrib*)

lemmas *scaleR-zero-left* [*simp*] = *scaleR-left.zero*

lemmas *scaleR-zero-right* [*simp*] = *scaleR-right.zero*

lemmas *scaleR-minus-left* [*simp*] = *scaleR-left.minus*

lemmas *scaleR-minus-right* [*simp*] = *scaleR-right.minus*

lemmas *scaleR-left-diff-distrib* = *scaleR-left.diff*

lemmas *scaleR-right-diff-distrib* = *scaleR-right.diff*

lemma *scaleR-eq-0-iff* [*simp*]:
fixes *x* :: *'a*::*real-vector*
shows (*scaleR a x* = *0*) = (*a* = *0* ∨ *x* = *0*)
proof *cases*
assume *a* = *0* **thus** *?thesis* **by** *simp*
next
assume *anz* [*simp*]: *a* ≠ *0*
{ assume *scaleR a x* = *0*
hence *scaleR* (*inverse a*) (*scaleR a x*) = *0* **by** *simp*
hence *x* = *0* **by** *simp* **}**
thus *?thesis* **by** *force*
qed

lemma *scaleR-left-imp-eq*:
fixes *x y* :: *'a*::*real-vector*
shows ⟦*a* ≠ *0*; *scaleR a x* = *scaleR a y*⟧ ⟹ *x* = *y*
proof −
assume *nonzero*: *a* ≠ *0*
assume *scaleR a x* = *scaleR a y*
hence *scaleR a* (*x* − *y*) = *0*
by (*simp add*: *scaleR-right-diff-distrib*)
hence *x* − *y* = *0* **by** (*simp add*: *nonzero*)
thus *x* = *y* **by** *simp*
qed

lemma *scaleR-right-imp-eq*:
fixes *x y* :: *'a*::*real-vector*
shows ⟦*x* ≠ *0*; *scaleR a x* = *scaleR b x*⟧ ⟹ *a* = *b*
proof −
assume *nonzero*: *x* ≠ *0*
assume *scaleR a x* = *scaleR b x*
hence *scaleR* (*a* − *b*) *x* = *0*
by (*simp add*: *scaleR-left-diff-distrib*)
hence *a* − *b* = *0* **by** (*simp add*: *nonzero*)
thus *a* = *b* **by** *simp*
qed

lemma *scaleR-cancel-left*:
fixes *x y* :: *'a*::*real-vector*
shows (*scaleR a x* = *scaleR a y*) = (*x* = *y* ∨ *a* = *0*)
by (*auto intro*: *scaleR-left-imp-eq*)

lemma *scaleR-cancel-right*:
fixes *x y* :: *'a*::*real-vector*
shows (*scaleR a x* = *scaleR b x*) = (*a* = *b* ∨ *x* = *0*)
by (*auto intro*: *scaleR-right-imp-eq*)

lemma *nonzero-inverse-scaleR-distrib*:
fixes x :: $'a$::*real-div-algebra* **shows**
$[\![a \neq 0; x \neq 0]\!] \Longrightarrow$ *inverse* (*scaleR a x*) = *scaleR* (*inverse a*) (*inverse x*)
by (*rule inverse-unique*, *simp*)

lemma *inverse-scaleR-distrib*:
fixes x :: $'a$::{*real-div-algebra*,*division-by-zero*}
shows *inverse* (*scaleR a x*) = *scaleR* (*inverse a*) (*inverse x*)
apply (*case-tac a = 0*, *simp*)
apply (*case-tac x = 0*, *simp*)
apply (*erule* (*1*) *nonzero-inverse-scaleR-distrib*)
done

15.3 Embedding of the Reals into any *real-algebra-1*: *of-real*

definition
of-real :: *real* ⇒ $'a$::*real-algebra-1* **where**
of-real r = *scaleR r 1*

lemma *scaleR-conv-of-real*: *scaleR r x* = *of-real r* ∗ *x*
by (*simp add*: *of-real-def*)

lemma *of-real-0* [*simp*]: *of-real 0* = *0*
by (*simp add*: *of-real-def*)

lemma *of-real-1* [*simp*]: *of-real 1* = *1*
by (*simp add*: *of-real-def*)

lemma *of-real-add* [*simp*]: *of-real* (*x* + *y*) = *of-real x* + *of-real y*
by (*simp add*: *of-real-def scaleR-left-distrib*)

lemma *of-real-minus* [*simp*]: *of-real* (− *x*) = − *of-real x*
by (*simp add*: *of-real-def*)

lemma *of-real-diff* [*simp*]: *of-real* (*x* − *y*) = *of-real x* − *of-real y*
by (*simp add*: *of-real-def scaleR-left-diff-distrib*)

lemma *of-real-mult* [*simp*]: *of-real* (*x* ∗ *y*) = *of-real x* ∗ *of-real y*
by (*simp add*: *of-real-def mult-commute*)

lemma *nonzero-of-real-inverse*:
x ≠ *0* ⟹ *of-real* (*inverse x*) =
inverse (*of-real x* :: $'a$::*real-div-algebra*)
by (*simp add*: *of-real-def nonzero-inverse-scaleR-distrib*)

lemma *of-real-inverse* [*simp*]:
of-real (*inverse x*) =
inverse (*of-real x* :: $'a$::{*real-div-algebra*,*division-by-zero*})

by (*simp add*: *of-real-def inverse-scaleR-distrib*)

lemma *nonzero-of-real-divide*:
  $y \neq 0 \Longrightarrow$ *of-real* $(x\ /\ y) =$
   (*of-real* x / *of-real* $y :: 'a$::*real-field*)
by (*simp add*: *divide-inverse nonzero-of-real-inverse*)

lemma *of-real-divide* [*simp*]:
  of-real $(x\ /\ y) =$
   (*of-real* x / *of-real* $y :: 'a$::{*real-field*,*division-by-zero*})
by (*simp add*: *divide-inverse*)

lemma *of-real-power* [*simp*]:
  of-real $(x\ \hat{}\ n) =$ (*of-real* $x :: 'a$::{*real-algebra-1*,*recpower*}) $\hat{}\ n$
by (*induct n*) (*simp-all add*: *power-Suc*)

lemma *of-real-eq-iff* [*simp*]: (*of-real* x = *of-real* y) = ($x = y$)
by (*simp add*: *of-real-def scaleR-cancel-right*)

lemmas *of-real-eq-0-iff* [*simp*] = *of-real-eq-iff* [*of - 0*, *simplified*]

lemma *of-real-eq-id* [*simp*]: *of-real* = (*id* :: *real* ⇒ *real*)
proof
  fix r
  show *of-real* r = *id* r
    by (*simp add*: *of-real-def*)
qed

Collapse nested embeddings

lemma *of-real-of-nat-eq* [*simp*]: *of-real* (*of-nat* n) = *of-nat* n
by (*induct n*) *auto*

lemma *of-real-of-int-eq* [*simp*]: *of-real* (*of-int* z) = *of-int* z
by (*cases z rule*: *int-diff-cases*, *simp*)

lemma *of-real-number-of-eq*:
  of-real (*number-of* w) = (*number-of* $w :: 'a$::{*number-ring*,*real-algebra-1*})
by (*simp add*: *number-of-eq*)

Every real algebra has characteristic zero

instance *real-algebra-1* < *ring-char-0*
proof
  fix $m\ n$ :: *nat*
  have (*of-real* (*of-nat* m) = (*of-real* (*of-nat* n)::$'a$)) = ($m = n$)
    by (*simp only*: *of-real-eq-iff of-nat-eq-iff*)
  thus (*of-nat* m = (*of-nat* n::$'a$)) = ($m = n$)
    by (*simp only*: *of-real-of-nat-eq*)
qed

15.4 The Set of Real Numbers

definition
Reals :: *'a*::*real-algebra-1 set* **where**
Reals ≡ *range of-real*

notation (*xsymbols*)
Reals (ℝ)

lemma *Reals-of-real* [*simp*]: *of-real r* ∈ *Reals*
by (*simp add*: *Reals-def*)

lemma *Reals-of-int* [*simp*]: *of-int z* ∈ *Reals*
by (*subst of-real-of-int-eq* [*symmetric*], *rule Reals-of-real*)

lemma *Reals-of-nat* [*simp*]: *of-nat n* ∈ *Reals*
by (*subst of-real-of-nat-eq* [*symmetric*], *rule Reals-of-real*)

lemma *Reals-number-of* [*simp*]:
(*number-of w*::*'a*::{*number-ring*,*real-algebra-1*}) ∈ *Reals*
by (*subst of-real-number-of-eq* [*symmetric*], *rule Reals-of-real*)

lemma *Reals-0* [*simp*]: *0* ∈ *Reals*
apply (*unfold Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-0* [*symmetric*])
done

lemma *Reals-1* [*simp*]: *1* ∈ *Reals*
apply (*unfold Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-1* [*symmetric*])
done

lemma *Reals-add* [*simp*]: ⟦*a* ∈ *Reals*; *b* ∈ *Reals*⟧ ⟹ *a* + *b* ∈ *Reals*
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-add* [*symmetric*])
done

lemma *Reals-minus* [*simp*]: *a* ∈ *Reals* ⟹ − *a* ∈ *Reals*
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-minus* [*symmetric*])
done

lemma *Reals-diff* [*simp*]: ⟦*a* ∈ *Reals*; *b* ∈ *Reals*⟧ ⟹ *a* − *b* ∈ *Reals*
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-diff* [*symmetric*])

done

lemma *Reals-mult* [*simp*]: $[\![a \in Reals;\ b \in Reals]\!] \Longrightarrow a * b \in Reals$
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-mult* [*symmetric*])
done

lemma *nonzero-Reals-inverse*:
  fixes *a* :: *'a*::*real-div-algebra*
  shows $[\![a \in Reals;\ a \neq 0]\!] \Longrightarrow inverse\ a \in Reals$
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*erule nonzero-of-real-inverse* [*symmetric*])
done

lemma *Reals-inverse* [*simp*]:
  fixes *a* :: *'a*::{*real-div-algebra*,*division-by-zero*}
  shows $a \in Reals \Longrightarrow inverse\ a \in Reals$
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-inverse* [*symmetric*])
done

lemma *nonzero-Reals-divide*:
  fixes *a b* :: *'a*::*real-field*
  shows $[\![a \in Reals;\ b \in Reals;\ b \neq 0]\!] \Longrightarrow a\ /\ b \in Reals$
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*erule nonzero-of-real-divide* [*symmetric*])
done

lemma *Reals-divide* [*simp*]:
  fixes *a b* :: *'a*::{*real-field*,*division-by-zero*}
  shows $[\![a \in Reals;\ b \in Reals]\!] \Longrightarrow a\ /\ b \in Reals$
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-divide* [*symmetric*])
done

lemma *Reals-power* [*simp*]:
  fixes *a* :: *'a*::{*real-algebra-1*,*recpower*}
  shows $a \in Reals \Longrightarrow a\ \hat{}\ n \in Reals$
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-power* [*symmetric*])
done

lemma *Reals-cases* [*cases set*: *Reals*]:

assumes $q \in \mathbb{R}$
obtains (*of-real*) r **where** $q = \mathit{of\text{-}real}\ r$
unfolding *Reals-def*
proof −
from ‹$q \in \mathbb{R}$› **have** $q \in \mathit{range}\ \mathit{of\text{-}real}$ **unfolding** *Reals-def* **.**
then obtain r **where** $q = \mathit{of\text{-}real}\ r$ **..**
then show *thesis* **..**
qed

lemma *Reals-induct* [*case-names of-real*, *induct set*: *Reals*]:
$q \in \mathbb{R} \Longrightarrow (\bigwedge r.\ P\ (\mathit{of\text{-}real}\ r)) \Longrightarrow P\ q$
by (*rule Reals-cases*) *auto*

15.5 Real normed vector spaces

class *norm* = *type* +
fixes *norm* :: $'a \Rightarrow \mathit{real}$

instance *real* :: *norm*
real-norm-def [*simp*]: *norm* $r \equiv |r|$ **..**

class *sgn-div-norm* = *scaleR* + *norm* + *sgn* +
assumes *sgn-div-norm*: *sgn* $x = x\ /_R\ \mathit{norm}\ x$

class *real-normed-vector* = *real-vector* + *sgn-div-norm* +
assumes *norm-ge-zero* [*simp*]: $0 \leq \mathit{norm}\ x$
and *norm-eq-zero* [*simp*]: $\mathit{norm}\ x = 0 \longleftrightarrow x = 0$
and *norm-triangle-ineq*: $\mathit{norm}\ (x + y) \leq \mathit{norm}\ x + \mathit{norm}\ y$
and *norm-scaleR*: $\mathit{norm}\ (\mathit{scaleR}\ a\ x) = |a| * \mathit{norm}\ x$

class *real-normed-algebra* = *real-algebra* + *real-normed-vector* +
assumes *norm-mult-ineq*: $\mathit{norm}\ (x * y) \leq \mathit{norm}\ x * \mathit{norm}\ y$

class *real-normed-algebra-1* = *real-algebra-1* + *real-normed-algebra* +
assumes *norm-one* [*simp*]: *norm* 1 = 1

class *real-normed-div-algebra* = *real-div-algebra* + *real-normed-vector* +
assumes *norm-mult*: $\mathit{norm}\ (x * y) = \mathit{norm}\ x * \mathit{norm}\ y$

class *real-normed-field* = *real-field* + *real-normed-div-algebra*

instance *real-normed-div-algebra* < *real-normed-algebra-1*
proof
fix $x\ y$ :: $'a$
show $\mathit{norm}\ (x * y) \leq \mathit{norm}\ x * \mathit{norm}\ y$
by (*simp add*: *norm-mult*)
next
have $\mathit{norm}\ (1 * 1::'a) = \mathit{norm}\ (1::'a) * \mathit{norm}\ (1::'a)$
by (*rule norm-mult*)

thus *norm* (*1*::*'a*) = *1* **by** *simp*
qed

instance *real* :: *real-normed-field*
apply (*intro-classes*, *unfold real-norm-def real-scaleR-def*)
apply (*simp add*: *real-sgn-def*)
apply (*rule abs-ge-zero*)
apply (*rule abs-eq-0*)
apply (*rule abs-triangle-ineq*)
apply (*rule abs-mult*)
apply (*rule abs-mult*)
done

lemma *norm-zero* [*simp*]: *norm* (*0*::*'a*::*real-normed-vector*) = *0*
by *simp*

lemma *zero-less-norm-iff* [*simp*]:
fixes *x* :: *'a*::*real-normed-vector*
shows (*0* < *norm x*) = (*x* ≠ *0*)
by (*simp add*: *order-less-le*)

lemma *norm-not-less-zero* [*simp*]:
fixes *x* :: *'a*::*real-normed-vector*
shows ¬ *norm x* < *0*
by (*simp add*: *linorder-not-less*)

lemma *norm-le-zero-iff* [*simp*]:
fixes *x* :: *'a*::*real-normed-vector*
shows (*norm x* ≤ *0*) = (*x* = *0*)
by (*simp add*: *order-le-less*)

lemma *norm-minus-cancel* [*simp*]:
fixes *x* :: *'a*::*real-normed-vector*
shows *norm* (− *x*) = *norm x*
proof −
have *norm* (− *x*) = *norm* (*scaleR* (− *1*) *x*)
by (*simp only*: *scaleR-minus-left scaleR-one*)
also have ... = |− *1*| ∗ *norm x*
by (*rule norm-scaleR*)
finally show *?thesis* **by** *simp*
qed

lemma *norm-minus-commute*:
fixes *a b* :: *'a*::*real-normed-vector*
shows *norm* (*a* − *b*) = *norm* (*b* − *a*)
proof −
have *norm* (− (*b* − *a*)) = *norm* (*b* − *a*)
by (*rule norm-minus-cancel*)
thus *?thesis* **by** *simp*

qed

lemma *norm-triangle-ineq2*:
fixes *a b* :: *'a*::*real-normed-vector*
shows *norm a − norm b ≤ norm* (*a − b*)
proof −
have *norm* (*a − b* + *b*) ≤ *norm* (*a − b*) + *norm b*
by (*rule norm-triangle-ineq*)
thus *?thesis* **by** *simp*
qed

lemma *norm-triangle-ineq3*:
fixes *a b* :: *'a*::*real-normed-vector*
shows |*norm a − norm b*| ≤ *norm* (*a − b*)
apply (*subst abs-le-iff*)
apply *auto*
apply (*rule norm-triangle-ineq2*)
apply (*subst norm-minus-commute*)
apply (*rule norm-triangle-ineq2*)
done

lemma *norm-triangle-ineq4*:
fixes *a b* :: *'a*::*real-normed-vector*
shows *norm* (*a − b*) ≤ *norm a* + *norm b*
proof −
have *norm* (*a* + − *b*) ≤ *norm a* + *norm* (− *b*)
by (*rule norm-triangle-ineq*)
thus *?thesis*
by (*simp only*: *diff-minus norm-minus-cancel*)
qed

lemma *norm-diff-ineq*:
fixes *a b* :: *'a*::*real-normed-vector*
shows *norm a − norm b ≤ norm* (*a* + *b*)
proof −
have *norm a − norm* (− *b*) ≤ *norm* (*a* − − *b*)
by (*rule norm-triangle-ineq2*)
thus *?thesis* **by** *simp*
qed

lemma *norm-diff-triangle-ineq*:
fixes *a b c d* :: *'a*::*real-normed-vector*
shows *norm* ((*a* + *b*) − (*c* + *d*)) ≤ *norm* (*a − c*) + *norm* (*b − d*)
proof −
have *norm* ((*a* + *b*) − (*c* + *d*)) = *norm* ((*a − c*) + (*b − d*))
by (*simp add*: *diff-minus add-ac*)
also have ... ≤ *norm* (*a − c*) + *norm* (*b − d*)
by (*rule norm-triangle-ineq*)
finally show *?thesis* **.**

qed

lemma *abs-norm-cancel* [*simp*]:
fixes *a* :: *'a*::*real-normed-vector*
shows |*norm a*| = *norm a*
by (*rule abs-of-nonneg* [*OF norm-ge-zero*])

lemma *norm-add-less*:
fixes *x y* :: *'a*::*real-normed-vector*
shows ⟦*norm x* < *r*; *norm y* < *s*⟧ ⟹ *norm* (*x* + *y*) < *r* + *s*
by (*rule order-le-less-trans* [*OF norm-triangle-ineq add-strict-mono*])

lemma *norm-mult-less*:
fixes *x y* :: *'a*::*real-normed-algebra*
shows ⟦*norm x* < *r*; *norm y* < *s*⟧ ⟹ *norm* (*x* * *y*) < *r* * *s*
apply (*rule order-le-less-trans* [*OF norm-mult-ineq*])
apply (*simp add*: *mult-strict-mono'*)
done

lemma *norm-of-real* [*simp*]:
norm (*of-real r* :: *'a*::*real-normed-algebra-1*) = |*r*|
unfolding *of-real-def* **by** (*simp add*: *norm-scaleR*)

lemma *norm-number-of* [*simp*]:
norm (*number-of w*::*'a*::{*number-ring*,*real-normed-algebra-1*})
= |*number-of w*|
by (*subst of-real-number-of-eq* [*symmetric*], *rule norm-of-real*)

lemma *norm-of-int* [*simp*]:
norm (*of-int z*::*'a*::*real-normed-algebra-1*) = |*of-int z*|
by (*subst of-real-of-int-eq* [*symmetric*], *rule norm-of-real*)

lemma *norm-of-nat* [*simp*]:
norm (*of-nat n*::*'a*::*real-normed-algebra-1*) = *of-nat n*
apply (*subst of-real-of-nat-eq* [*symmetric*])
apply (*subst norm-of-real*, *simp*)
done

lemma *nonzero-norm-inverse*:
fixes *a* :: *'a*::*real-normed-div-algebra*
shows *a* ≠ *0* ⟹ *norm* (*inverse a*) = *inverse* (*norm a*)
apply (*rule inverse-unique* [*symmetric*])
apply (*simp add*: *norm-mult* [*symmetric*])
done

lemma *norm-inverse*:
fixes *a* :: *'a*::{*real-normed-div-algebra*,*division-by-zero*}
shows *norm* (*inverse a*) = *inverse* (*norm a*)
apply (*case-tac a* = *0*, *simp*)

apply (*erule nonzero-norm-inverse*)
done

lemma *nonzero-norm-divide*:
fixes *a b* :: *'a*::*real-normed-field*
shows $b \neq 0 \Longrightarrow norm\ (a\ /\ b) = norm\ a\ /\ norm\ b$
by (*simp add*: *divide-inverse norm-mult nonzero-norm-inverse*)

lemma *norm-divide*:
fixes *a b* :: *'a*::{*real-normed-field*,*division-by-zero*}
shows $norm\ (a\ /\ b) = norm\ a\ /\ norm\ b$
by (*simp add*: *divide-inverse norm-mult norm-inverse*)

lemma *norm-power-ineq*:
fixes *x* :: *'a*::{*real-normed-algebra-1*,*recpower*}
shows $norm\ (x\ \hat{}\ n) \leq norm\ x\ \hat{}\ n$
proof (*induct n*)
case *0* **show** $norm\ (x\ \hat{}\ 0) \leq norm\ x\ \hat{}\ 0$ **by** *simp*
next
case (*Suc n*)
have $norm\ (x * x\ \hat{}\ n) \leq norm\ x * norm\ (x\ \hat{}\ n)$
by (*rule norm-mult-ineq*)
also from *Suc* **have** $\ldots \leq norm\ x * norm\ x\ \hat{}\ n$
using *norm-ge-zero* **by** (*rule mult-left-mono*)
finally show $norm\ (x\ \hat{}\ Suc\ n) \leq norm\ x\ \hat{}\ Suc\ n$
by (*simp add*: *power-Suc*)
qed

lemma *norm-power*:
fixes *x* :: *'a*::{*real-normed-div-algebra*,*recpower*}
shows $norm\ (x\ \hat{}\ n) = norm\ x\ \hat{}\ n$
by (*induct n*) (*simp-all add*: *power-Suc norm-mult*)

15.6 Sign function

lemma *norm-sgn*:
$norm\ (sgn(x::'a::real\text{-}normed\text{-}vector)) = (if\ x = 0\ then\ 0\ else\ 1)$
by (*simp add*: *sgn-div-norm norm-scaleR*)

lemma *sgn-zero* [*simp*]: $sgn(0::'a::real\text{-}normed\text{-}vector) = 0$
by (*simp add*: *sgn-div-norm*)

lemma *sgn-zero-iff*: $(sgn(x::'a::real\text{-}normed\text{-}vector) = 0) = (x = 0)$
by (*simp add*: *sgn-div-norm*)

lemma *sgn-minus*: $sgn\ (-\ x) = -\ sgn(x::'a::real\text{-}normed\text{-}vector)$
by (*simp add*: *sgn-div-norm*)

lemma *sgn-scaleR*:

sgn (scaleR r x) = scaleR (sgn r) (sgn(x::'a::real-normed-vector))
by (*simp add*: *sgn-div-norm norm-scaleR mult-ac*)

lemma *sgn-one* [*simp*]: *sgn (1::'a::real-normed-algebra-1) = 1*
by (*simp add*: *sgn-div-norm*)

lemma *sgn-of-real*:
sgn (of-real r::'a::real-normed-algebra-1) = of-real (sgn r)
unfolding *of-real-def* **by** (*simp only*: *sgn-scaleR sgn-one*)

lemma *sgn-mult*:
fixes *x y* :: *'a::real-normed-div-algebra*
shows *sgn (x * y) = sgn x * sgn y*
by (*simp add*: *sgn-div-norm norm-mult mult-commute*)

lemma *real-sgn-eq*: *sgn (x::real) = x / |x|*
by (*simp add*: *sgn-div-norm divide-inverse*)

lemma *real-sgn-pos*: *0 < (x::real) ⟹ sgn x = 1*
unfolding *real-sgn-eq* **by** *simp*

lemma *real-sgn-neg*: *(x::real) < 0 ⟹ sgn x = −1*
unfolding *real-sgn-eq* **by** *simp*

15.7 Bounded Linear and Bilinear Operators

locale *bounded-linear = additive +*
constrains *f* :: *'a::real-normed-vector ⇒ 'b::real-normed-vector*
assumes *scaleR*: *f (scaleR r x) = scaleR r (f x)*
assumes *bounded*: *∃ K. ∀ x. norm (f x) ≤ norm x * K*

lemma (**in** *bounded-linear*) *pos-bounded*:
*∃ K>0. ∀ x. norm (f x) ≤ norm x * K*
proof −
obtain *K* **where** *K*: *⋀x. norm (f x) ≤ norm x * K*
using *bounded* **by** *fast*
show *?thesis*
proof (*intro exI impI conjI allI*)
show *0 < max 1 K*
by (*rule order-less-le-trans* [*OF zero-less-one le-maxI1*])
next
fix *x*
have *norm (f x) ≤ norm x * K* **using** *K* **.**
also have *... ≤ norm x * max 1 K*
by (*rule mult-left-mono* [*OF le-maxI2 norm-ge-zero*])
finally show *norm (f x) ≤ norm x * max 1 K* **.**
qed
qed

lemma (**in** *bounded-linear*) *nonneg-bounded*:
  $\exists K{\geq}0.\ \forall x.\ norm\ (f\ x) \leq norm\ x * K$
proof −
  from *pos-bounded*
  show *?thesis* **by** (*auto intro*: *order-less-imp-le*)
qed

locale *bounded-bilinear* =
  fixes *prod* :: ['*a*::*real-normed-vector*, '*b*::*real-normed-vector*]
            ⇒ '*c*::*real-normed-vector*
    (**infixl** ** 70*)
  assumes *add-left*: *prod* (*a* + *a'*) *b* = *prod a b* + *prod a' b*
  assumes *add-right*: *prod a* (*b* + *b'*) = *prod a b* + *prod a b'*
  assumes *scaleR-left*: *prod* (*scaleR r a*) *b* = *scaleR r* (*prod a b*)
  assumes *scaleR-right*: *prod a* (*scaleR r b*) = *scaleR r* (*prod a b*)
  assumes *bounded*: $\exists K.\ \forall a\ b.$ *norm* (*prod a b*) ≤ *norm a* * *norm b* * *K*

lemma (**in** *bounded-bilinear*) *pos-bounded*:
  $\exists K{>}0.\ \forall a\ b.$ *norm* (*a* ** *b*) ≤ *norm a* * *norm b* * *K*
apply (*cut-tac bounded*, *erule exE*)
apply (*rule-tac x*=*max 1 K* **in** *exI*, *safe*)
apply (*rule order-less-le-trans* [*OF zero-less-one le-maxI1*])
apply (*drule spec*, *drule spec*, *erule order-trans*)
apply (*rule mult-left-mono* [*OF le-maxI2*])
apply (*intro mult-nonneg-nonneg norm-ge-zero*)
done

lemma (**in** *bounded-bilinear*) *nonneg-bounded*:
  $\exists K{\geq}0.\ \forall a\ b.$ *norm* (*a* ** *b*) ≤ *norm a* * *norm b* * *K*
proof −
  from *pos-bounded*
  show *?thesis* **by** (*auto intro*: *order-less-imp-le*)
qed

lemma (**in** *bounded-bilinear*) *additive-right*: *additive* (λ*b*. *prod a b*)
by (*rule additive.intro*, *rule add-right*)

lemma (**in** *bounded-bilinear*) *additive-left*: *additive* (λ*a*. *prod a b*)
by (*rule additive.intro*, *rule add-left*)

lemma (**in** *bounded-bilinear*) *zero-left*: *prod 0 b* = *0*
by (*rule additive.zero* [*OF additive-left*])

lemma (**in** *bounded-bilinear*) *zero-right*: *prod a 0* = *0*
by (*rule additive.zero* [*OF additive-right*])

lemma (**in** *bounded-bilinear*) *minus-left*: *prod* (− *a*) *b* = − *prod a b*
by (*rule additive.minus* [*OF additive-left*])

lemma (**in** *bounded-bilinear*) *minus-right*: *prod a* (− *b*) = − *prod a b*
by (*rule additive.minus* [*OF additive-right*])

lemma (**in** *bounded-bilinear*) *diff-left*:
prod (*a* − *a'*) *b* = *prod a b* − *prod a' b*
by (*rule additive.diff* [*OF additive-left*])

lemma (**in** *bounded-bilinear*) *diff-right*:
prod a (*b* − *b'*) = *prod a b* − *prod a b'*
by (*rule additive.diff* [*OF additive-right*])

lemma (**in** *bounded-bilinear*) *bounded-linear-left*:
bounded-linear (λ*a*. *a* ** *b*)
apply (*unfold-locales*)
apply (*rule add-left*)
apply (*rule scaleR-left*)
apply (*cut-tac bounded*, *safe*)
apply (*rule-tac x*=*norm b* ∗ *K* **in** *exI*)
apply (*simp add*: *mult-ac*)
done

lemma (**in** *bounded-bilinear*) *bounded-linear-right*:
bounded-linear (λ*b*. *a* ** *b*)
apply (*unfold-locales*)
apply (*rule add-right*)
apply (*rule scaleR-right*)
apply (*cut-tac bounded*, *safe*)
apply (*rule-tac x*=*norm a* ∗ *K* **in** *exI*)
apply (*simp add*: *mult-ac*)
done

lemma (**in** *bounded-bilinear*) *prod-diff-prod*:
(*x* ** *y* − *a* ** *b*) = (*x* − *a*) ** (*y* − *b*) + (*x* − *a*) ** *b* + *a* ** (*y* − *b*)
by (*simp add*: *diff-left diff-right*)

interpretation *mult*:
bounded-bilinear [*op* ∗ :: *'a* ⇒ *'a* ⇒ *'a*::*real-normed-algebra*]
apply (*rule bounded-bilinear.intro*)
apply (*rule left-distrib*)
apply (*rule right-distrib*)
apply (*rule mult-scaleR-left*)
apply (*rule mult-scaleR-right*)
apply (*rule-tac x*=*1* **in** *exI*)
apply (*simp add*: *norm-mult-ineq*)
done

interpretation *mult-left*:
bounded-linear [(λ*x*::*'a*::*real-normed-algebra*. *x* ∗ *y*)]
by (*rule mult.bounded-linear-left*)

interpretation *mult-right*:
  bounded-linear [(λ*y*::′*a*::*real-normed-algebra*. *x* ∗ *y*)]
by (*rule mult.bounded-linear-right*)

interpretation *divide*:
  bounded-linear [(λ*x*::′*a*::*real-normed-field*. *x* / *y*)]
unfolding *divide-inverse* **by** (*rule mult.bounded-linear-left*)

interpretation *scaleR*: *bounded-bilinear* [*scaleR*]
apply (*rule bounded-bilinear.intro*)
apply (*rule scaleR-left-distrib*)
apply (*rule scaleR-right-distrib*)
apply *simp*
apply (*rule scaleR-left-commute*)
apply (*rule-tac x*=*1* **in** *exI*)
apply (*simp add*: *norm-scaleR*)
done

interpretation *scaleR-left*: *bounded-linear* [λ*r*. *scaleR r x*]
by (*rule scaleR.bounded-linear-left*)

interpretation *scaleR-right*: *bounded-linear* [λ*x*. *scaleR r x*]
by (*rule scaleR.bounded-linear-right*)

interpretation *of-real*: *bounded-linear* [λ*r*. *of-real r*]
unfolding *of-real-def* **by** (*rule scaleR.bounded-linear-left*)

end

theory *Real*
imports *ContNotDenum RealVector*
begin
end

16 Resource-Aware Operational semantics of Safe expressions

theory *SafeRASemanticsReal*
imports *../CoreSafe/SafeExpr ../SafeImp/SVMState ~~/src/HOL/Real/Real*
begin

types

$Delta = (Region \rightharpoonup real)$
$MinimumFreshCells = real$
$MinimumWords = real$
$Resources = Delta \times MinimumFreshCells \times MinimumWords$

types $FunDefEnv = string \rightharpoonup ProgVar\ list \times RegVar\ list \times unit\ Exp$
consts $\Sigma d :: FunDefEnv$

constdefs $sizeVal :: [HeapMap,\ Val] \Rightarrow int$
$sizeVal\ h\ v \equiv (case\ v\ of\ (Loc\ p) \Rightarrow int\ p \mid \text{-} \Rightarrow 0)$

constdefs $size :: [HeapMap, Location] \Rightarrow int$ **where**
$size\ h\ p \equiv (case\ h\ p\ of\ Some\ (j,C,vs) \Rightarrow (let\ rp = getRecursiveValuesCell\ (C,vs)\ in\ 1 + (\sum i \in rp.\ sizeVal\ h\ (vs!i))))$

constdefs $balanceCells :: Delta \Rightarrow real\ \ (\| \text{-} \| [71]\ 70)$
$balanceCells\ \delta \equiv (\sum n \in ran\ \delta.\ n)$

constdefs $addDelta :: Delta \Rightarrow Delta \Rightarrow Delta\ \ (\textbf{infix}\ \oplus\ 110)$
$addDelta\ \delta 1\ \delta 2 \equiv (\%x.\ (if\ x \in dom\ \delta 1 \cap dom\ \delta 2\ then\ (case\ \delta 1\ x\ of\ (Some\ y) \Rightarrow case\ \delta 2\ x\ of\ (Some\ z) \Rightarrow Some\ (y + z))\ else\ if\ x \in dom\ \delta 1 - dom\ \delta 2\ then\ \delta 1\ x\ else\ if\ x \in dom\ \delta 2 - dom\ \delta 1\ then\ \delta 2\ x\ else\ None))$

constdefs $emptyDelta :: nat \Rightarrow nat \rightharpoonup real\ \ (\ []\text{-}\ [71]\ 70)$
$[]_k \equiv (\%i.\ if\ i \in \{0..k\}\ then\ Some\ 0\ else\ None)$

consts $def\text{-}copy :: nat \Rightarrow Heap \Rightarrow bool$

inductive
$SafeRASem :: [Environment, HeapMap, nat, real, unit\ Exp, HeapMap, nat, Val, Resources] \Rightarrow bool$
$(\ \text{-} \vdash \text{-}, \text{-}, \text{-}, \text{-} \Downarrow \text{-}, \text{-}, \text{-}, \text{-}\ \ [71,71,71,71,71,71,71,71]\ 70)$
where

$litInt:\ \ E \vdash h,\ k,\ td,\ (ConstE\ (LitN\ i)\ a) \Downarrow h,\ k,\ IntT\ \ i,([]_k,0,1)$

| *litBool*: $E \vdash h\ ,\ k,\ td,\ (ConstE\ (LitB\ b)\ a) \Downarrow h,\ k,\ BoolT\ b,([]_k,0,1)$

| *var1* : $E1\ x = Some\ (Val.Loc\ p)$
$\Longrightarrow (E1,E2) \vdash h,\ k,\ td,\ (VarE\ x\ a) \Downarrow h,\ k,\ Val.Loc\ p,([]_k,0,1)$

| *var2* : $[\![E1\ x = Some\ (Val.Loc\ p);\ E2\ r = Some\ j;\ j <= k;$
$copy\ (h,k)\ p\ j = ((h',k),p');\ def\text{-}copy\ p\ (h,k);$
$m = real\ (size\ h\ p)]\!]$
$\Longrightarrow (E1,E2) \vdash h,\ k\ ,\ td,\ (x\ @\ r\ a) \Downarrow h',\ k,\ Val.Loc\ p',([j \mapsto m],m,2)$

| *var3* : $[\![E1\ x = Some\ (Val.Loc\ p);\ h\ p = Some\ c;\ SafeHeap.fresh\ q\ h\ ]\!]$
$\Longrightarrow (E1,E2) \vdash h,k,td,(ReuseE\ x\ a) \Downarrow ((h(p{:=}None))(q \mapsto c)),k,Val.Loc\ q,([]_k,0,1)$

| *let1* : $[\![\ \forall\ C\ as\ r\ a'.\ e1 \neq ConstrE\ C\ as\ r\ a';\ x1 \notin dom\ E1;$
$(E1,E2) \vdash h,k,0,e1 \Downarrow h',k,v1,(\delta 1,m1,s1);$
$(E1(x1 \mapsto v1),E2) \vdash h',k,(td{+}1),e2 \Downarrow h'',k,v2,(\delta 2,m2,s2)\ ]\!]$
$\Longrightarrow (E1,E2) \vdash h,k,\ td,\ \ Let\ x1 = e1\ In\ e2\ a \Downarrow h'',k,v2,$
$(\delta 1 \oplus \delta 2,max\ m1\ (m2\ +\ \|\delta 1\|),max\ (s1{+}2)\ (s2{+}1))$

| *let2* : $[\![\ E2\ r = Some\ j;\ j \leq k;\ fresh\ p\ h;\ x1 \notin dom\ E1;\ r \neq self;$
$(E1(x1 \mapsto Val.Loc\ p),E2) \vdash$
$h(p \mapsto (j,(C,map\ (atom2val\ E1)\ as))),k,\ (td{+}1),e2 \Downarrow h',k,v2,(\delta,m,s)]\!]$
$\Longrightarrow (E1,E2) \vdash h,k,\ td,\ Let\ x1 = ConstrE\ C\ as\ r\ a'\ In\ e2\ a \Downarrow h',k,v2,$
$(\delta \oplus (empty(j \mapsto 1)),m{+}1,s{+}1)$

| *case1*: $[\![\ \ i < length\ alts;$
$E1\ x\ = Some\ (Val.Loc\ p);$
$h\ p\ = Some\ (j,C,vs)\ ;$
$alts!i = (pati,\ ei);$
$pati = ConstrP\ C\ ps\ ms;$
$xs = (snd\ (extractP\ (fst\ (alts\ !\ i))));$
$E1' = extend\ E1\ xs\ vs;$
$def\text{-}extend\ E1\ xs\ vs;$
$nr = real\ (length\ vs);$
$(E1',E2) \vdash h,k,\ (td{+}\ nr),\ ei \Downarrow h',k,v,(\delta,m,s)]\!]$
$\Longrightarrow (E1,E2) \vdash h,k,\ td,\ Case\ (VarE\ x\ a)\ Of\ alts\ a' \Downarrow h',\ k,v,(\delta,m,\ (s{+}nr))$

| *case1-1*: $[\![\ i < length\ alts;$
$E1\ x\ = Some\ (IntT\ n);$
$alts!i = (pati,\ ei);$
$pati = ConstP\ (LitN\ n);$
$(E1,E2) \vdash h,k,\ td,\ ei \Downarrow h',k,v,(\delta,m,s)]\!]$
$\Longrightarrow (E1,E2) \vdash h,k,\ td,\ Case\ (VarE\ x\ a)\ Of\ alts\ a' \Downarrow h',\ k,v,(\delta,m,s)$

| *case1-2*: $[\![\ i < length\ alts;$
$E1\ x\ = Some\ (BoolT\ b);$
$alts!i = (pati,\ ei);$
$pati = ConstP\ (LitB\ b);$

$(E1,E2) \vdash h,k,\ td,\ ei \Downarrow h',k,v,(\delta,m,s)]\!]$
$\Longrightarrow (E1,E2) \vdash h,k,\ td,\ Case\ (VarE\ x\ a)\ Of\ alts\ a' \Downarrow h',\ k,v,(\delta,m,s)$

| *case2*: $[\![$ $i < length\ alts$;
$E1\ x\ = Some\ (Val.Loc\ p)$;
$h\ p\ = Some\ (j,C,vs)$;
$alts!i = (pati,\ ei)$;
$pati = ConstrP\ C\ ps\ ms$;
$xs = (snd\ (extractP\ (fst\ (alts\ !\ i))))$;
$E1' = extend\ E1\ xs\ vs$;
def-extend E1 xs vs;
$nr = real\ (length\ vs)$;
$j <= k$;
$(E1',E2) \vdash h(p :=None),k,(td\ +\ nr),\ ei \Downarrow h',k,v,(\delta,m,s)]\!]$
$\Longrightarrow (E1,E2) \vdash h,k,\ td,\ CaseD\ (VarE\ x\ a)\ Of\ alts\ a' \Downarrow h',\ k,v,$
$(\delta\oplus(empty(j\mapsto -1)),max\ 0\ (m\ -\ 1),s+nr)$

| *app-primops*: $[\![$ $primops\ f\ = Some\ oper$;
$v1\ = atom2val\ E1\ a1$;
$v2\ = atom2val\ E1\ a2$;
$v\ = execOp\ oper\ v1\ v2]\!]$
$\Longrightarrow (E1,E2) \vdash h,k,\ td,\ AppE\ f\ [a1,a2]\ []\ a \Downarrow\ h,\ k,v,([]_k,0,2)$

| *app*: $[\![$ $\Sigma d\ f\ = Some\ (xs,rs,e)$; $primops\ f\ = None$;
$distinct\ xs$; $distinct\ rs$; $dom\ E1\ \cap\ set\ xs\ = \{\}$;
$length\ xs\ = length\ as$; $length\ rs\ = length\ rr$;
$E1' = map\text{-}of\ (zip\ xs\ (map\ (atom2val\ E1)\ as))$;
$n\ = real\ (length\ xs)$;
$l\ = real\ (length\ rs)$;
$E2' = (map\text{-}of\ (zip\ rs\ \ (map\ (the\circ E2)\ rr)))\ (self\ \mapsto\ Suc\ k)$;
$(E1',E2')\ \vdash h,\ (Suc\ k),\ (n+l),\ e\Downarrow h',\ (Suc\ k),v,(\delta,m,s)$;
$h''=\ h'\ |`\ \{p.\ p \in dom\ h'\ \&\ fst\ (the\ (h'\ p)) \leq k\}\ ]\!]$
$\Longrightarrow (E1,E2) \vdash\ h,k,\ td,\ AppE\ f\ as\ rr\ a \Downarrow h'',\ k,v,$
$(\delta(k+1:=None),m,max(n\ +l)\ (s+n+l-td))$

end

17 Depth-Aware Operational semantics of Safe expressions

theory *SafeDepthSemanticsReal*
imports *SafeRASemanticsReal ../SafeImp/SVMState*
begin

inductive

SafeDepthSem :: [*Environment*, *HeapMap*, *nat*, *real*, *unit Exp*, *string*, *HeapMap*, *nat*,
Val, *Resources*, *nat*] ⇒ *bool*
(- ⊢ - , - , - , - ⇓ - - , - , - , - , - [71,71,71,71,71,71,71,71,71,71,71] 70)
where

litInt : $E \vdash h,\ k,\ td,\ (ConstE\ (LitN\ i)\ a) \Downarrow f\ h,\ k,\ IntT\ \ i,\ ([]_k,0,1),\ 0$

| *litBool*: $E \vdash h\ ,\ k,\ td,(ConstE\ (LitB\ b)\ a) \Downarrow f\ h,\ k,\ BoolT\ b,\ ([]_k,0,1),\ 0$

| *var1* : $E1\ x = Some\ (Val.Loc\ p)$
$\Longrightarrow (E1,E2) \vdash h,\ k,\ td,\ (VarE\ x\ a) \Downarrow f\ h,\ k,\ Val.Loc\ p,\ ([]_k,0,1),\ 0$

| *var2* : $[\![E1\ x = Some\ (Val.Loc\ p);\ E2\ r = Some\ j;\ \ j <= k;$
$copy\ (h,k)\ p\ j = ((h',k),p');\ def\text{-}copy\ p\ (h,k);$
$m = real\ (size\ h\ p)]\!]$
$\Longrightarrow (E1,E2) \vdash h,\ k\ ,\ td,\ (x\ @\ r\ a) \Downarrow f\ h',\ k,\ Val.Loc\ p',\ ([j \mapsto m],m,2),\ 0$

| *var3* : $[\![E1\ x = Some\ (Val.Loc\ p);\ h\ p = Some\ c;\ SafeHeap.fresh\ q\ h]\!]$
$\Longrightarrow (E1,E2) \vdash h,k,\ td,\ (ReuseE\ x\ a) \Downarrow f$
$((h(p{:=}None))(q \mapsto c)),k,Val.Loc\ q,([]_k,0,1),0$

| *let1* : $[\![\ \forall\ C\ as\ r\ a'.\ e1 \neq ConstrE\ C\ as\ r\ a';\ x1 \notin dom\ E1;$
$(E1,E2) \vdash h,\ k,\ 0,\ e1 \Downarrow f\ h',\ k,\ v1,\ (\delta 1,m1,s1),\ n1\ \wedge$
$(E1(x1 \mapsto v1),E2) \vdash h',\ k,\ (td+1),\ e2 \Downarrow f\ h'',\ k,\ v2,\ (\delta 2,m2,s2),\ n2]\!]$
$\Longrightarrow (E1,E2) \vdash h,\ k,\ td,\ Let\ x1 = e1\ In\ e2\ a \Downarrow f\ h'',\ k,\ v2,\ (\delta 1 \oplus \delta 2,max\ m1$
$(m2 + \|\delta 1\|),max\ (s1+2)\ (s2+1)),\ max\ n1\ n2$

| *let2* : $[\![\ E2\ r = Some\ j;\ j \leq k;\ fresh\ p\ h;\ x1 \notin dom\ E1;\ r \neq self;$
$(E1(x1 \mapsto Val.Loc\ p),E2) \vdash$
$h(p \mapsto (j,(C,map\ (atom2val\ E1)\ as))),\ k,\ (td+1),\ e2 \Downarrow f\ h',\ k,\ v2,$
$(\delta,m,s),\ n]\!]$
$\Longrightarrow (E1,E2) \vdash h,k,td,\ Let\ x1 = ConstrE\ C\ as\ r\ a'\ In\ e2\ a \Downarrow f\ h',k,v2,(\delta \oplus (empty(j \mapsto 1)),m+1,s+1),n$

| *case1*: $[\![\ \ i < length\ alts;$
$E1\ x\ = Some\ (Val.Loc\ p);$
$h\ p\ = Some\ (j,C,vs);$
$alts!i = (pati,\ ei);$
$pati = ConstrP\ C\ ps\ ms;$
$xs = (snd\ (extractP\ (fst\ (alts\ !\ i))));$
$E1' = extend\ E1\ xs\ vs;$
$def\text{-}extend\ E1\ xs\ vs;$
$nr = real\ (length\ vs);$
$(E1',E2) \vdash h,\ k,\ (td\ +\ nr),\ ei \Downarrow f\ h',\ k,\ v,\ (\delta,m,s),\ n]\!]$
$\Longrightarrow (E1,E2) \vdash h,k,\ td,\ Case\ (VarE\ x\ a)\ Of\ alts\ a' \Downarrow f\ h',\ k,\ v,\ (\delta,m,$
$(s+nr)),n$

| *case1-1*: $[\![\ \ i < length\ alts;$
$E1\ x\ = Some\ (IntT\ n);$

$alts!i = (pati,\ ei)$;
$pati = ConstP\ (LitN\ n)$;
$(E1,E2) \vdash h,\ k,\ td,\ ei \Downarrow f\ h',\ k,\ v,\ (\delta,m,s),\ nf\rrbracket$
$\Longrightarrow (E1,E2) \vdash h,\ k,\ td,\ Case\ (VarE\ x\ a)\ Of\ alts\ a' \Downarrow f\ h',\ k,\ v,\ (\delta,m,s),nf$

| *case1-2*: ⟦ $i < length\ alts$;
$E1\ x\ = Some\ (BoolT\ b)$;
$alts!i = (pati,\ ei)$;
$pati = ConstP\ (LitB\ b)$;
$(E1,E2) \vdash h,\ k,\ td,\ ei \Downarrow f\ h',\ k,\ v,(\delta,m,s),n\rrbracket$
$\Longrightarrow (E1,E2) \vdash h,\ k,\ td,\ Case\ (VarE\ x\ a)\ Of\ alts\ a' \Downarrow f\ h',\ k,\ v,\ (\delta,m,s),$
n

| *case2*: ⟦ $i < length\ alts$;
$E1\ x\ = Some\ (Val.Loc\ p)$;
$h\ p\ = Some\ (j,C,vs)$;
$alts!i = (pati,\ ei)$;
$pati = ConstrP\ C\ ps\ ms$;
$xs = (snd\ (extractP\ (fst\ (alts\ !\ i))))$;
$E1' = extend\ E1\ xs\ vs$;
$def\text{-}extend\ E1\ xs\ vs$;
$nr = real\ (length\ vs)$;
$j <= k$;
$(E1',E2) \vdash h(p := None),\ k,(td\ +\ nr),\ ei \Downarrow f\ h',\ k,\ v,\ (\delta,m,s),\ n\rrbracket$
$\Longrightarrow (E1,E2) \vdash h,\ k,\ td,\ CaseD\ (VarE\ x\ a)\ Of\ alts\ a' \Downarrow f\ h',\ k,\ v,$
$(\delta \oplus (empty(j \mapsto -1)),max\ 0\ (m\ -\ 1),s+nr),\ n$

| *app-primops*: ⟦ $primops\ g = Some\ oper$;
$v1 = atom2val\ E1\ a1$;
$v2 = atom2val\ E1\ a2$;
$v = execOp\ oper\ v1\ v2\rrbracket$
$\Longrightarrow (E1,E2) \vdash h,\ k,\ td,\ AppE\ g\ [a1,a2]\ []\ a \Downarrow f\ h,\ k,\ v,\ ([]_k,0,2),\ 1$

| *app*: ⟦ $\Sigma d\ f = Some\ (xs,rs,e)$; $primops\ f = None$;
$distinct\ xs$; $distinct\ rs$; $dom\ E1 \cap set\ xs = \{\}$;
$length\ xs = length\ as$; $length\ rs = length\ rr$;
$E1' = map\text{-}of\ (zip\ xs\ (map\ (atom2val\ E1)\ as))$;
$n = real\ (length\ xs)$;
$l = real\ (length\ rs)$;
$E2' = (map\text{-}of\ (zip\ rs\ \ (map\ (the \circ E2)\ rr)))\ (self \mapsto Suc\ k)$;
$(E1',E2') \vdash h,\ (Suc\ k),\ (n+l),\ e \Downarrow f\ h',\ (Suc\ k),\ v,\ (\delta,m,s),\ nf$;
$h'' =\ h'\ |`\ \{p.\ p \in dom\ h'\ \&\ fst\ (the\ (h'\ p)) \leq k\}\rrbracket$
$\Longrightarrow (E1,E2) \vdash h,\ k,\ td,\ (AppE\ f\ as\ rr\ a) \Downarrow f\ h'',\ k,\ v,\ (\delta(k+1:=None),m,max$
$(n\ +l)\ (s+n+l-td)),\ (nf+1)$

| *app2*: ⟦ $\Sigma d\ g = Some\ (xs,rs,e)$; $primops\ g = None$; $f \neq g$;
$distinct\ xs$; $distinct\ rs$; $dom\ E1 \cap set\ xs = \{\}$;
$length\ xs = length\ as$; $length\ rs = length\ rr$;
$E1' = map\text{-}of\ (zip\ xs\ (map\ (atom2val\ E1)\ as))$;

$n = real\ (length\ xs)$;
$l = real\ (length\ rs)$;
$E2' = (map\text{-}of\ (zip\ rs\ \ (map\ (the \circ E2)\ rr)))\ (self \mapsto Suc\ k)$;
$(E1',E2') \vdash h, (Suc\ k), (n+l), e \Downarrow f\ h', (Suc\ k), v, (\delta,m,s), nf$;
$h'' = h' \mid` \{p.\ p \in dom\ h' \ \&\ fst\ (the\ (h'\ p)) \leq k\}\rrbracket$
$\Longrightarrow (E1,E2) \vdash h, k, td, AppE\ g\ as\ rr\ a \Downarrow f\ h'', k, v, (\delta(k+1:=None),m,max\ (n\ +l)\ (s+n+l-td)), nf$

lemma *eqSemRADepth*[*rule-format*]: $(E1,E2) \vdash h,k,td,e \Downarrow h',k,v,r \longrightarrow (\exists\ n.\ (E1,E2) \vdash h,k,td,e \Downarrow f\ h',k,v,r,n)$
apply (*rule impI*)
apply (*erule SafeRASem.induct*)

apply (*rule-tac x=0* **in** *exI*)
apply (*rule SafeDepthSem.litInt*)

apply (*rule-tac x=0* **in** *exI*)
apply (*rule SafeDepthSem.litBool*)

apply (*rule-tac x=0* **in** *exI*)
apply (*rule SafeDepthSem.var1*,*assumption*)

apply (*rule-tac x=0* **in** *exI*)
apply (*rule SafeDepthSem.var2*,*simp*,*simp*,*assumption*,*assumption*,*assumption*,*assumption*)

apply (*rule-tac x=0* **in** *exI*)
apply (*rule SafeDepthSem.var3*,*assumption*+)

apply (*erule exE*)+
apply (*rename-tac n1 n2*)
apply (*rule-tac x=max n1 n2* **in** *exI*)
apply (*rule SafeDepthSem.let1*)
apply (*assumption*+)
apply (*rule conjI*,*simp*,*simp*)

apply (*erule exE*)+
apply (*rename-tac n2*)
apply (*rule-tac x=n2* **in** *exI*)
apply (*rule SafeDepthSem.let2*)
apply (*assumption*+)

apply (*elim exE*)
apply (*rename-tac ni*)
apply (*rule-tac x=ni* **in** *exI*)
apply (*rule SafeDepthSem.case1*)
apply (*assumption+*)

apply (*elim exE*)
apply (*rename-tac ni*)
apply (*rule-tac x=ni* **in** *exI*)
apply (*rule SafeDepthSem.case1-1*)
apply (*assumption+*)

apply (*elim exE*)
apply (*rename-tac ni*)
apply (*rule-tac x=ni* **in** *exI*)
apply (*rule SafeDepthSem.case1-2*)
apply (*assumption+*)

apply (*elim exE*)
apply (*rename-tac ni*)
apply (*rule-tac x=ni* **in** *exI*)
apply (*rule SafeDepthSem.case2*)
apply (*assumption+*)

apply (*rule-tac x=1* **in** *exI*)
apply (*rule SafeDepthSem.app-primops*)
apply *assumption+*

apply (*elim exE*)
apply (*rename-tac nf*)
apply (*case-tac fa=f*)

apply (*rule-tac x=nf+1* **in** *exI*)
apply (*rule-tac s=f* **and** *t=fa* **in** *subst*)
apply *simp*
apply (*rule-tac s=*h' |‘ $\{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\}$ **and** *t=*h'' **in** *subst*)
apply *simp*
apply *clarify*
apply (*rule-tac* h'*=*h' **in** *SafeDepthSem.app*)
apply *assumption+*
apply *simp*

apply *simp*
apply *simp*
apply *simp*
apply *assumption*
apply *simp*

apply (*rule-tac x=nf* **in** *exI*)
apply (*rule-tac s=h′* |‘ {*p* ∈ *dom h′*. *fst* (*the* (*h′ p*)) ≤ *k*} **and** *t=h″* **in** *subst*)
apply *simp*
apply (*rule SafeDepthSem.app2*)
apply *assumption+*
apply *simp*
apply *assumption+*
apply *simp*
done

lemma *eqSemDepthRA*[*rule-format*]: (*E1*,*E2*) ⊢ *h*,*k*,*td*,*e*⇓*f h′*,*k*,*v*,*r*,*n* ⟶
(*E1*,*E2*) ⊢ *h*,*k*,*td*,*e*⇓*h′*,*k*,*v*,*r*

apply (*rule impI*)
apply (*erule SafeDepthSem.induct*)

apply (*rule SafeRASem.litInt*)

apply (*rule SafeRASem.litBool*)

apply (*rule SafeRASem.var1*,*assumption*)

apply (*rule SafeRASem.var2*,*assumption+*)

apply (*rule SafeRASem.var3*,*assumption+*)

apply (*elim conjE*)
apply (*rule SafeRASem.let1*,*assumption+*)

apply (*rule SafeRASem.let2*,*assumption+*)

apply (*rule SafeRASem.case1*,*assumption+*)

apply (*rule SafeRASem.case1-1*,*assumption*+)

apply (*rule SafeRASem.case1-2*,*assumption*+)

apply (*rule SafeRASem.case2*,*assumption*+)

apply (*rule SafeRASem.app-primops*,*assumption*+)

apply (*rule SafeRASem.app*,*assumption*+)

apply (*rule SafeRASem.app*,*assumption*+)
done

constdefs
SafeBoundSem :: [*Environment*, *HeapMap*, *nat*, *real*, *unit Exp*, *string* × *nat*, *HeapMap*, *nat*, *Val*, *Resources*] ⇒ *bool*
(- ⊢ - , - , - , - ⇓ - - , - , - , - [*71*,*71*,*71*,*71*,*71*,*71*,*71*,*71*,*71*] *70*)

E ⊢ *h*, *k*, *td*, *e* ⇓*tup h′*, *k′*, *v*, *r* ≡
(*let* (*f*,*n*) = *tup in k*=*k′* ∧ (∃ *nf* . *E* ⊢ *h*, *k*, *td*, *e* ⇓*f h′*, *k*, *v*, *r*, *nf* ∧ *nf* ≤ *n*))

lemma *eqSemRABound* [*rule-format*]:
(*E1*,*E2*) ⊢ *h*,*k*,*td*,*e*⇓*h′*,*k*,*v*,*r* ≡
(∃ *n*.(*E1*,*E2*) ⊢ *h*,*k*,*td*,*e*⇓(*f*,*n*) *h′*,*k*,*v*,*r*)

apply (*rule eq-reflection*)
apply (*rule iffI*)

apply (*simp add*: *SafeBoundSem-def*)
apply (*drule-tac ?f*=*f* **in** *eqSemRADepth*)
apply (*elim exE*)
apply (*rule-tac x*=*x* **in** *exI*)
apply (*rule-tac x*=*x* **in** *exI*)
apply (*rule conjI*,*assumption*,*simp*)

apply (*simp add*: *SafeBoundSem-def del*:*Product-Type.split-paired-Ex*)
apply (*elim exE*)
apply (*elim conjE*)
apply (*drule eqSemDepthRA*)
by *assumption*

lemma *eqSemBoundRA* [*rule-format*]:
 $\exists\ n.\ (E1,E2) \vdash h,k,td,e\Downarrow(f,n)\ h',k,v,r \equiv$
  $(E1,E2) \vdash h,k,td,e\Downarrow h',k,v,r$
apply (*rule eq-reflection*)
apply (*rule iffI*)

apply (*simp add*: *SafeBoundSem-def del*:*Product-Type.split-paired-Ex*)
apply (*elim exE*)
apply (*elim conjE*)
apply (*drule eqSemDepthRA*)
apply *assumption*

apply (*simp add*: *SafeBoundSem-def*)
apply (*drule-tac ?f=f* **in** *eqSemRADepth*)
apply (*elim exE*)
apply (*rule-tac x=x* **in** *exI*)
apply (*rule-tac x=x* **in** *exI*)
apply (*rule conjI*,*assumption*,*simp*)
done

lemma *impSemBoundRA* [*rule-format*]:
 $(E1,E2) \vdash h,k,td,e\Downarrow(f,n)\ h',k,v,r \Longrightarrow$
  $(E1,E2) \vdash h,k,td,e\Downarrow h',k,v,r$
apply (*simp add*: *SafeBoundSem-def del*:*Product-Type.split-paired-Ex*)
apply (*elim exE*,*elim conjE*)
apply (*drule eqSemDepthRA*)
by *assumption*

end

18 Definitions for certifying memory bounds

theory *Costes-definitions*
imports *SafeRASemanticsReal SafeDepthSemanticsReal*
 ~~/*src*/*HOL*/*Real*/*Real*
begin

types *RegionTypeVariable* = *string*

constdefs

$\varrho self :: RegionTypeVariable$
$\varrho self \equiv ''rho\text{-}self''$

constdefs $\varrho self\text{-}f :: FunName \Rightarrow RegionTypeVariable$
$\varrho self\text{-}f\ f \equiv \varrho self\ @\ ''\text{-}''\ @\ f$

types $Cost = real\ list \rightharpoonup real$

types $AbstractDelta = RegionTypeVariable \rightharpoonup Cost$
$AbstractMu = Cost$
$AbstractSigma = Cost$

types $VarType = string$

datatype $TypeExpression = VarT\ VarType$
$\mid ConstrT\ string\ TypeExpression\ list\ VarType\ list$

types $FunctionResourcesSignature = FunName \rightharpoonup AbstractDelta \times AbstractMu \times AbstractSigma$
$FunctionTypesSignature = FunName \rightharpoonup (TypeExpression\ list \times VarType\ list \times TypeExpression)$
$FunctionDefinitionSignature = FunName \rightharpoonup (ProgVar\ list \times RegVar\ list \times unit\ Exp)$

types $RegionEnv = string \rightharpoonup TypeExpression\ list \times VarType\ list \times TypeExpression$
consts $\Sigma t :: FunctionTypesSignature$

types $ConstructorSignatureFun = string \rightharpoonup TypeExpression\ list \times VarType \times TypeExpression$
consts $constructorSignature :: ConstructorSignatureFun$

types $PhiMapping = (string \rightharpoonup Cost)$

types $TypeMapping = (string \rightharpoonup TypeExpression)$
types $RegMapping = (string \rightharpoonup string)$
types $ThetaMapping = TypeMapping \times RegMapping$

types $InstantiationMapping = VarType \rightharpoonup nat$

types *TypeMu* = (*string* ⇀ *TypeExpression*) × (*string* ⇀ *string*)

consts *mu-ext* :: *TypeMu* ⇒ *TypeExpression* ⇒ *TypeExpression*
mu-exts :: *TypeMu* ⇒ *TypeExpression list* ⇒ *TypeExpression list*
primrec
mu-ext μ (*VarT* a) = *the* ((*fst* μ) a)
mu-ext μ (*ConstrT* T tm ϱs) = (*ConstrT* T (*mu-exts* μ tm) (*map* (*the* ∘ (*snd* μ)) ϱs))

mu-exts μ [] = []
mu-exts μ (x#xs) = *mu-ext* μ x # (*mu-exts* μ xs)

constdefs
RecPos :: *Val list* ⇒ *TypeExpression list* ⇒ *TypeExpression* ⇒ *Val list*
RecPos vn tn t == *map fst* (*filter* (λ(*vi*,*ti*). *ti*=*t*) (*zip vn tn*))

function *size* :: *Val* ⇒ *HeapMap* ⇒ *real*
where
size (*IntT i*) *h* = *0*
| *size* (*BoolT b*) *h* = *0*
| *size* (*Loc p*) *h* = (*case* (*h p*) *of Some* (*j*,*C*,*vn*) ⇒
(*case* Σ*t C of Some* (*ti*,ϱs,*t*)
⇒ *1* + (∑ *v* ∈ *set* (*RecPos vn ti t*). *size v h*)
| - ⇒ *0*)
| - ⇒ *0*)
by *pat-completeness auto*

constdefs *sizeEnv* :: (*ProgVar* ⇀ *Val*) ⇒ *ProgVar* ⇒ *HeapMap* ⇒ *real*
sizeEnv E x h ≡ (*case E x of Some v* ⇒ *size v h*
| - ⇒ *0*)

constdefs
sizeAbstractDelta-si :: *AbstractDelta* ⇒ *real list* ⇒ *nat* ⇒ *InstantiationMapping*
⇒ *real*
sizeAbstractDelta-si Δ *si j* η ≡ ∑ ϱ∈{ϱ. η ϱ = *Some j*}. *the* (*the* (Δ ϱ) *si*)

constdefs
Delta-ge :: *AbstractDelta* ⇒ *real list* ⇒ *nat* ⇒ *InstantiationMapping*
⇒ *Delta* ⇒ *bool*
Delta-ge Δ *si k* η δ ≡ (∀ *j* ∈ {*0*..*k*}. *sizeAbstractDelta-si* Δ *si j* η ≥ *the* (δ *j*))

constdefs
mu-ge :: *AbstractMu* ⇒ *real list* ⇒ *MinimumFreshCells* ⇒ *bool*
mu-ge μ *si m* ≡ *the* (μ *si*) ≥ *m*

constdefs
sigma-ge :: *AbstractSigma* ⇒ *real list* ⇒ *MinimumWords* ⇒ *bool*
sigma-ge σ *si s* == *the* (σ *si*) ≥ *s*

fun *build-si* :: (*ProgVar* ⇀ *Val*) ⇒ *HeapMap* ⇒ *ProgVar list* ⇒ *real list*
where
build-si E h [] = []
| *build-si E h* (*x*#*xs*) = (*sizeEnv E x h*) # *build-si E h xs*

constdefs *defined* :: *AbstractDelta* ⇒ *AbstractMu* ⇒ *AbstractSigma* ⇒ *real list* ⇒ *bool*
defined Δ μ σ *si* ≡
(∀ ϱ ∈ *dom* Δ. (∀ *si'*. *length si* = *length si'* ⟶ *si'* ∈ *dom* (*the* (Δ ϱ))))
∧ (∀ *si'*. *length si* = *length si'* ⟶ *si'* ∈ *dom* μ)
∧ (∀ *si'*. *length si* = *length si'* ⟶ *si'* ∈ *dom* σ)

types *FunTypesEnv* = *string* ⇀ *ThetaMapping*
FunSizesEnv = *string* ⇀ *PhiMapping*

consts Σϑ :: *FunTypesEnv*
consts ΣΦ :: *FunSizesEnv*

constdefs *typesVarsAPP* :: *FunTypesEnv* ⇒ *string* ⇒ *TypeMapping*
typesVarsAPP Σ *f* ≡ (*case* Σ *f of Some* (ϑ*1*,ϑ*2*) ⇒ ϑ*1*)

constdefs *typesRegsAPP* :: *FunTypesEnv* ⇒ *string* ⇒ *RegMapping*
typesRegsAPP Σ *f* ≡ (*case* Σ *f of Some* (ϑ*1*,ϑ*2*) ⇒ ϑ*2*)

constdefs *typesAPP* :: *FunTypesEnv* ⇒ *string* ⇒ *ThetaMapping*
typesAPP Σ *f* ≡ (*typesVarsAPP* Σ *f*,*typesRegsAPP* Σ *f*)

constdefs *typesArgAPP* :: *RegionEnv* ⇒ *string* ⇒ *TypeExpression list*
typesArgAPP Σ *f* == (*case* Σ *f of Some* (*ti*,ϱ*s*,*tf*) ⇒ *ti*)

constdefs *regionsArgAPP* :: *RegionEnv* ⇒ *string* ⇒ *string list*
regionsArgAPP Σ *f* ≡ (*case* Σ *f of Some* (*ti*,ϱ*s*,*tf*) ⇒ ϱ*s*)

constdefs *typeResAPP* :: *RegionEnv* ⇒ *string* ⇒ *TypeExpression*
typeResAPP Σ *f* ≡ (*case* Σ *f of Some* (*ti*,ϱ*s*,*tf*) ⇒ *tf*)

constdefs *sizesAPP* :: *FunSizesEnv* ⇒ *string* ⇒ *PhiMapping*
sizesAPP Σ *f* ≡ (*case* Σ *f of Some* φ ⇒ φ)

constdefs *R-Args* :: *RegionEnv* ⇒ *FunName* ⇒ *string list*
R-Args Σ *f* ≡ (*case* Σ *f of Some* (*ts*,ϱ*s*,*t*) ⇒ ϱ*s*)

constdefs
admissible :: *FunName* ⇒ *InstantiationMapping* ⇒ *nat* ⇒ *bool*
admissible f η *k* ≡
ϱ*self-f f* ∈ *dom* η ∧
(∀ ϱ ∈ *dom* η.
∃ *k'*.
η ϱ = *Some k'* ∧
(ϱ = ϱ*self-f f* ⟶ *k'* = *k*) ∧
(ϱ ≠ ϱ*self-f f* ⟶ *k'* < *k*))

consts *regions* :: *TypeExpression* ⇒ *string set*
regions' :: *TypeExpression list* ⇒ *string set*

primrec
regions (*VarT a*) = {}
regions (*ConstrT T tm* ϱ*s*) = (*regions' tm*) ∪ *set* ϱ*s*

regions' [] = {}
regions' (*t*#*ts*) = *regions t* ∪ *regions' ts*

consts *variables* :: *TypeExpression* ⇒ *string set*
variables' :: *TypeExpression list* ⇒ *string set*

primrec
variables (*VarT a*) = {*a*}
variables (*ConstrT T tm* ϱ*s*) = (*variables' tm*)

variables' [] = {}
variables' (*t*#*ts*) = *variables t* ∪ *variables' ts*

fun
wellT :: *TypeExpression list* ⇒ *VarType* ⇒ *TypeExpression* ⇒ *bool*
where
wellT tn ϱ (*ConstrT T tm ϱs*) =
((*length ϱs* > 0 ∧ *ϱ* = *last ϱs* ∧ *distinct ϱs* ∧ *last ϱs* ∉ *regions*′ *tm*)
∧ (∀ *i* < *length tn*. *regions* (*tn*!*i*) ⊆ *regions* (*ConstrT T tm ϱs*) ∧
variables (*tn*!*i*) ⊆ *variables* (*ConstrT T tm ϱs*)))

inductive
consistent-v :: [*TypeExpression*, *InstantiationMapping*, *Val*, *HeapMap*] ⇒ *bool*
where
primitiveI : *consistent-v* (*ConstrT intType* [] []) *η* (*IntT i*) *h*
| *primitiveB* : *consistent-v* (*ConstrT boolType* [] []) *η* (*BoolT b*) *h*
| *variable* : *consistent-v* (*VarT a*) *η v h*
| *algebraic-None* : *p* ∉ *dom h* ⟹ *consistent-v t η* (*Loc p*) *h*
| *algebraic* : ⟦ *h p* = *Some* (*j*,*C*,*vn*);
ϱl = *last ϱs*;
ϱl ∈ *dom η*; *η* (*ϱl*) = *Some j*;
constructorSignature C = *Some* (*tn*′,*ϱ*′,*ConstrT T tm*′ *ϱs*′);
wellT tn′ (*last ϱs*′) (*TypeExpression.ConstrT T tm*′ *ϱs*′);
length vn = *length tn*′;
∃ *μ1 μ2*. (((*the* (*μ2* (*last ϱs*′))), *mu-ext* (*μ1*,*μ2*) (*ConstrT T tm*′ *ϱs*′)) =
(*ϱl*,*ConstrT T tm ϱs*) ∧
(∀ *i* < *length vn*. *consistent-v* ((*map* (*mu-ext* (*μ1*,*μ2*)) *tn*′)!*i*) *η* (*vn*!*i*) *h*)) ⟧
⟹ *consistent-v* (*ConstrT T tm ϱs*) *η* (*Loc p*) *h*

fun
consistent :: *ThetaMapping* ⇒ *InstantiationMapping* ⇒ *Environment* ⇒ *HeapMap* ⇒ *bool*
where
consistent (*ϑ1*,*ϑ2*) *η* (*E1*,*E2*) *h* =
((∀ *x* ∈ *dom E1*. ∃ *t v*. *ϑ1 x* = *Some t*
∧ *E1 x* = *Some v*
∧ *consistent-v t η v h*)
∧ (∀ *r* ∈ *dom E2* . ∃ *r*′ *r*″. *ϑ2 r* = *Some r*′
∧ *η r*′ = *Some r*″
∧ *E2 r* = *Some r*″)
∧ *self* ∈ *dom E2*
∧ *ϑ2 self* = *Some ϱself*)

fun *valid-f* :: *FunName* ⇒ *ThetaMapping* ⇒ *PhiMapping* ⇒ *bool*

where
valid-f f $(\vartheta 1,\vartheta 2)$ Φ =
$(\forall\ e.\ (set\ (varsAPP\ \Sigma d\ f)) \cup fv\ (e\ ()::unit\ Exp) \subseteq dom\ \vartheta 1$
$\wedge$ *fvReg* $(e\ ()::unit\ Exp) \cup \{\varrho self\text{-}f\ f\} \subseteq dom\ \vartheta 2$
$\wedge$ $(set\ (varsAPP\ \Sigma d\ f)) \cup fv\ (e\ ()::unit\ Exp) \subseteq dom\ \Phi$
$\wedge$ $(\forall\ E1\ E2\ h\ k\ td\ hh\ v\ r\ \eta\ si\ .$
SafeRASem $(E1,E2)\ h\ k\ td\ (e\ ()::unit\ Exp)\ hh\ k\ v\ r$
$\wedge\ \ si = build\text{-}si\ E1\ h\ (varsAPP\ \Sigma d\ f)$
$\wedge\ \ admissible\ f\ \eta\ k$
$\longrightarrow consistent\ (\vartheta 1,\vartheta 2)\ \eta\ (E1,E2)\ h$
$\wedge\ (\forall\ y \in dom\ \Phi.\ the\ ((the\ (\Phi\ y))\ si) >= sizeEnv\ E1\ y\ h)))$

constdefs *valid* :: *FunDefEnv* $\Rightarrow$ *FunTypesEnv* $\Rightarrow$ *FunSizesEnv* $\Rightarrow$ *bool*
valid Σ-*d* Σ-ϑ Σ-$\Phi \equiv (\forall\ f \in dom\ \Sigma\text{-}d.\ valid\text{-}f\ f\ (typesAPP\ \Sigma\text{-}\vartheta\ f)\ (sizesAPP\ \Sigma\text{-}\Phi\ f))$

declare *valid-f.simps* [*simp del*]
declare *valid-def* [*simp del*]

fun *SafeResourcesDAss*::
unit Exp $\Rightarrow$
FunName $\Rightarrow$ *ThetaMapping* $\Rightarrow$ *PhiMapping* $\Rightarrow$ *real* $\Rightarrow$
AbstractDelta $\Rightarrow$ *AbstractMu* $\Rightarrow$ *AbstractSigma* $\Rightarrow$ *bool*
(- :- - - - $\{\!|$ - , - , - $|\!\}$ *1000*)
where
SafeResourcesDAss e f $(\vartheta 1,\vartheta 2)$ Φ *td* Δ μ σ =
$(valid\ \Sigma d\ \Sigma\vartheta\ \Sigma\Phi \longrightarrow$ $(*\ valid\ *)$
$(set\ (R\text{-}Args\ \Sigma t\ f)) \cup \{\varrho self\text{-}f\ f\} = dom\ \Delta$ $(*\ P\text{-}static\ *)$
$\wedge\ (\forall\ E1\ E2\ h\ k\ hh\ v\ \delta\ m\ s\ \eta\ si.$
SafeRASemanticsReal.SafeRASem $(E1,E2)\ h\ k\ td\ e\ hh\ k\ v\ (\delta,m,s)$
$\wedge\ (set\ (varsAPP\ \Sigma d\ f)) \cup fv\ e \subseteq dom\ E1 \wedge fvReg\ e \subseteq dom\ E2$ $(*\ P\text{-}dyn\ *)$
$\wedge\ dom\ \Delta = dom\ \eta$
$\wedge\ si = build\text{-}si\ E1\ h\ (varsAPP\ \Sigma d\ f)$ $(*\ P\text{-}size\ *)$
$\wedge\ admissible\ f\ \eta\ k$ $(*\ P\text{-}\eta\ *)$
$\longrightarrow Delta\text{-}ge\ \Delta\ si\ k\ \eta\ \delta$ $(*\ P\text{-}\Delta\ *)$
$\wedge\ mu\text{-}ge\ \mu\ si\ m$ $(*\ P\text{-}\mu\ *)$
$\wedge\ sigma\text{-}ge\ \sigma\ si\ s))$

constdefs *regionsType* :: *string* $\Rightarrow$ *string set*
regionsType f $\equiv$ (*case* $\Sigma t\ f$ *of Some* $(ti,\ \varrho s,\ t) \Rightarrow regions'\ ti \cup regions\ t \cup set\ \varrho s$)

constdefs *constantCost* :: *real* ⇒ *Cost* ([]_ *1000*)
constantCost r ≡ (λ*xs*. *Some r*)

constdefs *emptyAbstractDelta* :: *string* ⇒ *AbstractDelta*
([]_ *1000*)
emptyAbstractDelta f ==
(%ϱ . *if* ϱ ∈ (*set* (*R-Args* Σ*t f*)) ∪ {ϱ*self-f f*}
then Some []$_0$
else None)

fun *foldr1* :: ('*a* ⇒ '*a* ⇒ '*a*) ⇒ '*a list* ⇒ '*a*
where
foldr1 f [*x*] = *x*
| *foldr1 f* (*x*#*xs*) = *f x* (*foldr1 f xs*)

constdefs *addCost* :: *Cost* ⇒ *Cost* ⇒ *Cost* (- +$_c$ - *1000*)
c1 +$_c$ *c2* ≡ (λ *x*. *if* (*x* ∈ *dom c1* ∩ *dom c2*) *then Some* (*the* (*c1 x*) + *the* (*c2 x*))
else None)

constdefs *addCostList* :: *Cost list* ⇒ *Cost*
addCostList ≡ *foldr1 addCost*

constdefs *addAbstractDelta* :: *AbstractDelta* ⇒ *AbstractDelta* ⇒ *AbstractDelta*
(- +$_\Delta$ - *1000*)
addAbstractDelta Δ*1* Δ*2* ==
(%ϱ . *if* (ϱ ∈ *dom* Δ*1* ∩ *dom* Δ*2*)
then Some ((*the* (Δ*1* ϱ)) +$_c$ (*the* (Δ*2* ϱ)))
else None)

constdefs *addCostReal* :: *Cost* ⇒ *real* ⇒ *Cost*
addCostReal c r ≡ (λ *x*. *if x* ∈ *dom c then Some* (*the* (*c x*) + *r*)
else None)

constdefs *substCostReal* :: *Cost* ⇒ *real* ⇒ *Cost*
substCostReal c r ≡ (λ *x*. *if x* ∈ *dom c then Some* (*the* (*c x*) − *r*)
else None)

constdefs *maxCost* :: *Cost* ⇒ *Cost* ⇒ *Cost* (⨆$_c$ { - , - } *1000*)
⨆$_c$ { *c1* , *c2* } ≡ (λ *x*. *if x* ∈ *dom c1* ∩ *dom c2 then Some* (*max* (*the* (*c1 x*))

$(the\ (c2\ x)))$
$else\ None)$

constdefs $maxAbstractDelta :: AbstractDelta \Rightarrow AbstractDelta \Rightarrow AbstractDelta$
$maxAbstractDelta\ \Delta 1\ \Delta 2 \equiv (\%\varrho\ .\ if\ (\varrho \in dom\ \Delta 1 \cap dom\ \Delta 2)$
$then\ Some \bigsqcup_c\ \{\ the\ (\Delta 1\ \varrho)\ ,\ the\ (\Delta 2\ \varrho)\ \}$
$else\ None)$

constdefs $maxCostList :: Cost\ list \Rightarrow Cost\ (\bigsqcup_c\ -\ 1000)$
$maxCostList\ xs \equiv foldr\ maxCost\ xs\ []_0$

constdefs $maxAbstractDeltaList :: FunName \Rightarrow AbstractDelta\ list \Rightarrow Abstract\text{-}Delta$
$(\bigsqcup_\Delta\ --\ 1000)$
$maxAbstractDeltaList\ f\ xs \equiv foldr\ maxAbstractDelta\ xs\ (emptyAbstractDelta\ f)$

constdefs $sizeAbstractDelta :: AbstractDelta \Rightarrow Cost\ (|\text{-}|\ 1000)$
$sizeAbstractDelta\ d \equiv fold\ addCost\ (\%x.\ x)\ []_0\ (ran\ d)$

fun $AbstractDeltaSpaceCost :: AbstractDelta \times AbstractMu \times AbstractSigma \Rightarrow AbstractDelta$
where
$AbstractDeltaSpaceCost\ (\Delta,\ \mu,\ \sigma) = \Delta$

fun $AbstractMuSpaceCost :: AbstractDelta \times AbstractMu \times AbstractSigma \Rightarrow AbstractMu$
where
$AbstractMuSpaceCost\ (\Delta,\ \mu,\ \sigma) = \mu$

fun $AbstractSigmaSpaceCost :: AbstractDelta \times AbstractMu \times AbstractSigma \Rightarrow AbstractSigma$
where
$AbstractSigmaSpaceCost\ (\Delta,\ \mu,\ \sigma) = \sigma$

constdefs $num\text{-}r :: ('a\ Patron \times 'a\ Exp) \Rightarrow real$
$num\text{-}r\ alt \equiv real\ (length\ (snd\ (extractP\ (fst\ alt))))$

constdefs $list\text{-}ge :: real\ list \Rightarrow real\ list \Rightarrow bool$
$list\text{-}ge\ xs\ ys == (\forall\ i < length\ xs.\ xs!i \geq ys!i)$

constdefs *monotonic-bound* :: (*real list* ⇀ *real*) ⇒ *bool*
monotonic-bound b == (∀ *x* ∈ *dom b*. (∀ *y* ∈ *dom b*. *list-ge x y* ⟶ *the* (*b x*) ≥ *the* (*b y*)))

constdefs *monotonic-AbstractDelta* :: *AbstractDelta* ⇒ *bool*
monotonic-AbstractDelta Δ ≡ (∀ ϱ ∈ *dom* Δ. *monotonic-bound* (*the* (Δ ϱ)))

constdefs *defined-bound* :: (*real list* ⇀ *real*) ⇒ *FunName* ⇒ *bool*
defined-bound b f ≡ (∀ *xs*. *length xs* = *length* (*varsAPP* Σ*d f*) ⟶ *xs* ∈ *dom b*)

constdefs *defined-AbstractDelta* :: *AbstractDelta* ⇒ *FunName* ⇒ *bool*
defined-AbstractDelta Δ *f* ≡ (∀ ϱ ∈ *dom* Δ. *defined-bound* (*the* (Δ ϱ)) *f*)

fun *argP-aux* :: (*string* ⇀ *TypeExpression*) ⇒ *TypeExpression* ⇒ *'a Exp* ⇒ *bool*
where
argP-aux ϑ *t* (*ConstE* (*LitN* -) -) = (*t* = (*ConstrT intType* [] []))
| *argP-aux* ϑ *t* (*ConstE* (*LitB* -) -) = (*t* = (*ConstrT boolType* [] []))
| *argP-aux* ϑ *t* (*VarE x* -) = (ϑ *x* = *Some t*)

fun *argP* :: (*string* ⇀ *string*) ⇒ *string list* ⇒ *RegMapping* ⇒ *string list* ⇒ *bool*
where
argP ψ ϱ*s* ϑ2 *rs* = (*length* ϱ*s* = *length rs*
∧ (∀ *i* < *length* ϱ*s*. ψ (ϱ*s*!*i*) = ϑ2 (*rs*!*i*)))

constdefs η-*ef* :: *InstantiationMapping* ⇒ (*string* ⇀ *string*) ⇒ *string list*
⇒ *InstantiationMapping*
η-*ef* η φ ϱ*s* ≡ (%ϱ. η (*the* (φ ϱ))) |‘ (*set* ϱ*s*)

consts *phiMapping-app* :: (*Cost option*) *list* ⇒ *real list* ⇒ *real list*
primrec
phiMapping-app [] *xs* = []
phiMapping-app (*c*#*cs*) *xs* = (*case c of Some c'* ⇒ *the* (*c' xs*)# (*phiMapping-app cs xs*))

constdefs *cost-PhiMapping* :: *Cost* ⇒ (*'a Exp*) *list* ⇒ *PhiMapping* ⇒ *Cost*
cost-PhiMapping c as φ ≡ (λ *xs*. *c* (*phiMapping-app* (*map* φ (*map atom2var as*)) *xs*))

constdefs *mu-app* :: *AbstractMu* ⇒ (*'a Exp*) *list* ⇒ *PhiMapping* ⇒ *AbstractMu*
mu-app μ*g as* φ ≡ *cost-PhiMapping* μ*g as* φ

constdefs *sigma-app* :: *AbstractSigma* ⇒ (*'a Exp*) *list* ⇒ *PhiMapping* ⇒ *AbstractSigma*

sigma-app σg as φ ≡ cost-PhiMapping σg as φ

constdefs
setsumCost :: ('a => Cost) => 'a set => Cost
setsumCost f A ≡ if finite A then fold addCost f []$_0$ A else []$_0$
constdefs *sum-rho :: FunName ⇒ AbstractDelta ⇒ (string ⇀ string) ⇒ PhiMapping*
⇒ ('a Exp) list ⇒ string ⇒ Cost
sum-rho g Δg ψ Φ as ϱ ≡ (%xs. (setsumCost
(λ ϱ. the (Δg ϱ))
{ϱ'. ψ ϱ' = Some ϱ ∧ ϱ' ∈ set (R-Args Σt g)})
(phiMapping-app (map Φ (map atom2var as)) xs))

constdefs *instance-f ::*
FunName ⇒ FunName ⇒ AbstractDelta ⇒ (string ⇀ string) ⇒ PhiMapping
⇒ ('a Exp) list ⇒ AbstractDelta
instance-f f g Δg ψ Φ as ≡
(%ϱ. Some (sum-rho g Δg ψ Φ as ϱ)) |` ((set (R-Args Σt f)) ∪ {ϱself-f f})

consts *varToExp :: string list ⇒ 'a ⇒ 'a Exp list*
primrec
varToExp [] a = []
varToExp (v#vs) a = VarE v a # varToExp vs a

constdefs *projection :: FunName ⇒ AbstractDelta ⇒ AbstractDelta*
projection f Δ ≡ (%ϱ. if (ϱ = ϱself-f f) then None else Δ ϱ)

inductive
ValidGlobalResourcesEnv :: FunctionResourcesSignature ⇒ bool
(⊨⊨ - 1000)

where
base: ⊨⊨ *empty*
| *step*: ⟦ ⊨⊨ Σb; *f* ∉ *dom* Σb;
(*bodyAPP* Σd *f*) :*f* (*typesAPP* Σϑ *f*) (*sizesAPP* ΣΦ *f*) *real*(*length* (*varsAPP* Σd *f*) + *length* (*regionsAPP*
{| Δ, μ, σ |} ⟧ ⟹ ⊨⊨ Σb(*f*↦(*projection f* Δ,μ,σ))

fun *SafeResourcesDAssCntxt* ::
unit Exp ⇒ FunctionResourcesSignature ⇒
FunName ⇒ ThetaMapping ⇒ PhiMapping ⇒ real ⇒
AbstractDelta ⇒ AbstractMu ⇒ AbstractSigma ⇒ bool
(- , - : - - - {| - , - , - |} 1000)

where
SafeResourcesDAssCntxt e Σb f (ϑ1,ϑ2) Φ td Δ μ σ =
(⊨⊨ Σb ⟶ *e* :*f* (ϑ1,ϑ2) Φ *td* {| Δ, μ, σ |})

fun *SafeResourcesDAssDepth*::
unit Exp ⇒
FunName ⇒ *ThetaMapping* ⇒ *PhiMapping* ⇒ *real* ⇒ *nat* ⇒
AbstractDelta ⇒ *AbstractMu* ⇒ *AbstractSigma* ⇒ *bool*
(- :- - - - , - ⦃ - , - , - ⦄ *1000*)
where
SafeResourcesDAssDepth e f (*ϑ1*,*ϑ2*) Φ *td n* Δ *μ σ* =
(*valid* Σ*d* Σ*ϑ* ΣΦ ⟶ (∗ *valid* ∗)
(*set* (*R-Args* Σ*t f*)) ∪ {*ϱself-f f*} = *dom* Δ (∗ *P-static* ∗)
∧ (∀ *E1 E2 h k hh v δ m s η si*.
SafeDepthSemanticsReal.SafeBoundSem (*E1*,*E2*) *h k td e* (*f*,*n*) *hh k v*
(*δ*,*m*,*s*)
∧ (*set* (*varsAPP* Σ*d f*)) ∪ *fv e* ⊆ *dom E1* ∧ *fvReg e* ⊆ *dom E2* (∗ *P-dyn* ∗)
∧ *dom* Δ = *dom η*
∧ *si* = *build-si E1 h* (*varsAPP* Σ*d f*) (∗ *P-size* ∗)
∧ *admissible f η k* (∗ *P-η* ∗)
⟶ *Delta-ge* Δ *si k η δ* (∗ *P-*Δ ∗)
∧ *mu-ge μ si m* (∗ *P-μ* ∗)
∧ *sigma-ge σ si s*))

inductive *ValidGlobalResourcesEnvDepth* ::
string ⇒ *nat* ⇒ *FunctionResourcesSignature* ⇒ *bool*
(⊨⊨- , - - *1000*)
where
base : ⟦ ⊨⊨ Σ*b*; *f* ∉ *dom* Σ*b*⟧ ⟹ ⊨⊨$_{f,n}$ Σ*b*
| *depth0* : ⟦ ⊨⊨ Σ*b*; *f* ∉ *dom* Σ*b*⟧ ⟹ ⊨⊨$_{f,0}$ Σ*b*(*f*↦ (*projection f* Δ,*μ*,*σ*))
| *step* : ⟦ ⊨⊨ Σ*b*; *f* ∉ *dom* Σ*b*;
(*bodyAPP* Σ*d f*) :$_f$ (*typesAPP* Σ*ϑ f*) (*sizesAPP* ΣΦ *f*) *real*(*length* (*varsAPP* Σ*d f*) + *length* (*regionsAPP*
⦃ Δ, *μ*, *σ* ⦄ ⟧ ⟹ ⊨⊨$_{f,Suc\ n}$ Σ*b*(*f*↦(*projection f* Δ,*μ*,*σ*))
| *g* : ⟦ ⊨⊨$_{f,n}$ Σ*b*; *g* ∉ *dom* Σ*b*; *g*≠*f*;
(*bodyAPP* Σ*d g*) :$_g$ (*typesAPP* Σ*ϑ g*) (*sizesAPP* ΣΦ *g*) *real*(*length* (*varsAPP* Σ*d g*) + *length* (*regionsAPP*
⦃ Δ, *μ*, *σ* ⦄ ⟧ ⟹ ⊨⊨$_{f,\ n}$ Σ*b*(*g*↦(*projection g* Δ,*μ*,*σ*))

fun *SafeResourcesDAssDepthCntxt* ::
unit Exp ⇒ *FunctionResourcesSignature* ⇒
FunName ⇒ *ThetaMapping* ⇒ *PhiMapping* ⇒ *real* ⇒ *nat* ⇒
AbstractDelta ⇒ *AbstractMu* ⇒ *AbstractSigma* ⇒ *bool*
(- , - :- - - - , - ⦃ - , - , - ⦄ *1000*)
where
SafeResourcesDAssDepthCntxt e Σ*b f* (*ϑ1*,*ϑ2*) Φ *td n* Δ *μ σ* =
(⊨⊨$_{f,n}$ Σ*b* ⟶ *e* :$_{f\ (\vartheta 1,\vartheta 2)\ \Phi\ td,\ n}$ ⦃ Δ, *μ*, *σ* ⦄)

end

19 Auxiliary lemmas of the soundness theorem

theory *CostesDepth*
imports *Costes-definitions*
begin

lemma *dom-emptyAbstractDelta*:
  dom $[]_f$ = (*set* (*R-Args* $\Sigma t\ f$)) $\cup$ {ϱ*self-f f*}
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp add*: *emptyAbstractDelta-def*,*clarsimp*)
apply (*split split-if-asm*,*simp*,*simp*)
apply (*simp add*: *emptyAbstractDelta-def*)
by (*rule conjI*,*force*,*force*)

lemma *Delta-ge-emptyAbstractDelta-emptyDelta*:
  ⟦ *dom* $[]_f$ = *dom* η ⟧
  ⟹ *Delta-ge* $[]_f$ *si k* η (*SafeRASemanticsReal.emptyDelta k*)
apply (*subst* (*asm*) *dom-emptyAbstractDelta*)
apply (*simp add*: *Delta-ge-def*)
apply (*simp add*: *sizeAbstractDelta-si-def*)
apply (*simp add*: *emptyDelta-def*)
apply (*simp add*: *emptyAbstractDelta-def*)
apply (*rule ballI*)
apply (*rule setsum-nonneg*)
apply (*simp add*: *constantCost-def*)
apply (*rule allI*, *intro impI*)
by *clarsimp*

lemma *mu-ge-constantCost*:
  mu-ge $[]_0$ *si 0*
by (*simp add*: *mu-ge-def add*: *constantCost-def*)

lemma *sigma-ge-constantCost*:
  sigma-ge $[]_1$ *si 1*
by (*simp add*: *sigma-ge-def add*: *constantCost-def*)

lemma *SafeResourcesDADepth-LitInt*:
  ConstE (*LitN i*) *a* $:_f$ ($\vartheta 1$, $\vartheta 2$) φ *td*, *n*
  {$[]_f$, $[]_0$, $[]_1$}

apply (*simp only*: *SafeResourcesDAssDepth.simps*)
apply (*rule impI*)
apply (*rule conjI*)

apply (*subst dom-emptyAbstractDelta*,*simp*)

apply (*intro allI*,*rule impI*)
apply (*elim conjE*)
apply (*frule impSemBoundRA* [**where** *td*=*td*])
apply (*erule SafeRASem.cases*,*simp-all*)
apply *clarsimp*
apply (*rule conjI*)

apply (*rule Delta-ge-emptyAbstractDelta-emptyDelta*,*assumption*+)
apply (*rule conjI*)

apply (*rule mu-ge-constantCost*)

by (*rule sigma-ge-constantCost*)

lemma *SafeResourcesDADepth-LitBool*:
ConstE (*LitB b*) *a* :$_f$ (*ϑ1*, *ϑ2*) *φ td*, *n* {|$\square_f$, $\square_0$, $\square_1$|}
apply (*simp only*: *SafeResourcesDAssDepth.simps*)
apply (*rule impI*)
apply (*rule conjI*)

apply (*subst dom-emptyAbstractDelta*,*simp*)

apply (*intro allI*,*rule impI*)
apply (*elim conjE*)
apply (*frule impSemBoundRA* [**where** *td*=*td*])
apply (*erule SafeRASem.cases*,*simp-all*)
apply *clarsimp*
apply (*rule conjI*)

apply (*rule Delta-ge-emptyAbstractDelta-emptyDelta*,*assumption*+)
apply (*rule conjI*)

apply (*rule mu-ge-constantCost*)

by (*rule sigma-ge-constantCost*)

axioms *SafeResourcesDADepth-Var1*:
$VarE\ x\ a :_f (\vartheta 1, \vartheta 2)\ \varphi\ td, n$
$\{\!|[]_f , []_0, []_1|\!\}$

axioms *SafeResourcesDADepth-Var2*:
$[\![\vartheta 2\ r = Some\ \varrho; \varphi\ x = Some\ \eta]\!]$
$\Longrightarrow CopyE\ x\ r\ d :_f (\vartheta 1, \vartheta 2)\ \varphi\ td, n \ \{\!|[\varrho \mapsto \eta], \eta, []_2|\!\}$

axioms *SafeResourcesDADepth-Var3*:
$[\![\vartheta 2\ r = Some\ \varrho; \varphi\ x = Some\ \eta]\!]$
$\Longrightarrow ReuseE\ x\ a :_f (\vartheta 1, \vartheta 2)\ \varphi\ td, n$
$\{\!|[]_f , []_0, []_1|\!\}$

axioms *SafeResourcesDADepth-APP-PRIMOP*:
$[\![primops\ g = Some\ oper]\!]$
$\Longrightarrow AppE\ g\ [a1,a2]\ []\ a :_f (\vartheta 1, \vartheta 2)\ \varphi\ td, n$
$\{\!|[]_g , []_0, []_2|\!\}$

lemma *dom-addAbstractDelta*:
$dom\ (\Delta 1 +_\Delta\ \Delta 2) = dom\ \Delta 1 \cap dom\ \Delta 2$
apply (*simp add*: *addAbstractDelta-def*)
apply *auto*
by (*split split-if-asm*,*force*,*force*)+

lemma *P-static-dom-Δ-Let*:
$[\![set\ (R\text{-}Args\ \Sigma t\ f) \cup \{\varrho self\text{-}f\ f\} = dom\ \Delta 1;$
$set\ (R\text{-}Args\ \Sigma t\ f) \cup \{\varrho self\text{-}f\ f\} = dom\ \Delta 2]\!]$
$\Longrightarrow set\ (R\text{-}Args\ \Sigma t\ f) \cup \{\varrho self\text{-}f\ f\} = dom\ \Delta 1 +_\Delta\ \Delta 2$
by (*subst dom-addAbstractDelta*,*simp*)

lemma *P1-f-n-LET*:
$[\![\forall C\ as\ r\ a'.\ e1 \neq ConstrE\ C\ as\ r\ a';$
$SafeDepthSemanticsReal.SafeBoundSem\ (E1, E2)\ h\ k\ td\ (Let\ x1 = e1\ In\ e2\ a)$
$(f, n)\ hh\ k\ v\ (\delta, m, s)]\!]$
$\Longrightarrow \exists h'\ v1\ v2\ \delta 1\ m1\ s1\ \delta 2\ m2\ s2\ n1\ n2.$
$SafeDepthSemanticsReal.SafeBoundSem\ (E1, E2)\ h\ k\ 0\ e1$
$(f, n)\ h'\ k\ v1\ (\delta 1, m1, s1)$
$\wedge\ SafeDepthSemanticsReal.SafeBoundSem\ (E1(x1 \mapsto v1), E2)\ h'\ k\ (td + 1)\ e2$
$(f, n)\ hh\ k\ v2\ (\delta 2, m2, s2)$

$\wedge\ \ \delta = SafeRASemanticsReal.addDelta\ \delta 1\ \delta 2$
$\wedge\ m = max\ m1\ (m2 + SafeRASemanticsReal.balanceCells\ \delta 1)$
$\wedge\ s = max\ (s1 + 2)\ (s2 + 1)$
$\wedge\ x1 \notin dom\ E1$
apply (*simp add*: *SafeBoundSem-def*)
apply (*elim exE*, *elim conjE*)
apply (*erule SafeDepthSem.cases*,*simp-all*)
apply (*elim conjE*)
apply (*rule-tac x*=h' **in** *exI*)
apply (*rule-tac x*=*v1* **in** *exI*)
apply (*rule-tac x*=*v2* **in** *exI*)
apply (*rule-tac x*=$\delta 1$ **in** *exI*)
apply (*rule-tac x*=*m1* **in** *exI*)
apply (*rule-tac x*=*s1* **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac x*=*n1* **in** *exI*)+
apply *simp*
apply (*rule-tac x*=$\delta 2$ **in** *exI*)
apply (*rule-tac x*=*m2* **in** *exI*)
apply (*rule-tac x*=*s2* **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac x*=*n2* **in** *exI*)+
apply *simp*
by *simp*

lemma *P-dyn-dom-E1-Let-e1*:
$[\![\ set\ (varsAPP\ \Sigma d\ f) \cup fv\ (Let\ x1 = e1\ In\ e2\ a) \subseteq dom\ E1;$
$x1 \notin fv\ e1\ ]\!]$
$\Longrightarrow set\ (varsAPP\ \Sigma d\ f) \cup fv\ e1 \subseteq dom\ E1$
by (*simp*,*elim conjE*, *blast*)

lemma *P-dyn-dom-E1-Let-e2*:
$[\![\ set\ (varsAPP\ \Sigma d\ f) \cup fv\ (Let\ x1 = e1\ In\ e2\ a) \subseteq dom\ E1;$
$x1 \notin fv\ e1\ ]\!]$
$\Longrightarrow\ set\ (varsAPP\ \Sigma d\ f) \cup fv\ e2 \subseteq dom\ (E1(x1 \mapsto v1))$
by (*simp*,*elim conjE*, *blast*)

lemma *P-dyn-dom-Δ-Let-e1*:
$[\![\ set\ (R\text{-}Args\ \Sigma t\ f) \cup \{\varrho self\text{-}f\ f\} = dom\ \Delta 1;$
$set\ (R\text{-}Args\ \Sigma t\ f) \cup \{\varrho self\text{-}f\ f\} = dom\ \Delta 2;$
$dom\ \Delta 1 +_\Delta\ \Delta 2\ = dom\ \eta\ ]\!]$
$\Longrightarrow dom\ \Delta 1 = dom\ \eta$
by (*subst* (*asm*) *dom-addAbstractDelta*,*simp*)

lemma *P-dyn-dom-Δ-Let-e2*:
$[\![\ set\ (R\text{-}Args\ \Sigma t\ f) \cup \{\varrho self\text{-}f\ f\} = dom\ \Delta 1;$
$set\ (R\text{-}Args\ \Sigma t\ f) \cup \{\varrho self\text{-}f\ f\} = dom\ \Delta 2;$
$dom\ \Delta 1 +_\Delta\ \Delta 2\ = dom\ \eta\ ]\!]$

$\Longrightarrow$ *dom* $\Delta 1$ = *dom* η
by (*subst* (*asm*) *dom-addAbstractDelta*,*simp*)

lemma *length-build-si*:
length (*build-si E1 h xs*) = *length xs*
by (*induct xs*,*simp-all*)

lemma *sizeAbstractDelta-si-addf*:
⟦ *set* (*R-Args* Σt *f*) $\cup$ {ϱ*self-f f*} = *dom* $\Delta 1$; *defined-AbstractDelta* $\Delta 1$ *f*;
set (*R-Args* Σt *f*) $\cup$ {ϱ*self-f f*} = *dom* $\Delta 2$; *defined-AbstractDelta* $\Delta 2$ *f*;
dom $\Delta 1$ $+_\Delta$ $\Delta 2$ = *dom* η ⟧
$\Longrightarrow$ *sizeAbstractDelta-si* $\Delta 1$ $+_\Delta$ $\Delta 2$ (*build-si E1 h* (*varsAPP* Σd *f*)) *j* η
= *sizeAbstractDelta-si* $\Delta 1$ (*build-si E1 h* (*varsAPP* Σd *f*)) *j* η + *sizeAbstractDelta-si* $\Delta 2$ (*build-si E1 h* (*varsAPP* Σd *f*)) *j* η
apply (*frule P-static-dom-*Δ*-Let*,*simp*)
apply (*subgoal-tac* *dom* $\Delta 1$ $+_\Delta$ $\Delta 2$ = *dom* $\Delta 2$)
apply (*subgoal-tac dom* $\Delta 1$ = *dom* $\Delta 2$,*simp*)
apply (*simp only*: *sizeAbstractDelta-si-def*)
apply (*subgoal-tac* ($\sum \varrho$ | η ϱ = *Some j*. *the* (*the* ($\Delta 1$ $+_\Delta$ $\Delta 2$ ϱ) (*build-si E1 h* (*varsAPP* Σd *f*)))) =
($\sum \varrho$ | η ϱ = *Some j*. *the* (*the* ($\Delta 1$ ϱ) (*build-si E1 h* (*varsAPP* Σd *f*))) + *the* (*the* ($\Delta 2$ ϱ) (*build-si E1 h* (*varsAPP* Σd *f*)))),*simp*)
apply (*subst setsum-addf*)
apply (*simp only*: *addAbstractDelta-def*)
apply (*rule setsum-cong2*)
apply (*simp add*: *addAbstractDelta-def*)
apply (*rule conjI*)
apply (*rule impI*)+
apply (*simp add*: *addCost-def*)
apply (*rule impI*)
apply (*simp add*: *defined-AbstractDelta-def*)
apply (*erule-tac x*=*x* **in** *ballE*)
prefer *2* **apply** *simp*
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x*=(*build-si E1 h* (*varsAPP* Σd *f*)) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*)
apply *simp*
apply (*erule-tac x*=*x* **in** *ballE*)
prefer *2* **apply** *simp*
apply (*erule-tac x*=(*build-si E1 h* (*varsAPP* Σd *f*)) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*)
apply *simp*
apply (*simp add*: *dom-def*)
apply *simp*
by *simp*

lemma *sizeAbstractDelta-si-nonneg*:

$[\![$ *dom* Δ = *dom* η;
$\forall \varrho \in dom\ \Delta.\ 0 \leq the\ (the\ (\Delta\ \varrho)\ si)\ ]\!]$
$\Longrightarrow$ *sizeAbstractDelta-si* Δ *si j* $\eta \geq 0$
apply (*simp add*: *sizeAbstractDelta-si-def*)
by (*rule setsum-nonneg*,*simp*,*force*)

axioms *wellFormed-*Δ:
$\forall \varrho \in dom\ \Delta.$
0 $\leq$ *the* (*the* (Δ ϱ) (*build-si E1 h* (*varsAPP* Σ*d f*)))

axioms *semantic-no-capture-E1*:
$[\![$ *SafeDepthSemanticsReal.SafeBoundSem* (*E1*, *E2*) *h k td e* (*f*, *n*) *h'* *k v* (δ, *m*, *s*);
E1 x = *Some* (*Loc p*);
p $\notin$ *dom h* $]\!]$
$\Longrightarrow$ *p* $\notin$ *dom h'*

axioms *semantic-extend-pointers*:
SafeDepthSemanticsReal.SafeBoundSem (*E1*, *E2*) *h k td e* (*f*, *n*) *h'* *k v* (δ, *m*, *s*)
$\Longrightarrow$ ($\forall$ *p*$\in$*dom h*. *h p* = *h' p*)

axioms *size-equals-h-h'*:
v = *Loc p*
$\Longrightarrow$ *size v h'* = *Costes-definitions.size v h*

axioms *axiom-size-dom*:
($\forall$ *x* $\in$ *set* (*varsAPP* Σ*d f*). *size-dom* (*the* (*E1 x*), *h*))

lemma *sizeEnv-equals-h-h'*:
$[\![$ *x1* $\notin$ *dom E1*;
SafeDepthSemanticsReal.SafeBoundSem (*E1*, *E2*) *h k td e* (*f*, *n*) *h'* *k v* (δ, *m*, *s*);
x1 $\notin$ *set xs*; *x1* $\neq$ *x*;
size-dom (*the* (*E1 x*), *h*); *size-dom* (*the* (*E1 x*), *h'*);
($\forall$ *x* $\in$ *set xs*. *size-dom* (*the* (*E1 x*), *h*) $\wedge$ *size-dom* (*the* (*E1 x*),*h'*))$]\!]$
$\Longrightarrow$ *sizeEnv E1 x h* = *sizeEnv* (*E1*(*x1* $\mapsto$ *v1*)) *x h'*
apply (*subgoal-tac sizeEnv* (*E1*(*x1* $\mapsto$ *v1*)) *x h'* = *sizeEnv E1 x h'*,*simp*)
prefer *2* **apply** (*simp add*: *sizeEnv-def*)
apply (*frule semantic-extend-pointers*)
apply (*simp add*: *sizeEnv-def*)
apply (*case-tac E1 x*,*simp-all*)
apply (*case-tac a*)

apply (*rename-tac* v' p)
apply (*case-tac* $p \notin dom\ h$)

apply (*frule semantic-no-capture-E1*,*simp*,*assumption*+)
apply (*subgoal-tac* $h\ p = None \wedge h'\ p = None$)
apply (*simp add*: *size.psimps*(*3*))
apply *force*

apply (*subgoal-tac* $\exists\ j\ C\ vn.\ h\ p = Some\ (j,C,vn)$)
prefer *2* **apply** *force*
apply (*elim exE*)
apply (*rule size-equals-h-h'*,*assumption*)

apply *simp*

by *simp*

lemma *build-si-equals-h-h'* [*rule-format*]:
$x1 \notin dom\ E1$
$\longrightarrow$ *SafeDepthSemanticsReal.SafeBoundSem* $(E1,\ E2)\ h\ k\ td\ e\ (f,\ n)\ h'\ k\ v\ (\delta,\ m,\ s)$
$\longrightarrow x1 \notin set\ xs$
$\longrightarrow (\forall\ x \in set\ xs.\ size\text{-}dom\ (the\ (E1\ x),\ h) \wedge size\text{-}dom\ (the\ (E1\ x),h'))$
$\longrightarrow (build\text{-}si\ E1\ h\ xs) = (build\text{-}si\ (E1(x1 \mapsto v1))\ h'\ xs)$
apply (*rule impI*)
apply (*induct xs*,*simp-all*)
apply (*intro impI*)
apply (*drule mp*,*simp*)+
apply (*elim conjE*)
by (*rule sizeEnv-equals-h-h'*,*assumption*+)

lemma *P-Δ-Let*:
$[\![\ set\ (R\text{-}Args\ \Sigma t\ f) \cup \{\varrho self\text{-}f\ f\} = dom\ \Delta 1;\ defined\text{-}AbstractDelta\ \Delta 1\ f;$
$set\ (R\text{-}Args\ \Sigma t\ f) \cup \{\varrho self\text{-}f\ f\} = dom\ \Delta 2;\ defined\text{-}AbstractDelta\ \Delta 2\ f;$
$dom\ \Delta 1 +_\Delta \Delta 2 = dom\ \eta;$
$Delta\text{-}ge\ \Delta 1\ (build\text{-}si\ E1\ h\ (varsAPP\ \Sigma d\ f))\ k\ \eta\ \delta 1;$
$Delta\text{-}ge\ \Delta 2\ (build\text{-}si\ E1\ h\ (varsAPP\ \Sigma d\ f))\ k\ \eta\ \delta 2\ ]\!]$
$\Longrightarrow Delta\text{-}ge\ \Delta 1 +_\Delta \Delta 2\ (build\text{-}si\ E1\ h\ (varsAPP\ \Sigma d\ f))\ k\ \eta\ (\delta 1 \oplus \delta 2)$
apply (*unfold Delta-ge-def*)
apply (*rule ballI*)
apply (*subst sizeAbstractDelta-si-addf*,*simp*,*simp*,*simp*,*simp*,*simp*)
apply (*erule-tac x*=*j* **in** *ballE*)
prefer *2* **apply** *simp*
apply (*erule-tac x*=*j* **in** *ballE*)

prefer *2* **apply** *simp*
apply (*simp add*: *addDelta-def*)
apply (*rule conjI*)
apply (*rule impI*)
apply (*rule conjI*)
apply (*rule impI*,*clarsimp*)
apply (*rule impI*)
apply (*subgoal-tac 0* $\leq$ *sizeAbstractDelta-si* $\Delta 1$ (*build-si E1 h* (*varsAPP* Σd *f*)) *j* η,*simp*)
apply (*rule sizeAbstractDelta-si-nonneg*)
apply (*subst* (*asm*) *dom-addAbstractDelta*,*simp*)
apply (*rule wellFormed-*Δ)
apply (*rule impI*)
apply (*rule conjI*)
apply (*rule impI*)
apply (*subgoal-tac 0* $\leq$ *sizeAbstractDelta-si* $\Delta 2$ (*build-si E1 h* (*varsAPP* Σd *f*)) *j* η,*simp*)
apply (*rule sizeAbstractDelta-si-nonneg*)
apply (*subst* (*asm*) *dom-addAbstractDelta*,*simp*)
apply (*rule wellFormed-*Δ)
apply (*rule impI*)
apply (*subgoal-tac 0* $\leq$ *sizeAbstractDelta-si* $\Delta 2$ (*build-si E1 h* (*varsAPP* Σd *f*)) *j* η)
apply (*subgoal-tac 0* $\leq$ *sizeAbstractDelta-si* $\Delta 1$ (*build-si E1 h* (*varsAPP* Σd *f*)) *j* η)
apply *clarsimp*
apply (*rule sizeAbstractDelta-si-nonneg*)
apply (*subst* (*asm*) *dom-addAbstractDelta*,*simp*)
apply (*rule wellFormed-*Δ)
apply (*rule sizeAbstractDelta-si-nonneg*)
apply (*subst* (*asm*) *dom-addAbstractDelta*,*simp*)
by (*rule wellFormed-*Δ)

axioms *sizeAbstractDelta-*$\Delta 1$*-ge-balanceCells-*$\delta 1$:
Delta-ge $\Delta 1$ *si k* η $\delta 1$
$\Longrightarrow$ *the* (*sizeAbstractDelta* $\Delta 1$ *si*) $\geq$
SafeRASemanticsReal.balanceCells $\delta 1$

lemma *defined-AbstractDelta-si-in-dom-*$\Delta 1$:
defined-bound $|\Delta 1|$ *f*
$\Longrightarrow$ *build-si E1 h* (*varsAPP* Σd *f*) $\in$ *dom* $|\Delta 1|$
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x*=*build-si E1 h* (*varsAPP* Σd *f*) **in** *allE*)
by (*subst* (*asm*) *length-build-si*,*simp*)

lemma *P-*μ*-Let*:

⟦ *Delta-ge* Δ*1* (*build-si E1 h* (*varsAPP* Σ*d f*)) *k* η δ*1*;
defined-AbstractDelta Δ*1 f*; *defined-bound* |Δ*1*| *f*;
mu-ge μ*1* (*build-si E1 h* (*varsAPP* Σ*d f*)) *m1*; *defined-bound* μ*1 f*;
mu-ge μ*2* (*build-si E1 h* (*varsAPP* Σ*d f*)) *m2*; *defined-bound* μ*2 f* ⟧
⟹ *mu-ge* ⨆$_c$ { μ*1* , |Δ*1*| +$_c$ μ*2* } (*build-si E1 h* (*varsAPP* Σ*d f*)) (*max m1*
(*m2* + *SafeRASemanticsReal.balanceCells* δ*1*))
apply (*simp add*: *mu-ge-def*)
apply (*rule conjI*)
apply (*simp add*: *maxCost-def*)
apply (*rule conjI*)
apply (*rule impI*)
apply (*simp add*: *maxCost-def*)
apply (*rule impI*)+
apply (*simp add*: *defined-bound-def* [**where** *b*=μ*1*])
apply (*erule-tac x*= *build-si E1 h* (*varsAPP* Σ*d f*) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*simp add*: *defined-bound-def* [**where** *b*=μ*2*])
apply (*erule-tac x*= *build-si E1 h* (*varsAPP* Σ*d f*) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*frule-tac ?E1.0*=*E1* **and** *h*=*h* **in** *defined-AbstractDelta-si-in-dom-*Δ*1*)
apply (*simp add*: *addCost-def*)
apply (*simp add*: *dom-def*)
apply (*frule sizeAbstractDelta-*Δ*1-ge-balanceCells-*δ*1*)
apply (*simp add*: *maxCost-def*)
apply (*rule conjI*,*rule impI*)
apply (*simp add*: *addCost-def*)
apply (*rule conjI*,*rule impI*)
apply (*frule sizeAbstractDelta-*Δ*1-ge-balanceCells-*δ*1*,*simp*)
apply (*rule impI*)
apply (*simp add*: *defined-bound-def* [**where** *b*=μ*1*])
apply (*erule-tac x*= *build-si E1 h* (*varsAPP* Σ*d f*) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*simp add*: *defined-bound-def* [**where** *b*=μ*2*])
apply (*erule-tac x*= *build-si E1 h* (*varsAPP* Σ*d f*) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*frule-tac ?E1.0*=*E1* **and** *h*=*h* **in** *defined-AbstractDelta-si-in-dom-*Δ*1*)
apply *simp*
apply (*rule impI*)
apply (*simp add*: *addCost-def*)
apply (*simp add*: *defined-bound-def* [**where** *b*=μ*1*])
apply (*erule-tac x*= *build-si E1 h* (*varsAPP* Σ*d f*) **in** *allE*)
apply (*simp add*: *defined-bound-def* [**where** *b*=μ*2*])
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP* Σ*d f*) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*frule-tac ?E1.0*=*E1* **and** *h*=*h* **in** *defined-AbstractDelta-si-in-dom-*Δ*1*)
by (*simp add*: *dom-def*)

lemma *P-σ-Let*:
⟦ *sigma-ge σ1* (*build-si E1 h* (*varsAPP Σd f*)) *s1*; *defined-bound σ1 f*;
sigma-ge σ2 (*build-si E1 h* (*varsAPP Σd f*)) *s2*; *defined-bound σ2 f* ⟧
⟹ *sigma-ge* $\bigsqcup_c$ { $[]_2 +_c$ *σ1* , $[]_1 +_c$ *σ2* } (*build-si E1 h* (*varsAPP Σd f*))
(*max* (*s1* + *2*) (*s2* + *1*))
apply (*simp add*: *sigma-ge-def*)
apply (*rule conjI*)
apply (*simp add*: *maxCost-def*)
apply (*rule conjI*)
apply (*rule impI*)
apply (*simp add*: *constantCost-def*)
apply (*simp add*: *addCost-def*)
apply (*rule conjI*)
apply (*rule impI*)+
apply (*rule conjI*)
apply (*rule impI*,*simp*)
apply (*rule impI*)
apply (*simp add*: *defined-bound-def*)
apply (*rule impI*)
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP Σd f*) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP Σd f*) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply *clarsimp*
apply (*rule impI*)+
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP Σd f*) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP Σd f*) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*simp add*: *constantCost-def*)
apply (*simp add*: *addCost-def*)
apply (*simp add*: *dom-def*)
apply (*simp add*: *maxCost-def*)
apply (*rule conjI*)
apply (*rule impI*)
apply (*simp add*: *constantCost-def*)
apply (*simp add*: *addCost-def*)
apply (*rule conjI*)
apply (*rule impI*)+
apply (*rule conjI*)
apply (*rule impI*,*simp*)
apply (*rule impI*)
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP Σd f*) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP Σd f*) **in** *allE*)

apply (*subst* (*asm*) *length-build-si*,*simp*)
apply *clarsimp*
apply (*rule impI*)+
apply (*rule conjI*)
apply (*rule impI*)
apply (*simp add*: *defined-bound-def*)
apply (*rule impI*)+
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP* $\Sigma d\ f$) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP* $\Sigma d\ f$) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply *clarsimp*
apply (*rule impI*)+
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP* $\Sigma d\ f$) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP* $\Sigma d\ f$) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*simp add*: *constantCost-def*)
apply (*simp add*: *addCost-def*)
by (*simp add*: *dom-def*)

lemma *SafeResourcesDADepth-Let1*:
$[\![\ \forall C\ as\ r\ a'.\ e1 \neq ConstrE\ C\ as\ r\ a';$
$x1 \notin fv\ e1;$
$x1 \notin set\ (varsAPP\ \Sigma d\ f);$
$e1 :_f (\vartheta 1, \vartheta 2)\ \Phi\ 0,\ n\ \ \{\!|\Delta 1, \mu 1, \sigma 1|\!\};$
$defined\text{-}AbstractDelta\ \Delta 1\ f;\ defined\text{-}bound\ |\Delta 1|\ f;$
$defined\text{-}bound\ \mu 1\ f;\ defined\text{-}bound\ \sigma 1\ f;$
$e2 :_f (\vartheta 1, \vartheta 2)\ \Phi\ (td{+}1),\ n\ \ \{\!|\Delta 2, \mu 2, \sigma 2|\!\};$
$defined\text{-}AbstractDelta\ \Delta 2\ f;\ defined\text{-}bound\ \mu 2\ f;\ defined\text{-}bound\ \sigma 2\ f;$
$\Delta = \Delta 1\ +_\Delta\ \Delta 2;$
$\mu = \bigsqcup_c \{\mu 1, |\Delta 1| +_c\ \mu 2\};$
$\sigma = \bigsqcup_c \{[]_2 +_c\ \sigma 1,$
$[]_1 +_c\ \sigma 2\}\ ]\!]$
$\Longrightarrow Let\ x1 = e1\ In\ e2\ a :_f (\vartheta 1, \vartheta 2)\ \Phi\ td,\ n\ \ \{\!|\ \Delta, \mu\ , \sigma\ |\!\}$
apply (*simp only*: *SafeResourcesDAssDepth.simps*)

apply (*rule impI*)
apply (*drule mp*, *simp*)
apply (*drule mp*, *simp*)

apply (*elim conjE*)

apply (*rule conjI*)
apply (*rule P-static-dom-*Δ*-Let*,*simp*,*simp*)

apply (*intro allI*, *rule impI*)
apply (*elim conjE*)
apply (*frule P1-f-n-LET*,*simp*)
apply (*elim exE*)

apply (*erule-tac x*=*E1* **in** *allE*)
apply (*erule-tac x*=*E2* **in** *allE*)
apply (*erule-tac x*=*h* **in** *allE*)
apply (*erule-tac x*=*k* **in** *allE*)
apply (*erule-tac x*=*h'* **in** *allE*)
apply (*erule-tac x*=*v1* **in** *allE*)
apply (*erule-tac x*=δ*1* **in** *allE*)
apply (*erule-tac x*=*m1* **in** *allE*)
apply (*erule-tac x*=*s1* **in** *allE*)
apply (*erule-tac x*=η **in** *allE*)
apply (*erule-tac x*=*si* **in** *allE*)
apply (*drule mp*)
apply (*rule conjI*,*simp*)

apply (*rule conjI*, *rule P-dyn-dom-E1-Let-e1*, *assumption*+)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *frule P-dyn-dom-*Δ*-Let-e1*,*simp*,*simp*,*simp*)
apply (*rule conjI*, *simp*)
apply *simp*

apply (*erule-tac x*=*E1*($x1 \mapsto v1$) **in** *allE*)
apply (*erule-tac x*=*E2* **in** *allE*)
apply (*erule-tac x*=*h'* **in** *allE*)
apply (*erule-tac x*=*k* **in** *allE*)
apply (*erule-tac x*=*hh* **in** *allE*)
apply (*erule-tac x*=*v2* **in** *allE*)
apply (*erule-tac x*=δ*2* **in** *allE*)
apply (*erule-tac x*=*m2* **in** *allE*)
apply (*erule-tac x*=*s2* **in** *allE*)
apply (*erule-tac x*=η **in** *allE*)
apply (*erule-tac x*=*build-si* (*E1*($x1 \mapsto v1$)) h' (*varsAPP* $\Sigma d\ f$) **in** *allE*)
apply (*drule mp*)

apply (*rule conjI*,*simp*)

apply (*rule conjI*, *rule P-dyn-dom-E1-Let-e2*, *assumption*+)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *frule P-dyn-dom-Δ-Let-e2*, *simp*,*simp*,*simp*)
apply (*rule conjI*, *simp*)
apply *simp*

apply (*subgoal-tac* (∀ $x \in$ *set* (*varsAPP* Σ*d f*). *size-dom* (*the* (*E1 x*), *h*) ∧
size-dom (*the* (*E1 x*),*h*′)))
prefer *2* **apply** (*insert axiom-size-dom*,*force*)
apply (*subgoal-tac*
build-si E1 h (*varsAPP* Σ*d f*) =
build-si (*E1*(*x1* ↦ *v1*)) *h*′ (*varsAPP* Σ*d f*))
prefer *2* **apply** (*rule build-si-equals-h-h*′, *simp*,*force*,*assumption*+, *simp*,*simp*)

apply (*rule conjI*)
apply (*rule P-Δ-Let*)
apply (*simp*,*simp*,*simp*,*simp*,*simp*,*simp*,*simp*)

apply (*rule conjI*,*rule P-μ-Let*)
apply (*force*,*simp*,*simp*,*simp*,*simp*,*simp*,*simp*)

apply (*rule P-σ-Let*)
by (*simp*,*simp*,*simp*,*simp*)

axioms *SafeResourcesDADepth-Let2*:
⟦ ϑ*2 r* = *Some* ϱ; ϱ ∉ *dom* Δ;
e2 :*f* (ϑ*1*, ϑ*2*) φ *td*, *n* {|Δ, μ, σ|} ⟧
⟹*Let x1* = *ConstrE C as r a*′ *In e2 a* :*f* (ϑ*1*, ϑ*2*) φ *td*, *n* {|Δ ++ [ϱ ↦ []*1*], μ +*c* []*1*, σ*2* +*c* []*1*|}

lemma *dom-maxCost*:
dom (*maxCost c1 c2*) = *dom c1* ∩ *dom c2*

apply (*rule equalityI*)
apply (*rule subsetI*)
 apply (*simp add*: *maxCost-def*,*clarsimp*)
 apply (*split split-if-asm*,*simp*)
 apply (*simp add*: *maxCost-def*)
apply (*rule subsetI*)
 apply (*simp add*: *maxCost-def*,*clarsimp*)
 apply (*simp add*: *dom-def*)
done

lemma *dom-maxAbstractDelta*:
  dom (*maxAbstractDelta c1 c2*) = *dom c1* ∩ *dom c2*
apply (*rule equalityI*)
apply (*rule subsetI*)
 apply (*simp add*: *maxAbstractDelta-def*,*clarsimp*)
 apply (*split split-if-asm*,*simp*,*simp*)
apply (*rule subsetI*)
apply (*simp add*: *maxAbstractDelta-def*)
by (*clarsimp*,*simp add*: *dom-def*)

lemma *dom-maxCostList*:
  dom (*foldr maxCost* (*map AbstractMuSpaceCost xs*) []$_0$) =
   (⋂ *i* < *length xs*. *dom* (*AbstractMuSpaceCost* (*xs*! *i*)))
 apply (*induct xs*)
  apply (*simp add*: *constantCost-def*,*force*)
 apply *clarsimp*
apply (*simp add*: *maxCost-def*)
apply (*rule equalityI*)
 apply (*rule subsetI*,*simp*)
 apply (*rule ballI*,*clarsimp*)
 apply (*split split-if-asm*,*simp*)
  apply (*case-tac i*)
   apply (*simp add*: *dom-def*)
  apply *simp*
  apply (*erule-tac x*=*nat* **in** *ballE*)
   apply (*simp add*: *dom-def*)
  apply *simp*
 apply *simp*
apply *clarsimp*
apply (*rule conjI*)
 apply *force*
by *force*

lemma *P-static-dom-Δ-Case* [*rule-format*]:
  length xs > *0*
   ⟶ (∀ *i*<*length xs*.

insert (*ϱself-f f*) (*set* (*R-Args* Σ*t f*)) = *dom* (*AbstractDeltaSpaceCost* (*xs* ! *i*)))
⟶ *insert* (*ϱself-f f*) (*set* (*R-Args* Σ*t f*)) = *dom* ⨆Δ *f map AbstractDeltaSpaceCost xs*
apply (*simp add*: *maxAbstractDeltaList-def*)
apply (*induct xs*,*clarsimp*,*clarsimp*)
apply (*subst dom-maxAbstractDelta*)
apply (*case-tac xs*)
apply (*simp add*: *maxAbstractDelta-def*)
apply (*subst dom-emptyAbstractDelta*,*simp*)
apply (*subgoal-tac dom a* = *insert* (*ϱself-f f*) (*set* (*R-Args* Σ*t f*)))
prefer *2* **apply** (*erule-tac x=0* **in** *allE*,*simp*)
apply *clarsimp*
apply (*drule mp*)
apply (*rule allI*,*rule impI*)
apply (*erule-tac x=Suc i* **in** *allE*,*simp*)
by *simp*

lemma *P-dyn-dom-E1-Case-ej*:
⟦ *set* (*varsAPP* Σ*d f*) ⊆ *dom E1*;
def-extend E1 (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs* ⟧
⟹ *set* (*varsAPP* Σ*d f*) ⊆ *dom* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*)
apply (*simp add*: *extend-def*)
apply (*simp add*: *def-extend-def*)
by *blast*

lemma *P-dyn-fv-dom-E1-Case-ej* [*rule-format*]:
fvAlts alts ⊆ *dom E1*
⟶ *i* < *length alts*
⟶ *def-extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*
⟶ *fv* (*snd* (*alts* ! *i*)) ⊆ *dom* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*)
apply (*induct alts arbitrary*: *i* ,*simp-all*)
apply (*rule impI*)+
apply (*elim conjE*)
apply (*case-tac alts* = [],*simp-all*)
apply (*simp add*: *def-extend-def*)
apply (*case-tac a*, *simp-all*)
apply (*elim conjE*)
apply (*simp add*: *extend-def*)
apply *blast*
apply (*case-tac i*,*simp-all*)
apply (*case-tac a*, *simp-all*)
apply (*simp add*: *def-extend-def*)
apply (*simp add*: *extend-def*)
apply (*case-tac aa*,*simp-all*)
apply (*simp add*: *Let-def*)

by *blast*

lemma *P-dyn-fvReg-dom-E2-Case-ej*:
⟦ *fvAltsReg alts* ⊆ *dom E2*; *i* < *length alts* ⟧
⟹ *fvReg* (*snd* (*alts* ! *i*)) ⊆ *dom E2*
apply (*induct alts arbitrary*: *i*,*simp-all*)
apply (*case-tac i*,*simp*)
by (*case-tac a*, *simp-all*)

lemma *P-dyn-fv-dom-E1-Case-LitN-ej* [*rule-format*]:
fvAlts alts ⊆ *dom E1*
⟶ *i* < *length alts*
⟶ *fst* (*alts* ! *i*) = *ConstP* (*LitN n*)
⟶ *fv* (*snd* (*alts* ! *i*)) ⊆ *dom E1*
apply (*induct alts arbitrary*: *i* ,*simp-all*)
apply (*rule impI*)+
apply (*elim conjE*)
apply (*case-tac alts* = [],*simp-all*)
apply (*case-tac a*, *simp-all*)
apply (*case-tac i*,*simp-all*)
by (*case-tac a*, *simp-all*)

lemma *P-dyn-fv-dom-E1-Case-LitB-ej* [*rule-format*]:
fvAlts alts ⊆ *dom E1*
⟶ *i* < *length alts*
⟶ *fst* (*alts* ! *i*) = *ConstP* (*LitB b*)
⟶ *fv* (*snd* (*alts* ! *i*)) ⊆ *dom E1*
apply (*induct alts arbitrary*: *i* ,*simp-all*)
apply (*rule impI*)+
apply (*elim conjE*)
apply (*case-tac alts* = [],*simp-all*)
apply (*case-tac a*, *simp-all*)
apply (*case-tac i*,*simp-all*)
by (*case-tac a*, *simp-all*)

lemma *si-in-dom-AbstractDeltaSpaceCost*:
⟦ *defined-AbstractDelta* (*AbstractDeltaSpaceCost x*) *f*; *ϱ* ∈ *dom* (*AbstractDeltaSpaceCost x*) ⟧
⟹ *build-si E1 h* (*varsAPP* Σ*d f*) ∈ *dom* (*the* (*AbstractDeltaSpaceCost x ϱ*))
apply (*simp add*: *defined-AbstractDelta-def*)
apply (*erule-tac x*=*ϱ* **in** *ballE*)
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP* Σ*d f*) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)

by *simp*

lemma *si-in-dom-emptyAbstractDelta*:
$\varrho \in dom\ []_f$
$\Longrightarrow$ *build-si E1 h* (*varsAPP* $\Sigma d\ f$) $\in$ *dom* (*the* ($[]_f\ \varrho$))
apply (*simp add*: *emptyAbstractDelta-def*, *clarsimp*)
apply (*split split-if-asm*)
apply *clarsimp*
apply (*rule conjI*)
apply (*simp add*: *constantCost-def*,*force*)
apply (*rule conjI*)
apply (*simp add*: *constantCost-def*,*force*)
apply *force*
by *simp*

lemma *defined-AbstracDelta-imp-defined-bound*:
⟦ (∀ *i*<*length assert*.
insert (*ϱself-f f*) (*set* (*R-Args* $\Sigma t\ f$)) = *dom* (*AbstractDeltaSpaceCost* (*assert* ! *i*)));
$\varrho \in$ *insert* (*ϱself-f f*) (*set* (*R-Args* $\Sigma t\ f$));
(∀ *i* < *length assert*. *defined-AbstractDelta* (*AbstractDeltaSpaceCost* (*assert* ! *i*)) *f*) ⟧
$\Longrightarrow$ (∀ *i* < *length assert*. *defined-bound* (*the* (*AbstractDeltaSpaceCost* (*assert* ! *i*) *ϱ*)) *f*)
apply (*rule allI*,*rule impI*)
apply (*erule-tac x*=*i* **in** *allE*,*simp*)
apply (*simp only*: *defined-AbstractDelta-def*)
done

lemma *all-i-defined-bound-all-si-in-dom-Delta*:
(∀ *i* < *length xs*. *defined-bound* (*the* (*AbstractDeltaSpaceCost* (*xs* ! *i*) *ϱ*)) *f*)
$\Longrightarrow$ ∀ *i* < *length xs*. *build-si E1 h* (*varsAPP* $\Sigma d\ f$) $\in$ *dom* (*the* (*AbstractDeltaSpaceCost* (*xs* ! *i*) *ϱ*))
apply (*rule allI*,*rule impI*)
apply (*erule-tac x*=*i* **in** *allE*,*simp*)
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP* $\Sigma d\ f$) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*)
by *simp*

lemma *all-i-si-in-Delta-si-in-dom-maxCost-Delta*:
∀ *i* < *length xs*. *build-si E1 h* (*varsAPP* $\Sigma d\ f$) $\in$ *dom* (*the* (*AbstractDeltaSpaceCost* (*xs* ! *i*) *ϱ*))
$\Longrightarrow$ *build-si E1 h* (*varsAPP* $\Sigma d\ f$) $\in$ ($\bigcap$ *i* < *length xs*. *dom* (*the* (*AbstractDeltaSpaceCost* (*xs* ! *i*) *ϱ*)))
by (*induct xs*,*simp*,*simp*)

lemma *dom-maxCostDeltaList* [*rule-format*]:
  $\varrho \in$ *insert* (*ϱself-f f*) (*set* (*R-Args* Σt *f*))
   $\longrightarrow \varrho \in$ *dom* (*foldr maxAbstractDelta* (*map AbstractDeltaSpaceCost xs*) $[]_f$)
   $\longrightarrow$ *dom* (*the* (*foldr maxAbstractDelta* (*map AbstractDeltaSpaceCost xs*) $[]_f$ ϱ))
=
      ($\bigcap$ $i <$ *length xs*. *dom* (*the* (*AbstractDeltaSpaceCost* (*xs* ! *i*) ϱ)))
apply (*rule impI*)
apply (*induct xs*)
 apply (*simp add*: *emptyAbstractDelta-def*)
 apply (*simp add*: *constantCost-def*)
 apply *force*
apply *clarsimp*
apply (*rule equalityI*)
 apply (*rule subsetI*,*simp*)
 apply (*rule ballI*,*clarsimp*)
 apply (*case-tac i*)
  apply (*simp add*: *maxAbstractDelta-def*)
  apply (*split split-if-asm*)
   apply (*simp add*: *maxCost-def*)
   apply *clarsimp*
   apply (*split split-if-asm*)
    apply (*simp add*: *dom-def*)
   apply *simp*
  apply *simp*
 apply *clarsimp*
 apply (*simp add*: *maxAbstractDelta-def*)
 apply (*split split-if-asm*)
  apply (*simp add*: *maxCost-def*)
  apply *clarsimp*
  apply (*split split-if-asm*)
   apply (*erule-tac x*=*nat* **in** *ballE*)
    apply (*simp add*: *dom-def*)
   apply *simp*
  apply *simp*
 apply *simp*
apply (*rule subsetI*,*simp*)
apply (*simp add*: *maxAbstractDelta-def*)
apply (*split split-if-asm*)
 apply (*simp add*: *maxCost-def*)
 apply *clarsimp*
 apply (*rule conjI*)
  apply (*simp add*: *dom-def*)
  apply *force*
 apply (*rule ballI*)
 apply (*erule-tac x*=*Suc i* **in** *ballE*)
  apply *simp*
 apply *simp*
by *simp*

lemma *si-dom-Delta-si-dom-maxCost-Delta*:
⟦ ϱ ∈ *insert* (ϱ*self-f f*) (*set* (*R-Args* Σ*t f*));
ϱ ∈ *dom* (*foldr maxAbstractDelta* (*map AbstractDeltaSpaceCost xs*) []$_f$);
∀ *i* < *length xs*. *build-si E1 h* (*varsAPP* Σ*d f*) ∈ *dom* (*the* (*AbstractDeltaSpaceCost* (*xs* ! *i*) ϱ)) ⟧
⟹ *build-si E1 h* (*varsAPP* Σ*d f*) ∈ *dom* (*the* (*foldr maxAbstractDelta* (*map AbstractDeltaSpaceCost xs*) []$_f$ ϱ))
apply (*frule all-i-si-in-Delta-si-in-dom-maxCost-Delta*)
by (*subst dom-maxCostDeltaList*,*simp*,*simp*,*simp*)

lemma ϱ*-in-dom-AbstractDeltaSpaceCost*:
⟦ ϱ ∈ *insert* (ϱ*self-f f*) (*set* (*R-Args* Σ*t f*));
ϱ ∈ *dom* (*maxAbstractDelta* (*AbstractDeltaSpaceCost x*) (*foldr maxAbstract-Delta* (*map AbstractDeltaSpaceCost xs*) []$_f$)) ⟧
⟹ ϱ ∈ *dom* (*AbstractDeltaSpaceCost x*) ∧ ϱ ∈ *dom* (*foldr maxAbstractDelta* (*map AbstractDeltaSpaceCost xs*) []$_f$)
apply (*simp add*: *maxAbstractDelta-def*,*clarsimp*)
by (*split split-if-asm*,*simp*,*simp*)

lemma *si-in-dom-AbstractDeltaSpaceCost*:
⟦ ϱ ∈ *insert* (ϱ*self-f f*) (*set* (*R-Args* Σ*t f*));
si ∈ *dom* (*the* (*maxAbstractDelta* (*AbstractDeltaSpaceCost x*) (*foldr maxAb-stractDelta* (*map AbstractDeltaSpaceCost xs*) []$_f$) ϱ));
ϱ ∈ *dom* (*maxAbstractDelta* (*AbstractDeltaSpaceCost x*) (*foldr maxAbstract-Delta* (*map AbstractDeltaSpaceCost xs*) []$_f$)) ⟧
⟹ *si* ∈ *dom* (*the* (*AbstractDeltaSpaceCost x* ϱ)) ∩
dom (*the* (*foldr maxAbstractDelta* (*map AbstractDeltaSpaceCost xs*) []$_f$ ϱ))
apply (*frule* ϱ*-in-dom-AbstractDeltaSpaceCost*,*simp*,*clarsimp*)
apply (*simp add*: *maxAbstractDelta-def*)
apply (*split split-if-asm*)
apply (*simp add*: *maxCost-def*,*clarsimp*)
apply (*split split-if-asm*,*simp*,*simp*)
by *simp*

lemma *maxAbstractDeltaSpaceCost-ge-AbstractDeltaSpaceCost* [*rule-format*]:
i < *length assert*
⟶ ϱ ∈ *insert* (ϱ*self-f f*) (*set* (*R-Args* Σ*t f*))
⟶ ϱ ∈ *dom* (*foldr maxAbstractDelta* (*map AbstractDeltaSpaceCost assert*) []$_f$)
⟶ *build-si E1 h* (*varsAPP* Σ*d f*) ∈ *dom* (*the* (*foldr maxAbstractDelta* (*map AbstractDeltaSpaceCost assert*) []$_f$ ϱ))

$\longrightarrow$ *the* (*the* (*AbstractDeltaSpaceCost* (*assert* ! *i*) ϱ) (*build-si E1 h* (*varsAPP* $\Sigma d\ f$)))
$\leq$ *the* (*the* ($\bigsqcup_\Delta$ *f map AbstractDeltaSpaceCost assert* ϱ) (*build-si E1 h* (*varsAPP* $\Sigma d\ f$)))
apply (*rule impI*)
apply (*induct assert i rule*: *list-induct3*,*simp-all*)
apply (*rule conjI*)
apply (*rule impI*)+
apply (*simp only*: *maxAbstractDeltaList-def*)
apply (*simp* (*no-asm*) *add*: *maxAbstractDelta-def*)
apply (*rule conjI*)
apply (*rule impI*)
apply (*simp add*: *maxCost-def*)
apply (*rule impI*)+
apply (*subgoal-tac*
build-si E1 h (*varsAPP* $\Sigma d\ f$) $\in$
dom (*the* (*AbstractDeltaSpaceCost x* (ϱ*self-f f*))) $\cap$
dom (*the* (*foldr maxAbstractDelta* (*map AbstractDeltaSpaceCost xs*) $[]_f$ (ϱ*self-f f*))), *simp*)
apply (*rule si-in-dom-AbstractDeltaSpaceCost*,*simp*,*simp*,*simp*)
apply (*rule impI*)+
apply (*subgoal-tac* ϱ*self-f f* $\in$ *dom* (*AbstractDeltaSpaceCost x*) $\wedge$
ϱ*self-f f* $\in$ *dom* (*foldr maxAbstractDelta* (*map AbstractDeltaSpaceCost xs*) $[]_f$),*simp*)
apply (*rule* ϱ*-in-dom-AbstractDeltaSpaceCost*,*simp*,*simp*)
apply (*rule impI*)+
apply (*simp only*: *maxAbstractDeltaList-def*)
apply (*simp* (*no-asm*) *add*: *maxAbstractDelta-def*)
apply (*rule conjI*)
apply (*rule impI*)
apply (*simp add*: *maxCost-def*)
apply (*rule impI*)+
apply (*subgoal-tac build-si E1 h* (*varsAPP* $\Sigma d\ f$) $\in$ *dom* (*the* (*AbstractDeltaSpaceCost x* ϱ)) $\cap$
dom (*the* (*foldr maxAbstractDelta* (*map AbstractDeltaSpaceCost xs*) $[]_f$ ϱ)), *simp*)
apply (*rule si-in-dom-AbstractDeltaSpaceCost*,*simp*,*force*,*simp*)
apply (*rule impI*)+
apply (*subgoal-tac* ϱ $\in$ *dom* (*AbstractDeltaSpaceCost x*) $\wedge$
ϱ $\in$ *dom* (*foldr maxAbstractDelta* (*map AbstractDeltaSpaceCost xs*) $[]_f$),*simp*)
apply (*rule* ϱ*-in-dom-AbstractDeltaSpaceCost*,*simp*,*simp*)
apply (*rule conjI*)
apply (*rule impI*)+
apply *clarsimp*
apply (*simp only*: *maxAbstractDeltaList-def*)
apply (*simp* (*no-asm*) *add*: *maxAbstractDelta-def*)
apply (*rule conjI*)
apply (*rule impI*)
apply (*simp add*: *maxCost-def*)

apply (*rule conjI*)
apply (*rule impI*)+
apply *simp*
apply (*rule impI*)+
apply (*simp add*: *maxAbstractDelta-def*)
apply (*simp add*: *maxCost-def*)
apply *clarsimp*
apply (*split split-if-asm*,*simp*,*simp*)
apply (*rule impI*)+
apply (*simp add*: *maxAbstractDelta-def*)
apply (*split split-if-asm*,*simp*,*simp*)
apply *clarsimp*
apply (*simp only*: *maxAbstractDeltaList-def*)
apply (*simp* (*no-asm*) *add*: *maxAbstractDelta-def*)
apply (*rule conjI*)
apply (*rule impI*)
apply (*simp add*: *maxCost-def*)
apply (*rule conjI*)
apply (*rule impI*)+
apply *simp*
apply (*rule impI*)+
apply (*simp add*: *maxAbstractDelta-def*)
apply (*simp add*: *maxCost-def*)
apply *clarsimp*
apply (*split split-if-asm*,*simp*,*simp*)
apply (*rule impI*)+
apply (*simp add*: *maxAbstractDelta-def*)
by (*split split-if-asm*,*simp*,*simp*)

lemma *all-i-defined-bound-all-si-in-dom-mu*:
$\forall\ i < length\ xs.\ defined\text{-}bound\ (AbstractMuSpaceCost\ (xs!\ i))\ f$
$\Longrightarrow\ \forall\ i < length\ xs.\ build\text{-}si\ E1\ h\ (varsAPP\ \Sigma d\ f) \in dom\ (AbstractMuSpaceCost\ (xs!\ i))$
apply (*rule allI*,*rule impI*)
apply (*erule-tac x*=*i* **in** *allE*,*simp*)
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP* $\Sigma d\ f$) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*)
by *simp*

lemma *all-i-si-in-dom-mu-si-in-dom-maxCost-mu*:

$\forall\ i < length\ xs.\ build\text{-}si\ E1\ h\ (varsAPP\ \Sigma d\ f) \in dom\ (AbstractMuSpaceCost\ (xs!\ i))$
$\Longrightarrow build\text{-}si\ E1\ h\ (varsAPP\ \Sigma d\ f) \in (\bigcap\ i < length\ xs.\ dom\ (AbstractMuSpaceCost\ (xs!\ i)))$
by (*induct xs*,*simp*,*simp*)

lemma *si-dom-mu-si-dom-maxCost-mu*:
$\forall\ i < length\ xs.\ build\text{-}si\ E1\ h\ (varsAPP\ \Sigma d\ f) \in dom\ (AbstractMuSpaceCost\ (xs!\ i))$
$\Longrightarrow build\text{-}si\ E1\ h\ (varsAPP\ \Sigma d\ f) \in dom\ (foldr\ maxCost\ (map\ AbstractMuSpaceCost\ xs)\ []_0)$
apply (*frule all-i-si-in-dom-mu-si-in-dom-maxCost-mu*)
by (*subst dom-maxCostList*,*simp*)

lemma *maxAbstractMuSpaceCost-ge-AbstractMuSpaceCost* [*rule-format*]:
$i < length\ assert$
$\longrightarrow (\forall\ i < length\ assert.\ defined\text{-}bound\ (AbstractMuSpaceCost\ (assert\ !\ i))\ f)$
$\longrightarrow the\ (AbstractMuSpaceCost\ (assert\ !\ i)\ (build\text{-}si\ E1\ h\ (varsAPP\ \Sigma d\ f)))$
$\leq the\ (\bigsqcup_c\ map\ AbstractMuSpaceCost\ assert\ (build\text{-}si\ E1\ h\ (varsAPP\ \Sigma d\ f)))$
apply (*rule impI*)
apply (*simp add*: *maxCostList-def*)
apply (*induct assert i rule*: *list-induct3*, *simp-all*)
apply (*case-tac xs*,*simp*)
apply (*rule impI*)
apply (*simp only*: *maxCost-def*)
apply (*simp* (*no-asm*) *add*: *maxCost-def*)
apply (*simp add*: *constantCost-def*)
apply (*simp add*: *dom-def*)
apply *clarsimp*
apply (*simp add*: *maxCost-def*)
apply (*rule impI*)+
apply (*drule mp*)
apply (*erule-tac x=0* **in** *allE*,*simp*)
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x= build-si E1 h* (*varsAPP* $\Sigma d\ f$) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply *clarsimp*
apply (*rule conjI*)
apply (*erule-tac x=Suc 0* **in** *allE*,*simp*)
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x= build-si E1 h* (*varsAPP* $\Sigma d\ f$) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*rule si-dom-mu-si-dom-maxCost-mu*)
apply (*rule all-i-defined-bound-all-si-in-dom-mu*,*force*)
apply (*rule impI*,*clarsimp*)
apply (*drule mp*)

apply *force*
apply (*simp add*: *maxCost-def*)
apply (*rule conjI*)
apply (*rule impI*)+
apply *simp*
apply (*rule impI*)+
apply (*drule mp*)
apply (*erule-tac x=0* **in** *allE*,*simp*)
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x= build-si E1 h* (*varsAPP* $\Sigma d\ f$) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*subgoal-tac*
build-si E1 h (*varsAPP* $\Sigma d\ f$) $\in$ *dom* (*foldr maxCost* (*map AbstractMuSpaceCost xs*) $[]_0$),*simp*)
apply (*rule si-dom-mu-si-dom-maxCost-mu*)
apply (*rule all-i-defined-bound-all-si-in-dom-mu*)
by *force*

lemma *sizeAbstractDelta-si-case-Lit-ge-sizeAbstracDelta-si-alt*:
$[\![$ *i* < *length assert*;
insert (ϱ*self-f f*) (*set* (*R-Args* $\Sigma t\ f$)) = *dom* $\bigsqcup_\Delta$ *f map AbstractDeltaSpaceCost assert*;
($\forall$ *i*<*length assert*.
insert (ϱ*self-f f*) (*set* (*R-Args* $\Sigma t\ f$)) = *dom* (*AbstractDeltaSpaceCost* (*assert* ! *i*)));
($\forall$ *i* < *length assert*. *defined-AbstractDelta* (*AbstractDeltaSpaceCost* (*assert* ! *i*)) *f*);
insert (ϱ*self-f f*) (*set* (*R-Args* $\Sigma t\ f$)) = *dom* η $]\!]$
$\Longrightarrow$ *sizeAbstractDelta-si* (*AbstractDeltaSpaceCost* (*assert* ! *i*)) (*build-si E1 h* (*varsAPP* $\Sigma d\ f$)) *j* η
$\leq$ *sizeAbstractDelta-si* $\bigsqcup_\Delta$ *f map AbstractDeltaSpaceCost assert* (*build-si E1 h* (*varsAPP* $\Sigma d\ f$)) *j* η
apply (*simp* (*no-asm*) *only*: *sizeAbstractDelta-si-def*)
apply (*rule setsum-mono*)
apply (*subgoal-tac* $\varrho \in$ *insert* (ϱ*self-f f*) (*set* (*R-Args* $\Sigma t\ f$)))
prefer *2* **apply** (*simp add*: *dom-def*,*blast*)
apply (*rule maxAbstractDeltaSpaceCost-ge-AbstractDeltaSpaceCost*,*simp*,*simp*)
apply (*simp add*: *maxAbstractDeltaList-def*)
apply (*subst dom-maxCostDeltaList*,*simp*)
apply (*simp add*: *maxAbstractDeltaList-def*)
apply (*rule all-i-si-in-Delta-si-in-dom-maxCost-Delta*)
apply (*rule all-i-defined-bound-all-si-in-dom-Delta*)
apply (*frule defined-AbstracDelta-imp-defined-bound*,*simp*,*simp*)
by *simp*

lemma *P-Δ-Case-Lit*:
⟦ $i <$ *length assert*;
insert (*ϱself-f f*) (*set* (*R-Args* Σ*t f*)) = *dom* $\bigsqcup_{\Delta}$ *f map AbstractDeltaSpaceCost assert*;
∀ *i*<*length assert*. *insert* (*ϱself-f f*) (*set* (*R-Args* Σ*t f*)) = *dom* (*AbstractDeltaSpaceCost* (*assert* ! *i*));
∀ *i*<*length assert*. *defined-AbstractDelta* (*AbstractDeltaSpaceCost* (*assert* ! *i*)) *f*;
insert (*ϱself-f f*) (*set* (*R-Args* Σ*t f*)) = *dom* η;
Delta-ge (*AbstractDeltaSpaceCost* (*assert* ! *i*)) (*build-si E1 h* (*varsAPP* Σ*d f*)) *k* η δ ⟧
⟹ *Delta-ge* $\bigsqcup_{\Delta}$ *f map AbstractDeltaSpaceCost assert* (*build-si E1 h* (*varsAPP* Σ*d f*)) *k* η δ
apply (*simp* (*no-asm*) *only*: *Delta-ge-def*)
apply (*rule ballI*)
apply (*frule-tac*
?E1.0=*E1* **and** *h* =*h* **and** *j*=*j*
in *sizeAbstractDelta-si-case-Lit-ge-sizeAbstracDelta-si-alt*, *assumption*+)
apply (*simp add*: *Delta-ge-def*)
by (*erule-tac x*=*j* **in** *ballE*,*simp*,*simp*)

lemma *nth-build-si*:
i< *length xs*
⟹ *build-si E1 h xs*!*i* = *sizeEnv E1* (*xs*!*i*) *h*
apply (*induct xs arbitrary*:*i*, *simp-all*)
by (*case-tac i*,*simp-all*)

lemma *sizeEnv-alt-equals-sizeEnv-Case*:
⟦ *set* (*varsAPP* Σ*d f*) ⊆ *dom E1*; *ia* < *length* (*varsAPP* Σ*d f*);
def-extend E1 (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs* ⟧
⟹ *sizeEnv* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*) (*varsAPP* Σ*d f* ! *ia*) *h* = *sizeEnv E1* (*varsAPP* Σ*d f* ! *ia*) *h*
apply (*subgoal-tac*
extend E1 (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs* (*varsAPP* Σ*d f* ! *ia*) =
E1 (*varsAPP* Σ*d f* ! *ia*))
apply (*simp add*: *sizeEnv-def*)
apply (*simp add*: *def-extend-def*)
apply (*elim conjE*)
apply (*subgoal-tac* (*varsAPP* Σ*d f* ! *ia*) ∉ *set* (*snd* (*extractP* (*fst* (*alts* ! *i*)))))
apply (*rule sym*)
apply (*subst extend-monotone*,*simp*,*simp*)
apply (*subgoal-tac varsAPP* Σ*d f* ! *ia* ∈ *dom E1*,*blast*)
by (*subst* (*asm*) *set-conv-nth*,*blast*)

lemma *build-si-Case-ge-build-si-alt*:
⟦ *set* (*varsAPP* Σ*d f*) ⊆ *dom E1*; *ia* < *length* (*varsAPP* Σ*d f*);
def-extend E1 (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs* ⟧

$\Longrightarrow$ *build-si* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*) *h* (*varsAPP* Σ*d f*) ! *ia* = *build-si E1 h* (*varsAPP* Σ*d f*) ! *ia*
apply (*subst nth-build-si*,*simp*)+
by (*rule sizeEnv-alt-equals-sizeEnv-Case*,*assumption*+)

lemma *sizeEnv-alt-equals-sizeEnv-Case-2*:
⟦ *x* ∈ *dom E1*;
def-extend E1 (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs* ⟧
$\Longrightarrow$ *sizeEnv* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*) *x h* = *sizeEnv E1 x h*
apply (*subgoal-tac*
extend E1 (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs x* =
E1 x)
apply (*simp add*: *sizeEnv-def*)
apply (*simp add*: *def-extend-def*)
apply (*elim conjE*)
apply (*subgoal-tac x* ∉ *set* (*snd* (*extractP* (*fst* (*alts* ! *i*)))))
apply (*rule sym*)
apply (*subst extend-monotone*,*simp*,*simp*)
by *blast*

lemma *build-si-Case-ge-build-si-alt-2* [*rule-format*]:
set xs ⊆ *dom E1*
$\longrightarrow$ *def-extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*
$\longrightarrow$ *build-si* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*) *h xs* = *build-si E1 h xs*
apply (*induct-tac xs*,*simp-all*)
apply *clarsimp*
by (*rule sizeEnv-alt-equals-sizeEnv-Case-2*,*force*,*simp*)

lemma *list-ge-build-si-alt-build-si-Case*:
⟦ *set* (*varsAPP* Σ*d f*) ⊆ *dom E1*;
def-extend E1 (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs* ⟧
$\Longrightarrow$ *list-ge* (*build-si E1 h* (*varsAPP* Σ*d f*)) (*build-si* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*) *h* (*varsAPP* Σ*d f*))
apply (*simp add*: *list-ge-def*,*clarsimp*)
apply (*subst* (*asm*) *length-build-si*)
apply (*frule-tac ?E1.0=E1* **and** *h=h* **in** *build-si-Case-ge-build-si-alt*,*assumption*+)
by *simp*

lemma *sizeAbstractDelta-si-Case-ge-sizeAbstractDelta-si-alt*:
⟦ *i* < *length assert*; *set* (*varsAPP* Σ*d f*) ⊆ *dom E1*;
∀ *i*<*length assert*. *insert* (*ϱself-f f*) (*set* (*R-Args* Σ*t f*)) = *dom* (*AbstractDeltaSpaceCost* (*assert* ! *i*));

```
    (∀ i<length assert.
      insert (ϱself-f f) (set (R-Args Σt f)) = dom (AbstractDeltaSpaceCost (assert
! i)) );
    ∀ i<length assert. defined-AbstractDelta (AbstractDeltaSpaceCost (assert ! i))
f;
    def-extend E1 (snd (extractP (fst (alts ! i)))) vs;
    monotonic-AbstractDelta (AbstractDeltaSpaceCost (assert ! i));
    insert (ϱself-f f) (set (R-Args Σt f)) = dom η ⟧
  ⟹ sizeAbstractDelta-si (AbstractDeltaSpaceCost (assert ! i)) (build-si (extend
E1 (snd (extractP (fst (alts ! i)))) vs) h (varsAPP Σd f)) j η
  ≤ sizeAbstractDelta-si (AbstractDeltaSpaceCost (assert ! i)) (build-si E1 h (varsAPP
Σd f)) j η
apply (simp (no-asm) only: sizeAbstractDelta-si-def)
apply (rule setsum-mono,clarsimp)
by (subst build-si-Case-ge-build-si-alt-2,simp,simp,simp)


lemma sizeAbstractDelta-si-Case-equals-sizeAbstractDelta-si-alt:
  ⟦ i < length assert; set (varsAPP Σd f) ⊆ dom E1;
    def-extend E1 (snd (extractP (fst (alts ! i)))) vs ⟧
  ⟹ sizeAbstractDelta-si (AbstractDeltaSpaceCost (assert ! i)) (build-si (extend
E1 (snd (extractP (fst (alts ! i)))) vs) h (varsAPP Σd f)) j η
      = sizeAbstractDelta-si (AbstractDeltaSpaceCost (assert ! i)) (build-si E1 h
(varsAPP Σd f)) j η
apply (simp (no-asm) only: sizeAbstractDelta-si-def)
by (subst build-si-Case-ge-build-si-alt-2,simp,simp,simp)

lemma P-Δ-Case:
  ⟦ i < length assert;
   insert (ϱself-f f) (set (R-Args Σt f)) = dom ⨆Δ  f map AbstractDeltaSpaceCost
assert;
   ∀ i<length assert. insert (ϱself-f f) (set (R-Args Σt f)) = dom (AbstractDeltaSpaceCost
(assert ! i));
    ∀ i<length assert. defined-AbstractDelta (AbstractDeltaSpaceCost (assert ! i))
f;
    def-extend E1 (snd (extractP (fst (alts ! i)))) vs;
    insert (ϱself-f f) (set (R-Args Σt f)) = dom η;
   set (varsAPP Σd f) ⊆ dom E1;
   insert (ϱself-f f) (set (R-Args Σt f)) = dom η;
   monotonic-AbstractDelta (AbstractDeltaSpaceCost (assert ! i));
  Delta-ge (AbstractDeltaSpaceCost (assert ! i)) (build-si (extend E1 (snd (extractP
(fst (alts ! i)))) vs) h (varsAPP Σd f)) k η δ ⟧
  ⟹ Delta-ge ⨆Δ  f map AbstractDeltaSpaceCost assert (build-si E1 h (varsAPP
Σd f)) k η δ
apply (simp (no-asm) only: Delta-ge-def)
apply (rule ballI)
apply (frule-tac
  ?E1.0=E1 and h =h and j=j
  in sizeAbstractDelta-si-case-Lit-ge-sizeAbstracDelta-si-alt, assumption+)
```

apply (*subgoal-tac*
sizeAbstractDelta-si (*AbstractDeltaSpaceCost* (*assert* ! *i*)) (*build-si* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*) *h* (*varsAPP* Σ*d f*)) *j* η =
sizeAbstractDelta-si (*AbstractDeltaSpaceCost* (*assert* ! *i*)) (*build-si E1 h* (*varsAPP* Σ*d f*)) *j* η)
apply (*simp add*: *Delta-ge-def*)
apply (*erule-tac x*=*j* **in** *ballE*,*simp*,*simp*)
by (*rule sizeAbstractDelta-si-Case-equals-sizeAbstractDelta-si-alt*,*assumption*+)

lemma *AbstractMuSpaceCost-Case-equals-AbstractMuSpaceCost-alt*:
⟦ *set* (*varsAPP* Σ*d f*) ⊆ *dom E1*;
def-extend E1 (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs* ⟧
⟹ *the* (*AbstractMuSpaceCost* (*assert* ! *i*) (*build-si* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*) *h* (*varsAPP* Σ*d f*)))
= *the* (*AbstractMuSpaceCost* (*assert* ! *i*) (*build-si E1 h* (*varsAPP* Σ*d f*)))
by (*subst build-si-Case-ge-build-si-alt-2*,*simp*,*simp*,*simp*)

lemma *P-μ-Case*:
⟦ *set* (*varsAPP* Σ*d f*) ⊆ *dom E1*;
def-extend E1 (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*;
monotonic-bound (*AbstractMuSpaceCost* (*assert* ! *i*));
(∀ *i* < *length assert*. *defined-bound* (*AbstractMuSpaceCost* (*assert* ! *i*)) *f*);
i < *length assert*;
mu-ge (*AbstractMuSpaceCost* (*assert* ! *i*)) (*build-si* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*) *h* (*varsAPP* Σ*d f*)) *m* ⟧
⟹ *mu-ge* ⨆*c* *map AbstractMuSpaceCost assert* (*build-si E1 h* (*varsAPP* Σ*d f*)) *m*
apply (*simp add*: *mu-ge-def*)
apply (*frule-tac*
?E1.0=*E1* **and** *h* =*h*
in *maxAbstractMuSpaceCost-ge-AbstractMuSpaceCost*,*force*)
apply (*subgoal-tac*
the (*AbstractMuSpaceCost* (*assert* ! *i*) (*build-si* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*) *h* (*varsAPP* Σ*d f*))) =
the (*AbstractMuSpaceCost* (*assert* ! *i*) (*build-si E1 h* (*varsAPP* Σ*d f*))))
apply *force*
by (*rule AbstractMuSpaceCost-Case-equals-AbstractMuSpaceCost-alt*,*assumption*,*simp*)

lemma *P-μ-Case-Lit*:
⟦ (∀ *i* < *length assert*. *defined-bound* (*AbstractMuSpaceCost* (*assert* ! *i*)) *f*);
i < *length assert*;
mu-ge (*AbstractMuSpaceCost* (*assert* ! *i*)) (*build-si E1 h* (*varsAPP* Σ*d f*)) *m* ⟧
⟹ *mu-ge* ⨆*c* *map AbstractMuSpaceCost assert* (*build-si E1 h* (*varsAPP* Σ*d f*)) *m*
apply (*simp add*: *mu-ge-def*)

apply (*subgoal-tac*
  the ($\bigsqcup_C$ *map AbstractMuSpaceCost assert* (*build-si E1 h* (*varsAPP* $\Sigma d\, f$))) $\geq$
    the (*AbstractMuSpaceCost* (*assert* ! *i*) (*build-si E1 h* (*varsAPP* $\Sigma d\, f$))))
 apply *force*
by (*rule maxAbstractMuSpaceCost-ge-AbstractMuSpaceCost*,*assumption*,*simp*)

lemma *all-i-defined-bound-all-si-in-dom-sigma*:
  $\forall$ *i* < *length xs*. *defined-bound* (*AbstractSigmaSpaceCost* (*xs*! *i*)) *f*
  $\Longrightarrow$ $\forall$ *i* < *length xs*. *build-si E1 h* (*varsAPP* $\Sigma d\, f$) $\in$ *dom* (*AbstractSigmaSpaceCost* (*xs*! *i*))
apply (*rule allI*,*rule impI*)
apply (*erule-tac x*=*i* **in** *allE*,*simp*)
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x*= *build-si E1 h* (*varsAPP* $\Sigma d\, f$) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*)
by *simp*

lemma *all-i-si-in-dom-sigma-si-in-dom-maxCost-sigma*:
  $\forall$ *i* < *length xs*. *build-si E1 h* (*varsAPP* $\Sigma d\, f$) $\in$ *dom* (*AbstractSigmaSpaceCost* (*xs*! *i*))
  $\Longrightarrow$ *build-si E1 h* (*varsAPP* $\Sigma d\, f$) $\in$ ($\bigcap$ *i* < *length xs*. *dom* (*AbstractSigmaSpaceCost* (*xs*! *i*)))
by (*induct xs*,*simp*,*simp*)

lemma *dom-maxCostSigmaList*:
   length xs = *length ys*
    $\longrightarrow$ *dom* (*foldr maxCost* (*map* (λ(*x*, *y*). *addCostReal x y*) (*zip* (*map AbstractSigmaSpaceCost xs*) (*map num-r ys*))) $[]_0$) =
   ($\bigcap$ *i* < *length xs*. *dom* (*AbstractSigmaSpaceCost* (*xs*! *i*)))
 apply (*induct xs ys rule*:*list-induct2'*,*simp-all*)
 apply (*simp add*: *constantCost-def*)
 apply *force*
 apply *clarsimp*
apply (*simp add*: *maxCost-def*)
apply (*rule equalityI*)
 apply (*rule subsetI*,*simp*)
 apply (*rule ballI*,*clarsimp*)
 apply (*split split-if-asm*,*simp*)
  apply (*case-tac i*)
   apply (*simp add*: *dom-def*)
  apply (*simp add*: *addCostReal-def*)
 apply *clarsimp*
 apply (*split split-if-asm*)
 apply *clarsimp*
 apply *simp*
apply *clarsimp*
  apply (*erule-tac x*=*nat* **in** *ballE*)

apply (*simp add*: *dom-def*)
apply *simp*
apply *simp*
apply *clarsimp*
apply (*rule conjI*)
apply (*simp add*:*addCostReal-def*)
apply *force*
by *force*

lemma *si-dom-sigma-si-dom-maxCost-sigma*:
⟦ *length xs* = *length ys*;
∀ *i* < *length xs*. *build-si E1 h* (*varsAPP* Σ*d f*) ∈ *dom* (*AbstractSigmaSpaceCost* (*xs*! *i*)) ⟧
⟹ *build-si E1 h* (*varsAPP* Σ*d f*) ∈
dom (*foldr maxCost* (*map* (λ(*x*, *y*). *addCostReal x y*) (*zip* (*map AbstractSigmaSpaceCost xs*) (*map num-r ys*))) []$_0$)
apply (*frule all-i-si-in-dom-sigma-si-in-dom-maxCost-sigma*)
by (*subst dom-maxCostSigmaList*,*simp*)

lemma *maxAbstractSigmaSpaceCost-ge-AbstractSigmaSpaceCost* [*rule-format*]:
length assert = *length alts*
⟶ (∀ *i* < *length assert*. *defined-bound* (*AbstractSigmaSpaceCost* (*assert* ! *i*)) *f*)
⟶ (∀ *i* < *length assert*. *the* (*AbstractSigmaSpaceCost* (*assert* ! *i*) (*build-si E1 h* (*varsAPP* Σ*d f*)))
≤ *the* (⨆$_c$ *map* (λ(*x*, *y*). *addCostReal x y*) (*zip* (*map AbstractSigmaSpaceCost assert*) (*map num-r alts*)) (*build-si E1 h* (*varsAPP* Σ*d f*))))
apply (*induct assert alts rule*: *list-induct2'*,*simp-all*)
apply *clarsimp*
apply (*drule mp*,*force*)
apply (*simp add*: *maxCostList-def*)
apply (*simp add*: *maxCost-def*)
apply (*rule conjI*)
apply (*rule impI*,*clarsimp*)
apply (*case-tac i*,*simp*)
apply (*simp add*: *addCostReal-def*)
apply (*split split-if-asm*,*simp*)
apply (*subgoal-tac num-r* (*ab*,*ba*) >= *0*,*simp*)
apply (*simp add*: *num-r-def*)
apply *simp*
apply *simp*
apply (*erule-tac x*=*nat* **in** *allE*,*simp*)
apply (*erule-tac x*=*nat* **in** *allE*,*simp*)
apply *clarsimp*
apply (*drule mp*)
apply (*erule-tac x*=*0* **in** *allE*,*simp*)
apply (*simp add*: *defined-bound-def*)

apply (*erule-tac x= build-si E1 h* (*varsAPP* Σ*d f*) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*simp add*: *addCostReal-def*)
apply (*simp add*: *dom-def*)
apply (*subgoal-tac*
build-si E1 h (*varsAPP* Σ*d f*) ∈
dom (*foldr maxCost* (*map* (λ(*x*, *y*). *addCostReal x y*) (*zip* (*map AbstractSigmaSpaceCost xs*) (*map num-r ys*))) []$_0$),*simp*)
apply (*rule si-dom-sigma-si-dom-maxCost-sigma*,*assumption*+)
apply (*rule all-i-defined-bound-all-si-in-dom-sigma*)
by *force*

lemma *maxAbstractSigmaSpaceCost-ge-AbstractSigmaSpaceCost-2* [*rule-format*]:
length assert = *length alts*
⟶ (∀ *i* < *length assert*. *defined-bound* (*AbstractSigmaSpaceCost* (*assert* ! *i*)) *f*)
⟶ (∀ *i* < *length assert*. *the* (*AbstractSigmaSpaceCost* (*assert* ! *i*) (*build-si E1 h* (*varsAPP* Σ*d f*))) + *num-r* (*alts*!*i*)
≤ *the* (⨆$_C$ *map* (λ(*x*, *y*). *addCostReal x y*) (*zip* (*map AbstractSigmaSpaceCost assert*) (*map num-r alts*)) (*build-si E1 h* (*varsAPP* Σ*d f*))))
apply (*induct assert alts rule*: *list-induct2′*,*simp-all*)
apply *clarsimp*
apply (*drule mp*,*force*)
apply (*simp add*: *maxCostList-def*)
apply (*simp add*: *maxCost-def*)
apply (*rule conjI*)
apply (*rule impI*,*clarsimp*)
apply (*case-tac i*,*simp*)
apply (*simp add*: *addCostReal-def*)
apply (*split split-if-asm*,*simp*)
apply *simp*
apply *force*
apply *clarsimp*
apply (*drule mp*)
apply (*erule-tac x=0* **in** *allE*,*simp*)
apply (*simp add*: *defined-bound-def*)
apply (*erule-tac x= build-si E1 h* (*varsAPP* Σ*d f*) **in** *allE*)
apply (*subst* (*asm*) *length-build-si*,*simp*)
apply (*simp add*: *addCostReal-def*)
apply (*simp add*: *dom-def*)
apply (*subgoal-tac*
build-si E1 h (*varsAPP* Σ*d f*) ∈
dom (*foldr maxCost* (*map* (λ(*x*, *y*). *addCostReal x y*) (*zip* (*map AbstractSigmaSpaceCost xs*) (*map num-r ys*))) []$_0$),*simp*)
apply (*rule si-dom-sigma-si-dom-maxCost-sigma*,*assumption*+)
apply (*rule all-i-defined-bound-all-si-in-dom-sigma*)

by *force*

lemma *P-σ-Case-Lit*:
⟦ *i* < *length assert*; *length assert* = *length alts*;
(∀ *i* < *length assert*. *defined-bound* (*AbstractSigmaSpaceCost* (*assert* ! *i*)) *f*);
sigma-ge (*AbstractSigmaSpaceCost* (*assert* ! *i*)) (*build-si E1 h* (*varsAPP* Σ*d f*)) *s* ⟧
⟹ *sigma-ge* ⨆$_C$ *map* (λ(*x*, *y*). *addCostReal x y*) (*zip* (*map AbstractSigmaSpaceCost assert*) (*map num-r alts*)) (*build-si E1 h* (*varsAPP* Σ*d f*)) *s*
apply (*simp add*: *sigma-ge-def*)
apply (*subgoal-tac*
the (*AbstractSigmaSpaceCost* (*assert* ! *i*) (*build-si E1 h* (*varsAPP* Σ*d f*)))
≤ *the* (⨆$_C$ *map* (λ(*x*, *y*). *addCostReal x y*) (*zip* (*map AbstractSigmaSpaceCost assert*) (*map num-r alts*)) (*build-si E1 h* (*varsAPP* Σ*d f*))))
apply *force*
apply (*rule maxAbstractSigmaSpaceCost-ge-AbstractSigmaSpaceCost*,*assumption*+, *simp*)
by *simp*

lemma *AbstractSigmaSpaceCost-Case-equals-AbstractSigmaSpaceCost-alt*:
⟦ *set* (*varsAPP* Σ*d f*) ⊆ *dom E1*;
def-extend E1 (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs* ⟧
⟹ *the* (*AbstractSigmaSpaceCost* (*assert* ! *i*) (*build-si* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*) *h* (*varsAPP* Σ*d f*)))
= *the* (*AbstractSigmaSpaceCost* (*assert* ! *i*) (*build-si E1 h* (*varsAPP* Σ*d f*)))
by (*subst build-si-Case-ge-build-si-alt-2*,*simp*,*simp*,*simp*)

lemma *P-σ-Case*:
⟦ *i* < *length assert*; *length assert* = *length alts*;
s = *s*′ + *nr*; *nr* = *real* (*length vs*);
set (*varsAPP* Σ*d f*) ⊆ *dom E1*;
def-extend E1 (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*;
(∀ *i* < *length assert*. *defined-bound* (*AbstractSigmaSpaceCost* (*assert* ! *i*)) *f*);
sigma-ge (*AbstractSigmaSpaceCost* (*assert* ! *i*)) (*build-si* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*) *h* (*varsAPP* Σ*d f*)) *s*′ ⟧
⟹ *sigma-ge* ⨆$_C$ *map* (λ(*x*, *y*). *addCostReal x y*) (*zip* (*map AbstractSigmaSpaceCost assert*) (*map num-r alts*)) (*build-si E1 h* (*varsAPP* Σ*d f*)) *s*
apply (*simp add*: *sigma-ge-def*)
apply (*frule-tac*
?E1.0=*E1* **and** *h* =*h*
in *maxAbstractSigmaSpaceCost-ge-AbstractSigmaSpaceCost-2*,*force*,*simp*)
apply (*subgoal-tac num-r* (*alts* ! *i*) = *real* (*length vs*))
apply (*subgoal-tac*
the (*AbstractSigmaSpaceCost* (*assert* ! *i*) (*build-si* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*) *h* (*varsAPP* Σ*d f*))) =
the (*AbstractSigmaSpaceCost* (*assert* ! *i*) (*build-si E1 h* (*varsAPP* Σ*d f*))))

apply *simp*
apply (*rule AbstractSigmaSpaceCost-Case-equals-AbstractSigmaSpaceCost-alt*,*assumption*+)
apply (*simp add*: *num-r-def*)
by (*simp add*: *def-extend-def*)

lemma *P1-f-n-CASE*:
⟦ *E1 x* = *Some* (*Val.Loc p*);
SafeDepthSemanticsReal.SafeBoundSem (*E1*, *E2*) *h k td* (*Case VarE x* () *Of alts* ()) (*f*, *n*) *hh k v* (*δ*, *m*, *s*)⟧
⟹ ∃ *j C vs nr s′*.
h p = *Some* (*j*,*C*,*vs*) ∧
s = (*s′* + *nr*) ∧
nr = *real* (*length vs*) ∧
(∃ *i* < *length alts*.
def-extend E1 (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*
∧ *SafeDepthSemanticsReal.SafeBoundSem* (*extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs*, *E2*) *h k* (*td* + *nr*) (*snd* (*alts* ! *i*))
(*f*, *n*) *hh k v* (*δ*, *m*, *s′*))
apply (*simp add*: *SafeBoundSem-def*)
apply (*elim exE*, *elim conjE*)
apply (*erule SafeDepthSem.cases*,*simp-all*)
by *force*

lemma *P1-f-n-CASE-1-1*:
⟦ *E1 x* = *Some* (*IntT n′*);
SafeDepthSemanticsReal.SafeBoundSem (*E1*, *E2*) *h k td* (*Case VarE x* () *Of alts* ()) (*f*, *n*) *hh k v* (*δ*, *m*, *s*)⟧
⟹ (∃ *i* < *length alts*.
(*SafeDepthSemanticsReal.SafeBoundSem* (*E1*, *E2*) *h k td* (*snd* (*alts* ! *i*)) (*f*, *n*) *hh k v* (*δ*, *m*, *s*))
∧ *fst* (*alts* ! *i*) = *ConstP* (*LitN n′*))
apply (*simp add*: *SafeBoundSem-def*)
apply (*elim exE*, *elim conjE*)
apply (*erule SafeDepthSem.cases*,*simp-all*)
by *force*

lemma *P1-f-n-CASE-1-2*:
⟦ *E1 x* = *Some* (*BoolT b*);
SafeDepthSemanticsReal.SafeBoundSem (*E1*, *E2*) *h k td* (*Case VarE x* () *Of alts* ()) (*f*, *n*) *hh k v* (*δ*, *m*, *s*)⟧
⟹ (∃ *i* < *length alts*.
(*SafeDepthSemanticsReal.SafeBoundSem* (*E1*, *E2*) *h k td* (*snd* (*alts* ! *i*)) (*f*, *n*) *hh k v* (*δ*, *m*, *s*))
∧ *fst* (*alts* ! *i*) = *ConstP* (*LitB b*))

apply (*simp add*: *SafeBoundSem-def*)
apply (*elim exE*, *elim conjE*)
apply (*erule SafeDepthSem.cases*,*simp-all*)
by *force*

lemma *nr-Case*:
⟦ *nr* = *real* (*length vs*); *def-extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs* ⟧
⟹ *td* + *nr* = *td* + *num-r* (*alts*!*i*)
by (*simp add*: *num-r-def*, *simp add*: *def-extend-def*)

lemma *SafeResourcesDADepth-CASE*:
⟦ *length alts* = *length assert*; *length alts* > *0*;
(∀ *i* < *length assert*. *defined-AbstractDelta* (*AbstractDeltaSpaceCost* (*assert* ! *i*)) *f*);
(∀ *i* < *length assert*. *monotonic-AbstractDelta* (*AbstractDeltaSpaceCost* (*assert* ! *i*)));
(∀ *i* < *length assert*. *defined-bound* (*AbstractMuSpaceCost* (*assert* ! *i*)) *f*);
(∀ *i* < *length assert*. *monotonic-bound* (*AbstractMuSpaceCost* (*assert* ! *i*)));
(∀ *i* < *length assert*. *defined-bound* (*AbstractSigmaSpaceCost* (*assert* ! *i*)) *f*);
(∀ *i* < *length assert*. *monotonic-bound* (*AbstractSigmaSpaceCost* (*assert* ! *i*)));
∀ *i* < *length alts*.
snd (*alts* ! *i*) :$_f$ (*ϑ1*, *ϑ2*) *φ* (*td*+ *real* (*length* (*snd* (*extractP* (*fst* (*alts* ! *i*)))))), *n*
{|*AbstractDeltaSpaceCost* (*assert*!*i*),
AbstractMuSpaceCost (*assert*!*i*),
AbstractSigmaSpaceCost (*assert*!*i*)|};
Δ = $\bigsqcup_\Delta$ *f* (*map AbstractDeltaSpaceCost assert*);
μ = $\bigsqcup_c$ (*map AbstractMuSpaceCost assert*);
σ = $\bigsqcup_c$ (*map* (λ (Δ,*n*). *addCostReal* Δ *n*) (*zip* (*map AbstractSigmaSpaceCost assert*) (*map num-r alts*))) ⟧
⟹ *Case* (*VarE x a*) *Of alts a′* :$_f$ (*ϑ1*, *ϑ2*) *φ td*, *n* {|Δ,*μ*,*σ*|}
apply (*simp only*: *SafeResourcesDAssDepth.simps*)

apply (*rule impI*)
apply (*subgoal-tac* ∀ *i* < *length alts*. *valid* Σ*d* Σ*ϑ* Σ*Φ*)
prefer *2* **apply** *simp*
apply *simp*

apply (*rule conjI*)
apply (*rule P-static-dom-Δ-Case*)
apply (*simp*,*simp*)

apply (*intro allI*, *rule impI*)

apply (*elim conjE*)

apply (*case-tac E1 x*)

apply (*simp add*: *dom-def*)

apply (*case-tac a*)
apply (*rename-tac p*,*simp*)

apply (*frule P1-f-n-CASE*,*assumption*)
apply (*elim exE*, *elim conjE*)
apply (*elim exE*, *elim conjE*)
apply (*subgoal-tac*
  insert (*ϱself-f f*) (*set* (*R-Args* Σ*t f*))
  = *dom* $\bigsqcup_{\Delta}$ *f map AbstractDeltaSpaceCost assert*)
 prefer *2* **apply** (*rule P-static-dom-*Δ*-Case*,*simp*)
 apply (*rotate-tac 8*)
 apply (*erule-tac x=ia* **in** *allE*,*simp*)
apply (*subgoal-tac*
  ∀ *i*<*length assert*.
  insert (*ϱself-f f*) (*set* (*R-Args* Σ*t f*)) = *dom* (*AbstractDeltaSpaceCost* (*assert* ! *i*)))
 prefer *2* **apply** *force*
apply (*rotate-tac 8*)
apply (*erule-tac x=i* **in** *allE*)
apply (*drule mp*,*simp*)
apply (*elim conjE*)

apply (*erule-tac x=extend E1* (*snd* (*extractP* (*fst* (*alts* ! *i*)))) *vs* **in** *allE*)
apply (*erule-tac x=E2* **in** *allE*)
apply (*erule-tac x=h* **in** *allE*)
apply (*rotate-tac 30*)
apply (*erule-tac x=k* **in** *allE*)
apply (*erule-tac x=hh* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)
apply (*erule-tac x=*δ **in** *allE*)
apply (*erule-tac x=m* **in** *allE*)
apply (*erule-tac x=s'* **in** *allE*)
apply (*erule-tac x=*η **in** *allE*)
apply (*rotate-tac 19*)
apply (*drule mp*)
apply (*rule conjI*)
 apply (*subst* (*asm*) *nr-Case*,*assumption*+)+

apply (*simp add*: *num-r-def*)

apply (*rule conjI*, *rule P-dyn-dom-E1-Case-ej*,*assumption*+)
apply (*rule conjI*, *rule P-dyn-fv-dom-E1-Case-ej*,*assumption*+)
apply (*rule conjI*, *rule P-dyn-fvReg-dom-E2-Case-ej*,*assumption*+)
apply (*rule conjI*, *simp*)
apply *simp*

apply (*rule conjI*, *rule P-Δ-Case*)
apply (*simp*,*assumption*+,*force*,*assumption*+,*force*,*simp*,*simp*)

apply (*rule conjI*, *rule P-μ-Case*)
apply (*simp*,*assumption*+,*force*,*assumption*+,*force*,*simp*)

apply (*rule P-σ-Case*)
apply (*simp*,*simp*,*simp*,*force*,*simp*,*assumption*+,*simp*)

apply *simp*
apply (*frule P1-f-n-CASE-1-1*,*assumption*)
apply (*elim exE*, *elim conjE*)
apply (*subgoal-tac*
  insert (*ϱself-f f*) (*set* (*R-Args Σt f*))
  = *dom* $\bigsqcup_{\Delta}$ *f map AbstractDeltaSpaceCost assert*)
 prefer *2* **apply** (*rule P-static-dom-Δ-Case*,*simp*)
        apply (*rotate-tac 8*)
        apply (*erule-tac x=ia* **in** *allE*,*simp*)
apply (*subgoal-tac*
  ∀ *i*<*length assert*.
    insert (*ϱself-f f*) (*set* (*R-Args Σt f*)) = *dom* (*AbstractDeltaSpaceCost* (*assert*
! *i*)))
 prefer *2* **apply** *force*
apply (*rotate-tac 8*)
apply (*erule-tac x=i* **in** *allE*)
apply (*drule mp*,*simp*)
apply (*elim conjE*)

apply (*erule-tac x=E1* **in** *allE*)
apply (*erule-tac x=E2* **in** *allE*)

apply (*erule-tac* x=h **in** *allE*)
apply (*rotate-tac 27*)
apply (*erule-tac* x=k **in** *allE*)
apply (*erule-tac* x=hh **in** *allE*)
apply (*erule-tac* x=v **in** *allE*)
apply (*erule-tac* x=δ **in** *allE*)
apply (*erule-tac* x=m **in** *allE*)
apply (*erule-tac* x=s **in** *allE*)
apply (*erule-tac* x=η **in** *allE*)
apply (*rotate-tac 18*)
apply (*drule mp*)
apply (*rule conjI*,*simp*)

apply (*rule conjI*,*assumption*)
apply (*rule conjI*, *rule P-dyn-fv-dom-E1-Case-LitN-ej*,*assumption*+)
apply (*rule conjI*, *rule P-dyn-fvReg-dom-E2-Case-ej*,*assumption*+)
apply (*rule conjI*,*simp*)
apply *assumption*

apply (*rule conjI*)
apply (*rule P-*Δ*-Case-Lit*,*simp*,*assumption*+,*simp*,*simp*)

apply (*rule conjI*, *rule P-*μ*-Case-Lit*,*force*,*force*,*force*)

apply (*rule P-*σ*-Case-Lit*,*force*,*force*,*force*,*force*)

apply *simp*
apply (*frule P1-f-n-CASE-1-2*,*assumption*)
apply (*elim exE*, *elim conjE*)
apply (*subgoal-tac*
insert (*ϱself-f f*) (*set* (*R-Args* Σt *f*))
= *dom* $\bigsqcup_{\Delta}$ *f map AbstractDeltaSpaceCost assert*)
prefer *2* **apply** (*rule P-static-dom-*Δ*-Case*,*simp*)
apply (*rotate-tac 8*)
apply (*erule-tac* x=*ia* **in** *allE*,*simp*)
apply (*subgoal-tac*
$\forall\, i$<*length assert*.

insert (*ϱself-f f*) (*set* (*R-Args* Σ*t f*)) = *dom* (*AbstractDeltaSpaceCost* (*assert* ! *i*)))
prefer *2* **apply** *force*
apply (*rotate-tac 8*)
apply (*erule-tac x*=*i* **in** *allE*)
apply (*drule mp*,*simp*)
apply (*elim conjE*)

apply (*erule-tac x*=*E1* **in** *allE*)
apply (*erule-tac x*=*E2* **in** *allE*)
apply (*erule-tac x*=*h* **in** *allE*)
apply (*rotate-tac 27*)
apply (*erule-tac x*=*k* **in** *allE*)
apply (*erule-tac x*=*hh* **in** *allE*)
apply (*erule-tac x*=*v* **in** *allE*)
apply (*erule-tac x*=*δ* **in** *allE*)
apply (*erule-tac x*=*m* **in** *allE*)
apply (*erule-tac x*=*s* **in** *allE*)
apply (*erule-tac x*=*η* **in** *allE*)
apply (*rotate-tac 18*)
apply (*drule mp*)
apply (*rule conjI*,*simp*)

apply (*rule conjI*,*simp*)
apply (*rule conjI*, *rule P-dyn-fv-dom-E1-Case-LitB-ej*,*assumption*+)
apply (*rule conjI*, *rule P-dyn-fvReg-dom-E2-Case-ej*,*assumption*+)
apply (*rule conjI*, *simp*)
apply *simp*

apply (*rule conjI*)
apply (*rule P-*Δ*-Case-Lit*,*simp*,*assumption*+,*simp*,*simp*)

apply (*rule conjI*, *rule P-μ-Case-Lit*,*force*,*force*,*force*)

apply (*rule P-σ-Case-Lit*,*force*,*force*,*force*,*force*)
done

declare *atom.simps* [*simp del*]
declare *consistent.simps* [*simp del*]
declare *SafeResourcesDAssDepth.simps* [*simp del*]
declare *valid-def* [*simp del*]
declare *SafeResourcesDAss.simps* [*simp del*]
declare *argP.simps* [*simp del*]
declare *η-ef-def* [*simp del*]

axioms *equals-projection*:
projection f $\Delta' =$ *projection f* Δ
$\Longrightarrow \Delta' = \Delta$

lemma *lemma-19-aux* [*rule-format*]:
$\models\models \Sigma b$
$\longrightarrow \Sigma b\ g =$ *Some* (*projection g* $\Delta g,\mu g,\sigma g$)
$\longrightarrow \Sigma d\ g =$ *Some* (*xs*,*rs*,*eg*)
$\longrightarrow$ (*bodyAPP* $\Sigma d\ g$):$_g$ (*typesAPP* $\Sigma\vartheta\ g$) (*sizesAPP* $\Sigma\Phi\ \ g$) *real*(*length* (*varsAPP* $\Sigma d\ g$) + *length* (*regionsAPP* Σd
$\{\!|\ \Delta g,\mu g,\sigma g\ |\!\}$
apply (*rule impI*)
apply (*erule ValidGlobalResourcesEnv.induct*)
apply *simp*
apply (*rule impI*)+
apply (*case-tac g*=*f*)

apply (*simp add*: *typeResAPP-def regionsArgAPP-def typesArgAPP-def*)
apply (*elim conjE*, *frule equals-projection*,*simp*)

by (*simp add*: *typeResAPP-def regionsArgAPP-def typesArgAPP-def*)

lemma *equiv-SafeResourcesDAss-all-n-SafeResourcesDAssDepth*:
e :$_f$ (*typesAPP* $\Sigma\vartheta\ f$) (*sizesAPP* $\Sigma\Phi\ f$) *td* $\{\!|\Delta, \mu, \sigma|\!\}$
$\Longrightarrow \forall n.$ *SafeResourcesDAssDepth e f* (*typesAPP* $\Sigma\vartheta\ f$) (*sizesAPP* $\Sigma\Phi\ f$) *td n*
$\Delta\ \mu\ \sigma$
apply (*simp only*: *typesAPP-def*)
apply (*simp only*: *SafeResourcesDAss.simps*)
apply (*simp only*: *SafeResourcesDAssDepth.simps*)
apply *clarsimp*
apply (*simp only*: *SafeBoundSem-def*)
apply (*simp add*: *Let-def*)
apply (*elim exE*)
apply (*elim conjE*)
apply (*erule-tac x*=*E1* **in** *allE*)
apply (*erule-tac x*=*E2* **in** *allE*)
apply (*erule-tac x*=*h* **in** *allE*)

apply (*erule-tac x=k* **in** *allE*)
apply (*erule-tac x=hh* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)
apply (*erule-tac x=δ* **in** *allE*)
apply (*erule-tac x=m* **in** *allE*)
apply (*erule-tac x=s* **in** *allE*)
apply (*erule-tac x=η* **in** *allE*)
apply (*frule-tac td=td* **in** *eqSemDepthRA*)
apply (*drule mp*,*force*)
by *simp*

lemma *lemma-19* [*rule-format*]:
ValidGlobalResourcesEnvDepth f n Σb
⟶ *Σd g = Some* (*xs*,*rs*,*eg*)
⟶ *Σb g = Some* (*projection g Δg*,*μg*,*σg*)
⟶ *g ≠ f*
⟶ *SafeResourcesDAss eg g* (*typesAPP Σϑ g*) (*sizesAPP ΣΦ g*) (*real* (*length xs*) + *real* (*length rs*)) *Δg μg σg*
apply (*rule impI*)
apply (*erule ValidGlobalResourcesEnvDepth.induct*)

apply (*rule impI*)+
apply (*frule lemma-19-aux*,*force*,*force*)
apply (*simp add*: *bodyAPP-def varsAPP-def regionsAPP-def*)

apply (*rule impI*)+
apply (*frule lemma-19-aux*, *simp add*: *fun-upd-apply*, *force*)
apply (*subst* (*asm*) *fun-upd-apply*, *simp*)
apply (*simp add*: *bodyAPP-def varsAPP-def regionsAPP-def*)

apply (*rule impI*)+
apply (*frule lemma-19-aux*, *simp add*: *fun-upd-apply*, *force*)
apply (*subst* (*asm*) *fun-upd-apply*, *simp*)
apply (*simp add*: *bodyAPP-def varsAPP-def regionsAPP-def*)

apply (*case-tac ga=g*,*simp-all*)
apply (*rule impI*)+
apply (*simp add*: *bodyAPP-def varsAPP-def regionsAPP-def*)
apply (*elim conjE*, *frule equals-projection*,*simp*)
done

lemma *lemma-20* [*rule-format*]:
ValidGlobalResourcesEnvDepth f n Σb
⟶ *Σd f = Some (xs,rs,ef)*
⟶ *Σb f = Some (projection f Δf,μf,σf)*
⟶ *n = Suc n′*
⟶ *SafeResourcesDAssDepth ef f (typesAPP Σϑ f) (sizesAPP ΣΦ f) (real (length xs) + real (length rs)) n′ Δf μf σf*
apply (*rule impI*)
apply (*erule ValidGlobalResourcesEnvDepth.induct*)

apply (*rule impI*)+
apply (*frule lemma-19-aux*, *simp add*: *fun-upd-apply*, *force*)
apply (*frule equiv-SafeResourcesDAss-all-n-SafeResourcesDAssDepth*)
apply (*simp add*: *bodyAPP-def varsAPP-def regionsAPP-def*)

apply *simp*

apply (*rule impI*)+
apply (*simp add*: *bodyAPP-def varsAPP-def regionsAPP-def*)
apply (*elim conjE*, *frule equals-projection*,*simp*)

apply (*rule impI*)+
apply *simp*
done

lemma *P1-f-n-APP*:
⟦ (*E1*, *E2*) ⊢ *h*, *k*, *td*, *AppE f as rs′ a* ⇓(*f*,*n*) *hh* , *k* , *v*, (*δ*,*m*,*s*); *primops f = None*;
Σd f = Some (*xs*, *rs*, *ef*)⟧
⟹ ∃ *h′ δ′ s′*.
(*map-of* (*zip xs* (*map* (*atom2val E1*) *as*)), *map-of* (*zip rs* (*map* (*the* ∘ *E2*) *rs′*))(*self* ↦ *Suc k*)) ⊢
h, (*Suc k*), (*real* (*length as*) + *real* (*length rs′*)), *ef* ⇓(*f*,*n*) *h′*, (*Suc k*), *v*, (*δ′*,*m*,*s′*)
∧ *s = max*(*real* (*length xs*) + *real* (*length rs*)) ((*s′*+ *real* (*length xs*))+*real* (*length rs*) − *td*)
∧ *δ = δ′*(*k*+*1*:=*None*)
∧ *length xs = length as*

$\wedge$ *distinct xs*
$\wedge$ *length rs* = *length rs'*
$\wedge$ *distinct rs*
$\wedge$ *hh* = *h'* |` {*p*. *p* $\in$ *dom h'* & *fst* (*the* (*h' p*)) $\leq$ *k*}
$\wedge$ *dom E1* $\cap$ *set xs* = {}
$\wedge$ *n* > *0*
apply (*simp add*: *SafeBoundSem-def*)
apply (*elim exE*, *elim conjE*)
apply (*erule SafeDepthSem.cases*,*simp-all*)
apply *clarsimp*
apply (*rule-tac x*=*h'* **in** *exI*)
apply (*rule-tac x*=δ **in** *exI*)
apply (*rule-tac x*=*s* **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac x*=*nfa* **in** *exI*)
apply (*rule conjI*)
apply *force*
apply *force*
by *force*

lemma *P1-f-n-APP-2*:
⟦ (*E1*, *E2*) $\vdash$ *h* , *k* , *td*, *AppE g as rs'* *a* $\Downarrow$(*f*,*n*) *hh* , *k* , *v*, (δ,*m*,*s*); *primops g* = *None*; *f*$\neq$*g*;
Σ*d g* = *Some* (*xs*, *rs*, *eg*) ⟧
$\Longrightarrow$ $\exists$ *h' δ' s'*.
(*map-of* (*zip xs* (*map* (*atom2val E1*) *as*)), *map-of* (*zip rs* (*map* (*the* $\circ$ *E2*) *rs'*))(*self* $\mapsto$ *Suc k*)) $\vdash$
h, (*Suc k*), (*real* (*length as*) + *real* (*length rs'*)), *eg* $\Downarrow$(*f*,*n*) *h'*, (*Suc k*), *v*, (δ',*m*,*s'*)
$\wedge$ *s* = *max*(*real* (*length xs*) + *real* (*length rs*)) ((*s'*+ *real* (*length xs*))+*real* (*length rs*) − *td*)
$\wedge$ δ = δ'(*k*+*1*:=*None*)
$\wedge$ *length xs* = *length as*
$\wedge$ *distinct xs*
$\wedge$ *length rs* = *length rs'*
$\wedge$ *distinct rs*
$\wedge$ *hh* = *h'* |` {*p*. *p* $\in$ *dom h'* & *fst* (*the* (*h' p*)) $\leq$ *k*}
$\wedge$ *dom E1* $\cap$ *set xs* = {}
apply (*simp add*: *SafeBoundSem-def*)
apply (*elim exE*, *elim conjE*)
apply (*erule SafeDepthSem.cases*,*simp-all*)
apply *clarsimp*
apply (*rule-tac x*=*h'* **in** *exI*)
apply (*rule-tac x*=δ **in** *exI*)
apply (*rule-tac x*=*s* **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac x*=*nfa* **in** *exI*)
apply (*rule conjI*)

apply *force*
apply *force*
by *force*

lemma *P1-f-n-ge-0-APP*:
⟦ (*E1*, *E2*) ⊢ *h* , *k* , *td*, *AppE f as rs′ a* ⇓(*f*,*Suc n*) *hh* , *k* , *v*, (δ,*m*,*s*); *primops f* = *None*;
Σ*d f* = *Some* (*xs*, *rs*, *ef*) ⟧
⟹ ∃ *h′ δ′ s′*.
(*map-of* (*zip xs* (*map* (*atom2val E1*) *as*)), *map-of* (*zip rs* (*map* (*the* ∘ *E2*) *rs′*))(*self* ↦ *Suc k*)) ⊢
h , *Suc k* , (*real* (*length as*) + *real* (*length rs′*)), *ef* ⇓(*f*,*n*) *h′* , *Suc k* , *v*, (δ′,*m*,*s′*)
∧ *s* = *max*(*real* (*length xs*) + *real* (*length rs*)) ((*s′*+ *real* (*length xs*))+*real* (*length rs*) − *td*)
∧ δ = δ′(*k*+*1*:=*None*)
∧ *length xs* = *length as*
∧ *distinct xs*
∧ *length rs* = *length rs′*
∧ *distinct rs*
∧ *hh* = *h′* |‘ {*p*. *p* ∈ *dom h′* & *fst* (*the* (*h′ p*)) ≤ *k*}
∧ *dom E1* ∩ *set xs* = {}
apply (*simp add*: *SafeBoundSem-def*)
apply (*elim exE*, *elim conjE*)
apply (*erule SafeDepthSem.cases*,*simp-all*)
apply *clarsimp*
apply (*rule-tac x*=*h′* **in** *exI*)
apply (*rule-tac x*=δ **in** *exI*)
apply (*rule-tac x*=*s* **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac x*=*nfa* **in** *exI*)
apply (*rule conjI*)
apply *force*
apply *force*
by *force*

lemma *dom-map-of-zip*:
length xs = *length ys*
⟹ *set xs* ⊆ *dom* (*map-of* (*zip xs ys*))
apply (*induct xs ys rule*:*list-induct2′*,*simp-all*)
by *blast*

lemma *xs-in-dom-f*:
⟦ *length xs* = *length ys*;
(∀ *i* < *length ys*. *f* (*xs*!*i*) = *g* (*ys*!*i*));
set ys ⊆ *dom g* ⟧
⟹ ∀ *i* < *length xs*. (*xs*!*i*) ∈ *dom f*
by *force*

lemma *nth-in-dom-map*:
set xs ⊆ *dom m*
⟹ ∀ *i* < *length xs*. *xs*!*i* ∈ *dom m*
by *force*

lemma *set-xs-subseteq-dom-m*:
⟦ *x* ∈ *set xs*; (∀ *i*<*length xs*. *xs* ! *i* ∈ *dom m*) ⟧
⟹ *x* ∈ *dom m*
by (*subst* (*asm*) *in-set-conv-nth*,*force*)

lemma *dom-instance-f*:
dom (*instance-f f* Δ*g* φ *as rs* Φ) = (*set* (*R-Args* Σ*t f*)) ∪ {ϱ*self-f f*}
apply (*simp add*: *instance-f-def*)
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp add*: *instance-f-def*)
apply *clarsimp*
apply (*rule conjI*)
apply *clarsimp*
by *clarsimp*

lemma *dom-η-ef*:
⟦ Σ*d g* = *Some* (*xs*,*rs*,*eg*); *fvReg* (*AppE g as rs'* ()) ⊆ *dom E2*;
SafeDepthSemanticsReal.SafeBoundSem (*E1*, *E2*) *h k td* (*AppE g as rs'* ())
(*f*, *n*) *hh k v* (δ, *m*, *s*);
valid Σ*d* Σϑ ΣΦ; *admissible f* η *k*; *f* ∈ *dom* Σ*d*;
argP φ (*R-Args* Σ*t g*) (*typesRegsAPP* Σϑ *f*) *rs'*⟧
⟹ *dom* (η-*ef* η φ (*R-Args* Σ*t g*) ++ [ϱ*self-f g* ↦ *Suc k*]) = *set* (*R-Args* Σ*t g*)
∪ {ϱ*self-f g*}
apply (*simp add*: *argP.simps*)
apply (*simp add*: *valid-def*)
apply (*erule-tac x*=*f* **in** *ballE*)

prefer *2* **apply** *force*
apply (*simp only*: *typesAPP-def*)
apply (*simp add*: *valid-f.simps*)
apply (*elim conjE*)
apply (*erule-tac x=AppE g as rs′* **in** *allE*)
apply (*elim conjE*,*simp*)
apply (*erule-tac x=E1* **in** *allE*)
apply (*erule-tac x=E2* **in** *allE*)
apply (*erule-tac x=h* **in** *allE*)
apply (*rotate-tac 13*)
apply (*erule-tac x=k* **in** *allE*)
apply (*erule-tac x=td* **in** *allE*)
apply (*erule-tac x=hh* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)
apply (*erule-tac x=δ* **in** *allE*)
apply (*erule-tac x=m* **in** *allE*)
apply (*erule-tac x=s* **in** *allE*)
apply (*erule-tac x=η* **in** *allE*)
apply (*drule mp*, *simp*)
apply (*subst eqSemRABound*,*force*)
apply (*elim conjE*)
apply (*simp add*: *consistent.simps*)
apply (*elim conjE*)
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp add*: *η-ef-def*)
apply *force*
apply (*rule subsetI*)
apply (*simp add*: *η-ef-def*,*clarsimp*)
apply (*frule xs-in-dom-f*,*simp*,*simp*)
apply (*frule set-xs-subseteq-dom-m*, *simp*)
apply (*subst* (*asm*) *in-set-conv-nth*)
apply (*elim exE*)
apply (*rotate-tac 5*)
apply (*erule-tac x=i* **in** *allE*)
apply (*drule mp*,*simp*,*simp*)
apply (*rotate-tac 7*)
apply (*erule-tac x=rs′!i* **in** *ballE*)
apply (*elim exE*, *elim conjE*)+
apply *simp*
by (*frule nth-in-dom-map*,*simp*)

lemma *P-dyn-xs-subseteq-dom-E1-g*:

⟦ $\Sigma d\ g$ = *Some* (*xs*, *rs*, *eg*); *fv eg* ⊆ *set xs*;
∀ *i*<*length as*. *atom* (*as* ! *i*);*length xs* = *length as* ⟧
⟹ *set* (*varsAPP* $\Sigma d\ g$) ∪ *fv eg* ⊆ *dom* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*)))
apply (*subgoal-tac length xs* = *length* (*map* (*atom2val E1*) *as*))
prefer *2* **apply** (*induct xs*, *simp*,*clarsimp*)
apply (*frule-tac ys*=(*map* (*atom2val E1*) *as*) **in** *dom-map-of-zip*)
by (*simp add*: *varsAPP-def*)

lemma *P-dyn-fvReg-eg-subseteq-dom-E2-g*:
⟦ *fvReg eg* ⊆ *set rs*; *length rs* = *length rs'*;
set rs' ⊆ *dom E2* ⟧
⟹ *fvReg eg* ⊆ *dom* (*map-of* (*zip rs* (*map* (*the* ∘ *E2*) *rs'*))(*self* ↦ *Suc k*))
apply (*subgoal-tac set rs* ⊆ *dom* (*map-of* (*zip rs* (*map* (*the* ∘ *E2*) *rs'*))),*simp*)
apply *blast*
by (*rule dom-map-of-zip*,*simp*)

lemma *P-dyn-dom-η-ef*:
⟦ $\Sigma d\ g$ = *Some* (*xs*,*rs*,*eg*); *fvReg* (*AppE g as rs'* ()) ⊆ *dom E2*;
SafeDepthSemanticsReal.SafeBoundSem (*E1*, *E2*) *h k td* (*AppE g as rs'* ()) (*f*, *n*) *hh k v* (δ, *m*, *s*);
valid $\Sigma d\ \Sigma\vartheta\ \Sigma\Phi$; *argP* ψ (*R-Args* $\Sigma t\ g$) (*typesRegsAPP* $\Sigma\vartheta\ f$) *rs'*;
admissible f η *k*; *f* ∈ *dom* Σd;
set (*R-Args* $\Sigma t\ g$) ∪ {ϱ*self-f g*} = *dom* Δg;
dom (*instance-f f g* Δg ψ (*sizesAPP* $\Sigma\Phi\ f$) *as*) = *dom* η ⟧
⟹ *dom* Δg = *dom* (η-*ef* η ψ (*R-Args* $\Sigma t\ g$) ++ [ϱ*self-f g* ↦ *Suc k*])
by (*subst dom-η-ef*,*assumption*+,*simp*)

lemma *P-η-ef*:
⟦ *admissible f* η *k* ⟧
⟹ *admissible g* (η-*ef* η φ (*R-Args* $\Sigma t\ g$) ++ [ϱ*self-f g* ↦ *Suc k*]) (*Suc k*)
apply (*simp only*: *admissible-def*)
apply (*rule conjI*,*simp*)
apply (*elim conjE*)
apply (*rule ballI*,*simp*)
apply (*rule impI*,*simp*)
apply (*erule-tac x*=*the* (φ ϱ) **in** *ballE*)

apply (*elim exE*, *elim conjE*)
apply (*simp add*: *η-ef-def*)
apply *force*
by (*simp add*: *η-ef-def*, *simp add*: *dom-def*)

lemma *length-phiMapping-equals-length-as*:
⟦ *atom2var ‘ set as* ⊆ *dom* Φ ⟧
⟹ *length* (*phiMapping-app* (*map* Φ (*map atom2var as*)) *si*) = *length as*
by (*induct as*,*simp*,*clarsimp*)

lemma *length-build-si*:
length (*build-si E1 h xs*) = *length xs*
by (*induct xs*,*simp-all*)

lemma *valid-fv-APP-subseteq-*Φ:
⟦ *SafeDepthSemanticsReal.SafeBoundSem* (*E1*, *E2*) *h k td* (*AppE g as rs′* ()) (*f*, *n*) *hh k v* (*δ*, *m*, *s*);
valid Σ*d* Σ*ϑ* ΣΦ; *f* ∈ *dom* Σ*d* ⟧
⟹ *fv* (*AppE g as rs′* ()) ⊆ *dom* (*sizesAPP* ΣΦ *f*)
apply (*simp add*: *valid-def*)
apply (*erule-tac x=f* **in** *ballE*)
apply (*simp add*: *typesAPP-def add*: *valid-f.simps*)
apply (*elim conjE*)
apply (*erule-tac x=AppE g as rs′* **in** *allE*,*simp*)
by *force*

lemma *as-subseteq-dom-*Φ [*rule-format*]:
fvs′ as ⊆ *dom* Φ
⟶ (∀ *i*<*length as*. *atom* (*as* ! *i*))
⟶ *atom2var ‘ set as* ⊆ *dom* Φ
apply (*rule impI*)
apply (*induct as*,*simp*,*clarsimp*)
apply (*drule mp*,*force*,*simp*)
apply (*erule-tac x=0* **in** *allE*,*simp*)
apply (*simp add*: *atom.simps*)
by (*case-tac a*, *simp-all*)

lemma *nth-map-of-xs-atom2val*:
⟦ *length xs* = *length as*;
distinct xs ⟧
⟹ ∀ *i* < *length xs*.
map-of (*zip xs* (*map* (*atom2val E1*) *as*)) (*xs*!*i*) =
Some (*atom2val E1* (*as*!*i*))
apply *clarsimp*
apply (*induct xs as rule*: *list-induct2′*,*simp-all*)

by (*case-tac i*,*simp*,*clarsimp*)

lemma *nth-atom2val-in-dom-E*:
⟦ *atom2var ' set as* ⊆ *dom E1*;
i < *length as*; *as* ! *i* = *VarE x a* ⟧
⟹ *Some* (*atom2val E1* (*as* ! *i*)) = *E1 x*
apply (*subgoal-tac i* < *length* (*map atom2var as*))
apply (*frule-tac xs*=(*map atom2var as*) **in** *nth-mem*,*simp*)
apply (*subgoal-tac x* ∈ *dom E1*)
apply (*frule domD*,*elim exE*,*simp*)
apply *blast*
by (*induct as*, *simp*, *simp*)

lemma *nth-build-si*:
i< *length xs*
⟹ *build-si E1 h xs*!*i* = *sizeEnv E1* (*xs*!*i*) *h*
apply (*induct xs arbitrary*:*i*, *simp-all*)
by (*case-tac i*,*simp-all*)

lemma *sizeEnv-equals-build-si-i*:
⟦*length xs* = *length as*; *distinct xs*; (∀ *i*<*length as*. *atom* (*as* ! *i*));
Σ*d g* = *Some* (*xs*,*rs*,*eg*); *i* < *length as*; *atom2var ' set as* ⊆ *dom E1*⟧
⟹ *sizeEnv E1* (*map atom2var as* ! *i*) *h*
= *build-si* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) *h* (*varsAPP* Σ*d g*) ! *i*
apply (*simp add*: *varsAPP-def*)
apply (*subst nth-build-si*,*simp*)
apply (*frule-tac ?E1.0*=*E1* **in** *nth-map-of-xs-atom2val*,*simp*)
apply (*erule-tac x*=*i* **in** *allE*)
apply (*drule mp*,*simp*)
apply (*erule-tac x*=*i* **in** *allE*)
apply (*drule mp*,*simp*)
apply (*simp add*: *atom.simps*)
apply (*case-tac as*!*i*,*simp-all*)
apply (*frule-tac ?E1.0*=*E1* **in** *nth-map-of-xs-atom2val*,*simp*)
apply (*erule-tac x*=*i* **in** *allE*)
apply (*drule mp*,*simp*)
apply (*subst* (*asm*) *nth-atom2val-in-dom-E*,*assumption*+)
apply (*subgoal-tac sizeEnv* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) (*xs* ! *i*) *h*
=
sizeEnv E1 list h,*simp*)
by (*simp add*: *sizeEnv-def*)

lemma *fv-as-in-dom-E1* [*rule-format*]:
fvs' as ⊆ *dom E1*
⟶ (∀ *i*<*length as*. *atom* (*as* ! *i*))

$\longrightarrow$ *atom2var ‘ set as* $\subseteq$ *dom E1*
apply (*rule impI*)
apply (*induct as*,*simp*,*clarsimp*)
apply (*drule mp*,*force*,*simp*)
apply (*erule-tac x=0* **in** *allE*,*simp*)
apply (*simp add*: *atom.simps*)
by (*case-tac a*, *simp-all*)

lemma *nth-phiMapping-app*:
set ys $\subseteq$ *dom* φ
$\Longrightarrow \forall$ *i* $<$ *length ys*. *phiMapping-app* (*map* φ *ys*) *si* ! *i* =
the (*the* (φ (*ys*!*i*)) *si*)
apply (*rule allI*, *rule impI*)
apply (*induct ys arbitrary*: *i*, *simp-all*)
apply (*elim conjE*)
apply (*frule domD*, *elim exE*,*simp*)
by (*case-tac i*,*simp*,*simp*)

lemma *list-ge-phiMapping-app-build-si*:
⟦ (∀ *i* $<$ *length as*. *atom* (*as*!*i*)); *length xs* = *length as*; *distinct xs*;
fv (*AppE g as rs′ a*) $\subseteq$ *dom E1*;
SafeDepthSemanticsReal.SafeBoundSem (*E1*, *E2*) *h k td* (*AppE g as rs′* ()) (*f*, *n*) *hh k v* (δ, *m*, *s*);
admissible f η *k*; *f* ∈ *dom* Σ*d*;
Σ*d g* = *Some* (*xs*,*rs*,*eg*); *valid* Σ*d* Σϑ ΣΦ ⟧
$\Longrightarrow$ *list-ge* (*phiMapping-app* (*map* (*sizesAPP* ΣΦ *f*) (*map atom2var as*)) (*build-si E1 h* (*varsAPP* Σ*d f*)))
(*build-si* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) *h* (*varsAPP* Σ*d g*))
apply (*simp add*: *list-ge-def*)
apply (*rule allI*, *rule impI*)
apply (*simp only*: *valid-def*)
apply (*erule-tac x=f* **in** *ballE*)
prefer *2* **apply** *force*
apply (*simp add*: *typesAPP-def*)
apply (*simp add*: *valid-f.simps*)
apply (*elim conjE*)
apply (*erule-tac x=AppE g as rs′* **in** *allE*)
apply (*elim conjE*,*simp*)
apply (*erule-tac x=E1* **in** *allE*)
apply (*erule-tac x=E2* **in** *allE*)
apply (*erule-tac x=h* **in** *allE*)
apply (*rotate-tac 15*)
apply (*erule-tac x=k* **in** *allE*)
apply (*erule-tac x=td* **in** *allE*)
apply (*erule-tac x=hh* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)

apply (*erule-tac x=δ* **in** *allE*)
apply (*erule-tac x=m* **in** *allE*)
apply (*erule-tac x=s* **in** *allE*)
apply (*rotate-tac 15*)
apply (*erule-tac x=η* **in** *allE*)
apply (*drule mp,simp*)
apply (*subst eqSemRABound,force*)
apply (*elim conjE*)
apply (*frule-tac* Φ=(*sizesAPP* ΣΦ *f*) **in** *as-subseteq-dom-Φ,simp*)
apply (*frule-tac* Φ=(*sizesAPP* ΣΦ *f*) **and**
si=(*build-si E1 h* (*varsAPP* Σ*d f*)) **in** *length-phiMapping-equals-length-as*)
apply (*subst nth-phiMapping-app,simp,simp*)
apply (*erule-tac x=map atom2var as* ! *i* **in** *ballE*)
apply (*subgoal-tac sizeEnv E1* (*map atom2var as* ! *i*) *h*
= *build-si* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) *h* (*varsAPP* Σ*d g*) ! *i,simp*)
apply (*frule fv-as-in-dom-E1,simp*)
apply (*rule sizeEnv-equals-build-si-i,assumption+,simp,simp*)
apply (*subgoal-tac set* (*map atom2var as*) ⊆ *dom* (*sizesAPP* ΣΦ *f*))
apply (*frule-tac xs=map atom2var as* **in** *nth-in-dom-map,simp*)
by *simp*

lemma *Phi-Mapping-app-ge-build-si-ef*:
⟦ *SafeDepthSemanticsReal.SafeBoundSem* (*E1*, *E2*) *h k td* (*AppE g as rs′* ()) (*f*, *n*) *hh k v* (δ, *m*, *s*);
(∀ *i*<*length as*. *atom* (*as* ! *i*)); *length xs* = *length as*; *distinct xs*;
valid Σ*d* Σϑ ΣΦ; Σ*d g* = *Some* (*xs*, *rs*, *ef*); *fv* (*AppE g as rs′ a*) ⊆ *dom E1*;
monotonic-bound μ*g*; *f* ∈ *dom* Σ*d*; *admissible f η k*;
defined-bound μ*g g*⟧
⟹ *the* (μ*g* (*phiMapping-app* (*map* (*sizesAPP* ΣΦ *f*) (*map atom2var as*)) (*build-si E1 h* (*varsAPP* Σ*d f*)))) ≥
the (μ*g* (*build-si* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) *h* (*varsAPP* Σ*d g*)))
apply (*frule valid-fv-APP-subseteq-Φ,assumption+*)
apply (*subgoal-tac phiMapping-app* (*map* (*sizesAPP* ΣΦ *f*) (*map atom2var as*)) (*build-si E1 h* (*varsAPP* Σ*d f*)) ∈ *dom* μ*g*)
prefer *2* **apply** (*simp only*: *defined-bound-def*)
apply (*erule-tac x=phiMapping-app* (*map* (*sizesAPP* ΣΦ *f*) (*map atom2var as*)) (*build-si E1 h* (*varsAPP* Σ*d f*)) **in** *allE*)
apply (*drule mp*)
apply (*simp add*: *varsAPP-def*)
apply (*rule length-phiMapping-equals-length-as*)
apply (*rule as-subseteq-dom-Φ*, *assumption+,force*)
apply *simp*
apply (*subgoal-tac build-si* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) *h* (*varsAPP* Σ*d g*) ∈ *dom* μ*g*)
prefer *2* **apply** (*simp only*: *defined-bound-def*)
apply (*erule-tac x=build-si* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) *h* (*varsAPP*

$\Sigma d\ g$) **in** *allE*)
apply (*drule mp*)
apply (*subst length-build-si*,*simp*)
apply *simp*
apply (*simp add*: *monotonic-bound-def*)
apply (*erule-tac x=phiMapping-app* (*map* (*sizesAPP* $\Sigma\Phi$ *f*) (*map atom2var as*)) (*build-si E1 h* (*varsAPP* Σd *f*)) **in** *ballE*)
prefer *2* **apply** *simp*
apply (*erule-tac x=build-si* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) *h* (*varsAPP* Σd *g*) **in** *ballE*)
prefer *2* **apply** *simp*
apply (*drule mp*)
by (*rule list-ge-phiMapping-app-build-si*,*assumption+*,*simp*,*assumption+*)

lemma *P-μ-APP*:
⟦ *SafeDepthSemanticsReal.SafeBoundSem* (*E1*, *E2*) *h k td* (*AppE g as rs'* ()) (*f*, *n*) *hh k v* (δ, *m*, *s*);
(∀ *i* < *length as*. *atom* (*as*!*i*));
distinct xs; *length xs* = *length as*;
monotonic-bound μg; *defined-bound* μg *g*;
Σd *g* = *Some* (*xs*, *rs*, *ef*);
f ∈ *dom* Σd;
admissible f η *k*;
valid Σd $\Sigma\vartheta$ $\Sigma\Phi$;
set (*varsAPP* Σd *f*) ∪ *fv* (*AppE g as rs'* ()) ⊆ *dom E1*;
mu-ge μg (*build-si* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) *h* (*varsAPP* Σd *g*)) *m* ⟧
⟹ *mu-ge* (*mu-app* μg *as* (*sizesAPP* $\Sigma\Phi$ *f*)) (*build-si E1 h* (*varsAPP* Σd *f*)) *m*
apply (*simp only*: *mu-ge-def*)
apply (*simp only*: *mu-app-def*)
apply (*simp only*: *cost-PhiMapping-def*)
apply (*subgoal-tac the* (μg (*phiMapping-app* (*map* (*sizesAPP* $\Sigma\Phi$ *f*) (*map atom2var as*)) (*build-si E1 h* (*varsAPP* Σd *f*)))) >=
the (μg (*build-si* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) *h* (*varsAPP* Σd *g*))),*simp*)
apply (*rule Phi-Mapping-app-ge-build-si-ef*)
by (*assumption+*,*simp*,*assumption+*)

lemma *fvs-in-dom-sizesAPP*:
⟦ (∀ *i*<*length as*. *atom* (*as* ! *i*));
SafeDepthSemanticsReal.SafeBoundSem (*E1*, *E2*) *h k td* (*AppE g as rs'* ()) (*f*, *n*) *hh k v* (δ, *m*,*s*);
f ∈ *dom* Σd; *admissible f* η *k*;
valid Σd $\Sigma\vartheta$ $\Sigma\Phi$ ⟧
⟹ *atom2var* ‘ *set as* ⊆ *dom* (*sizesAPP* $\Sigma\Phi$ *f*)
apply (*simp only*: *valid-def*)

apply (*erule-tac x=f* **in** *ballE*)
prefer *2* **apply** *force*
apply (*simp add*: *typesAPP-def*)
apply (*simp add*: *valid-f.simps*)
apply (*elim conjE*)
apply (*erule-tac x=AppE g as rs′* **in** *allE*)
apply (*elim conjE*,*simp*)
apply (*erule-tac x=E1* **in** *allE*)
apply (*erule-tac x=E2* **in** *allE*)
apply (*erule-tac x=h* **in** *allE*)
apply (*rotate-tac 10*)
apply (*erule-tac x=k* **in** *allE*)
apply (*erule-tac x=td* **in** *allE*)
apply (*erule-tac x=hh* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)
apply (*erule-tac x=δ* **in** *allE*)
apply (*erule-tac x=m* **in** *allE*)
apply (*erule-tac x=s* **in** *allE*)
apply (*erule-tac x=η* **in** *allE*)
apply (*drule mp*,*simp*)
apply (*subst eqSemRABound*,*force*)
apply (*elim conjE*)
by (*rule as-subseteq-dom-Φ*,*simp*,*simp*)

lemma *P-σ-APP*:
⟦ *SafeDepthSemanticsReal.SafeBoundSem* (*E1*, *E2*) *h k td* (*AppE g as rs′* ()) (*f*, *n*) *hh k v* (*δ*, *m*, *s*);
(∀ *i*<*length as*. *atom* (*as* ! *i*)); *length xs* = *length as*; *distinct xs*;
valid Σ*d* Σ*ϑ* ΣΦ; Σ*d g* = *Some* (*xs*, *rs*, *ef*); *fv* (*AppE g as rs′ a*) ⊆ *dom E1*;
monotonic-bound σg; *f* ∈ *dom* Σ*d*; *admissible f η k*;
defined-bound σg g;
s = *max* (*l* + *q*) (*s′* + *l* + *q* − *td*);
sigma-ge σg (*build-si* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) *h* (*varsAPP* Σ*d g*)) *s′* ⟧
⟹ *sigma-ge* ⨆*c* {[]*l* + *q*, *substCostReal* (*addCostReal* (*addCostReal* (*sigma-app σg as* (*sizesAPP* ΣΦ *f*)) *l*) *q*) *td*} (*build-si E1 h* (*varsAPP* Σ*d f*)) *s*
apply (*frule fvs-in-dom-sizesAPP*, *assumption*+)
apply (*simp only*: *sigma-ge-def*)
apply (*simp only*: *sigma-app-def*)
apply (*simp only*: *cost-PhiMapping-def*)
apply (*subgoal-tac*
the (*σg* (*phiMapping-app* (*map* (*sizesAPP* ΣΦ *f*) (*map atom2var as*)) (*build-si E1 h* (*varsAPP* Σ*d f*)))) ≥
the (*σg* (*build-si* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) *h* (*varsAPP* Σ*d g*))),*simp*)
apply (*rule conjI*)
apply (*simp add*: *constantCost-def*)
apply (*simp add*: *maxCost-def*)

apply (*rule impI*)+
apply (*drule mp*,*force*)
apply (*subgoal-tac*
build-si E1 h (*varsAPP* Σ*d f*) ∈
dom (*substCostReal* (*addCostReal* (*addCostReal* (λ*xs*. σ*g* (*phiMapping-app* (*map* (*sizesAPP* ΣΦ *f*) (*map atom2var as*)) *xs*)) *l*) *q*) *td*),*simp*)
apply (*simp add*: *substCostReal-def*)
apply (*simp add*: *addCostReal-def*)
apply *clarsimp*
apply (*subgoal-tac phiMapping-app* (*map* (*sizesAPP* ΣΦ *f*) (*map atom2var as*)) (*build-si E1 h* (*varsAPP* Σ*d f*)) ∈ *dom* σ*g*)
prefer *2* **apply** (*simp only*: *defined-bound-def*)
apply (*erule-tac x*=*phiMapping-app* (*map* (*sizesAPP* ΣΦ *f*) (*map atom2var as*)) (*build-si E1 h* (*varsAPP* Σ*d f*)) **in** *allE*)
apply (*drule mp*)
apply (*subst length-phiMapping-equals-length-as*,*simp*)
apply (*simp add*: *varsAPP-def*)
apply *simp*
apply (*subgoal-tac phiMapping-app* (*map* (*sizesAPP* ΣΦ *f*) (*map atom2var as*)) (*build-si E1 h* (*varsAPP* Σ*d f*)) ∈ *dom* σ*g*)
prefer *2* **apply** (*simp only*: *defined-bound-def*)
apply (*simp add*: *dom-def*)
apply (*simp add*: *maxCost-def*)
apply (*rule conjI*)
apply (*rule impI*)+
apply *clarsimp*
apply (*simp add*: *substCostReal-def*)
apply (*split split-if-asm*,*simp*,*simp add*: *addCostReal-def*)
apply *clarsimp*
apply (*split split-if-asm*,*simp*,*clarsimp*)
apply (*split split-if-asm*,*simp*,*simp*)
apply *simp*
apply *simp*
apply (*rule impI*)+
apply (*drule mp*)
apply (*simp add*: *constantCost-def*)
apply (*simp add*: *dom-def*)
apply (*subgoal-tac*
build-si E1 h (*varsAPP* Σ*d f*) ∈
dom (*substCostReal* (*addCostReal* (*addCostReal* (λ*xs*. σ*g* (*phiMapping-app* (*map* (*sizesAPP* ΣΦ *f*) (*map atom2var as*)) *xs*)) *l*) *q*) *td*),*simp*)
apply (*simp add*: *substCostReal-def*)
apply (*simp add*: *addCostReal-def*)
apply *clarsimp*
apply (*subgoal-tac phiMapping-app* (*map* (*sizesAPP* ΣΦ *f*) (*map atom2var as*)) (*build-si E1 h* (*varsAPP* Σ*d f*)) ∈ *dom* σ*g*)
prefer *2* **apply** (*simp only*: *defined-bound-def*)
apply (*erule-tac x*=*phiMapping-app* (*map* (*sizesAPP* ΣΦ *f*) (*map atom2var as*)) (*build-si E1 h* (*varsAPP* Σ*d f*)) **in** *allE*)

apply (*drule mp*)
apply (*subst length-phiMapping-equals-length-as*,*simp*)
apply (*simp add*: *varsAPP-def*)
apply *simp*
apply (*subgoal-tac phiMapping-app* (*map* (*sizesAPP* $\Sigma\Phi$ *f*) (*map atom2var as*))
(*build-si E1 h* (*varsAPP* Σd *f*)) $\in$ *dom* σg)
prefer *2* **apply** (*simp only*: *defined-bound-def*)
apply (*simp add*: *dom-def*)
by (*rule Phi-Mapping-app-ge-build-si-ef*,*assumption*+)

lemma *setsumCost-empty*: *setsumCost f* {} = $[]_0$
by (*simp add*: *setsumCost-def*)

lemma *sumcost-conm*:
$x +_c y = y +_c x$
apply (*simp add*: *addCost-def*)
apply (*rule ext*)
apply *simp*
done

lemma *sumcost-dist*:
addCost (*addCost x y*) *z* = *addCost x* (*addCost y z*)
apply (*simp add*: *addCost-def*)
apply (*rule ext*,*clarsimp*)
apply (*rule conjI*, *rule impI*)
apply (*rule conjI*,*clarsimp*)
apply (*split split-if-asm*,*simp add*:*dom-def*,*simp*)
apply *clarsimp*
apply (*split split-if-asm*,*simp add*:*dom-def*,*simp*)
apply (*rule impI*)
apply (*rule conjI*,*clarsimp*)
apply (*split split-if-asm*,*simp add*:*dom-def*,*simp*)
apply (*rule impI*,*clarsimp*)
by (*split split-if-asm*,*simp add*:*dom-def*,*simp*)

lemma *setsumCost-insert* [*simp*]:
finite F ==> *a* $\notin$ *F* ==> *setsumCost f* (*insert a F*) = *addCost* (*f a*)
(*setsumCost f F*)
apply (*simp add*: *setsumCost-def*)
apply (*subst ACf.fold-insert*,*simp-all*)
apply (*rule ACf.intro*)
apply (*rule sumcost-conm*)
apply (*rule sumcost-dist*)
done

axioms *setsumCost-setsum*:
the ((*setsumCost* (λ*x*. *the* (*f x*)) *A*) *xs*) = *setsum* (λ*x*. *the* (*the* (*f x*) *xs*)) *A*

lemma *sumset-instance-f-equals-sumset-phiMapping*:
dom (*instance-f f g* Δ*g* ψ (*sizesAPP* ΣΦ *f*) *as*) = *dom* η
⟹
(∑ ϱ | η ϱ = *Some j*.
the (*the* (*instance-f f g* Δ*g* ψ (*sizesAPP* ΣΦ *f*) *as* ϱ) (*build-si E1 h* (*varsAPP* Σ*d f*))))
= (∑ ϱ | η ϱ = *Some j*.
∑ ϱ∈{ϱ′. ψ ϱ′ = *Some* ϱ ∧ ϱ′ ∈ *set* (*R-Args* Σ*t g*)}.
the (*the* (Δ*g* ϱ) (*phiMapping-app* (*map* (*sizesAPP* ΣΦ *f*) (*map atom2var as*)) (*build-si E1 h* (*varsAPP* Σ*d f*)))))
apply (*simp* (*no-asm*) *only*: *instance-f-def*)
apply (*subgoal-tac*
(∑ ϱ | η ϱ = *Some j*.
the (*the* (((λϱ. *Some* (*sum-rho g* Δ*g* ψ (*sizesAPP* ΣΦ *f*) *as* ϱ)) |‘ (*set* (*R-Args* Σ*t f*) ∪ {ϱ*self-f f*})) ϱ)
(*build-si E1 h* (*varsAPP* Σ*d f*)))) =
(∑ ϱ | η ϱ = *Some j*. *the* (*the* (*Some* (*sum-rho g* Δ*g* ψ (*sizesAPP* ΣΦ *f*) *as* ϱ)) (*build-si E1 h* (*varsAPP* Σ*d f*))),*simp*)
apply (*simp add*: *sum-rho-def*)
apply (*subst setsumCost-setsum*,*simp*)
apply (*rule setsum-cong2*,*simp*)
apply (*simp only*: *sum-rho-def*)
apply (*simp add*: *restrict-map-def*)
apply (*simp add*: *instance-f-def*)
by *force*

lemma *setsum-build-si-le-setsum-phiMapping*:
⟦ *SafeDepthSemanticsReal.SafeBoundSem* (*E1*, *E2*) *h k td* (*AppE g as rs′* ()) (*f*, *n*) *hh k v* (δ, *m*, *s*);
(∀ *i*<*length as*. *atom* (*as* ! *i*)); *length xs* = *length as*; *distinct xs*;
valid Σ*d* Σϑ ΣΦ; Σ*d g* = *Some* (*xs*, *rs*, *ef*); *fv* (*AppE g as rs′ a*) ⊆ *dom E1*;
monotonic-AbstractDelta Δ*g*; *f* ∈ *dom* Σ*d*; *admissible f* η *k*;
set (*R-Args* Σ*t g*) ∪ {ϱ*self-f g*} = *dom* Δ*g*;
dom (*instance-f f g* Δ*g* ψ (*sizesAPP* ΣΦ *f*) *as*) = *dom* η;
defined-AbstractDelta Δ*g g*⟧
⟹ (∑ ϱ | η ϱ = *Some j*.
∑ ϱ∈{ϱ′. ψ ϱ′ = *Some* ϱ ∧ ϱ′ ∈ *set* (*R-Args* Σ*t g*)}.
the (*the* (Δ*g* ϱ) (*phiMapping-app* (*map* (*sizesAPP* ΣΦ *f*) (*map atom2var as*)) (*build-si E1 h* (*varsAPP* Σ*d f*)))))
≥
(∑ ϱ | η ϱ = *Some j*.
∑ ϱ∈{ϱ′. ψ ϱ′ = *Some* ϱ ∧ ϱ′ ∈ *set* (*R-Args* Σ*t g*)}.
the (*the* (Δ*g* ϱ) (*build-si* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) *h* (*varsAPP* Σ*d g*))))
apply (*rule setsum-mono*)

apply (*rule setsum-mono*)
apply (*subgoal-tac* $\varrho' \in dom\ \Delta g$)
apply (*subgoal-tac monotonic-bound* (*the* ($\Delta g\ \varrho'$)))
apply (*subgoal-tac defined-bound* (*the* ($\Delta g\ \varrho'$)) g)
apply (*rule Phi-Mapping-app-ge-build-si-ef*,*assumption*+)
apply (*simp add*: *defined-AbstractDelta-def*)
apply (*simp add*: *monotonic-AbstractDelta-def*)
by *blast*

axioms *setsum-η-ef-equals-setsum-η*:
⟦ *SafeDepthSemanticsReal.SafeBoundSem* (*E1*, *E2*) $h\ k\ td$ (*AppE* g *as* rs' ()) (f, n) $hh\ k\ v$ (δ, m, s);
(∀ i<*length as*. *atom* (*as* ! i)); *length* xs = *length as*; *distinct* xs;
valid $\Sigma d\ \Sigma\vartheta\ \Sigma\Phi$; $\Sigma d\ g$ = *Some* (xs, rs, ef); *fv* (*AppE* g *as* rs' a) ⊆ *dom E1*;
monotonic-AbstractDelta Δg; $f \in dom\ \Sigma d$; *admissible* $f\ \eta\ k$;
set (*R-Args* $\Sigma t\ g$) ∪ {ϱ*self-f* g} = *dom* Δg;
dom (*instance-f* $f\ g\ \Delta g\ \psi$ (*sizesAPP* $\Sigma\Phi\ f$) *as*) = *dom* η;
defined-AbstractDelta $\Delta g\ g$⟧
⟹ ($\sum \varrho$ | (η-*ef* $\eta\ \psi$ (*R-Args* $\Sigma t\ g$)(ϱ*self-f* $g \mapsto$ *Suc* k)) ϱ = *Some* j.
the (*the* ($\Delta g\ \varrho$) (*build-si* (*map-of* (*zip* xs (*map* (*atom2val E1*) *as*))) h (*varsAPP* $\Sigma d\ g$)))) =
($\sum \varrho$ | $\eta\ \varrho$ = *Some* j. $\sum \varrho \in \{\varrho'.\ \psi\ \varrho'$ = *Some* $\varrho \wedge \varrho' \in$ *set* (*R-Args* $\Sigma t\ g$)$\}$.
the (*the* ($\Delta g\ \varrho$) (*build-si* (*map-of* (*zip* xs (*map* (*atom2val E1*) *as*))) h (*varsAPP* $\Sigma d\ g$))))

lemma *sizeAbstractDelta-si-APP-ge-sizeAbstractDelta-si-eg*:
⟦ *SafeDepthSemanticsReal.SafeBoundSem* (*E1*, *E2*) $h\ k\ td$ (*AppE* g *as* rs' ()) (f, n) $hh\ k\ v$ (δ, m, s);
(∀ i<*length as*. *atom* (*as* ! i)); *length* xs = *length as*; *distinct* xs;
valid $\Sigma d\ \Sigma\vartheta\ \Sigma\Phi$; $\Sigma d\ g$ = *Some* (xs, rs, ef); *fv* (*AppE* g *as* rs' a) ⊆ *dom E1*;
monotonic-AbstractDelta Δg; $f \in dom\ \Sigma d$; *admissible* $f\ \eta\ k$;
set (*R-Args* $\Sigma t\ g$) ∪ {ϱ*self-f* g} = *dom* Δg;
dom (*instance-f* $f\ g\ \Delta g\ \psi$ (*sizesAPP* $\Sigma\Phi\ f$) *as*) = *dom* η;
defined-AbstractDelta $\Delta g\ g$⟧
⟹ *sizeAbstractDelta-si* (*instance-f* $f\ g\ \Delta g\ \psi$ (*sizesAPP* $\Sigma\Phi\ f$) *as*) (*build-si E1* h (*varsAPP* $\Sigma d\ f$)) $j\ \eta$ >=
sizeAbstractDelta-si Δg (*build-si* (*map-of* (*zip* xs (*map* (*atom2val E1*) *as*))) h (*varsAPP* $\Sigma d\ g$)) j (η-*ef* $\eta\ \psi$ (*R-Args* $\Sigma t\ g$)(ϱ*self-f* $g \mapsto$ *Suc* k))
apply (*simp only*: *sizeAbstractDelta-si-def*)
apply (*subst sumset-instance-f-equals-sumset-phiMapping*,*simp*)
apply (*frule-tac* j=j **in** *setsum-build-si-le-setsum-phiMapping*,*assumption*+)
by (*subst setsum-η-ef-equals-setsum-η*,*assumption*+)

lemma *P-Δ-APP*:
⟦ *SafeDepthSemanticsReal.SafeBoundSem* (*E1*, *E2*) $h\ k\ td$ (*AppE* g *as* rs' ()) (f,

n) $hh\ k\ v\ (\delta,\ m,\ s)$;
$(\forall\, i{<}length\ as.\ atom\ (as\ !\ i))$; $length\ xs = length\ as$; $distinct\ xs$;
$valid\ \Sigma d\ \Sigma\vartheta\ \Sigma\Phi$; $\Sigma d\ g = Some\ (xs,\ rs,\ ef)$; $fv\ (AppE\ g\ as\ rs'\ a) \subseteq dom\ E1$;
$monotonic\text{-}AbstractDelta\ \Delta g$; $f \in dom\ \Sigma d$; $admissible\ f\ \eta\ k$;
$defined\text{-}AbstractDelta\ \Delta g\ g$;
$set\ (R\text{-}Args\ \Sigma t\ g) \cup \{\varrho self\text{-}f\ g\} = dom\ \Delta g$;
$Delta\text{-}ge\ \Delta g\ (build\text{-}si\ (map\text{-}of\ (zip\ xs\ (map\ (atom2val\ E1)\ as)))\ h\ (varsAPP\ \Sigma d\ g))\ (Suc\ k)$
$(\eta\text{-}ef\ \eta\ \psi\ (R\text{-}Args\ \Sigma t\ g)\ {+}{+}\ [\varrho self\text{-}f\ g \mapsto Suc\ k])\ \delta'$;
$dom\ (instance\text{-}f\ f\ g\ \Delta g\ \psi\ (sizesAPP\ \Sigma\Phi\ f)\ as) = dom\ \eta$;
$\delta = \delta'(k + 1 := None)$ ⟧
⟹ $Delta\text{-}ge\ (instance\text{-}f\ f\ g\ \Delta g\ \psi\ (sizesAPP\ \Sigma\Phi\ f)\ as)\ (build\text{-}si\ E1\ h\ (varsAPP\ \Sigma d\ f))\ k\ \eta\ \delta$
apply (*simp add*: *Delta-ge-def*)
apply (*rule ballI*)
apply (*erule-tac x=j* **in** *ballE*)
prefer *2* **apply** *simp*
apply (*frule-tac j=j* **and** $\psi{=}\psi$ **in** *sizeAbstractDelta-si-APP-ge-sizeAbstractDelta-si-eg*)
by (*assumption+*,*simp*,*assumption+*,*simp*,*assumption+*,*simp*)

lemma *SafeResourcesDADepth-APP*:
⟦ $\forall\, i{<}length\ as.\ atom\ (as\ !\ i)$;
$monotonic\text{-}bound\ \mu g$; $defined\text{-}bound\ \mu g\ g$;
$monotonic\text{-}bound\ \sigma g$; $defined\text{-}bound\ \sigma g\ g$;
$monotonic\text{-}AbstractDelta\ \Delta g$; $defined\text{-}AbstractDelta\ \Delta g\ g$;
$f \in dom\ \Sigma d$;
$\Sigma t\ g = Some(ti,\varrho s,t)$;
$fv\ eg \subseteq set\ xs$; $fvReg\ eg \subseteq set\ rs$; $fv\ eg \subseteq dom\ (sizesAPP\ \Sigma\Phi\ f)$;
$length\ xs = length\ ti$;

$\Sigma d\ g = Some\ (xs,rs,eg)$; $\Sigma b\ g = Some\ (projection\ g\ \Delta g,\mu g,\sigma g)$; $primops\ g = None$;
$l = real\ (length\ as)$; $q = real\ (length\ rs')$;
$argP\ \psi\ (R\text{-}Args\ \Sigma t\ g)\ (typesRegsAPP\ \Sigma\vartheta\ f)\ rs'$;
$\Delta = instance\text{-}f\ f\ g\ \Delta g\ \psi\ (sizesAPP\ \Sigma\Phi\ f)\ as$;
$\mu = mu\text{-}app\ \mu g\ as\ (sizesAPP\ \Sigma\Phi\ f)$;
$\sigma = \bigsqcup_c\ \{[]_{l\ +\ q},\ (substCostReal\ (addCostReal\ (addCostReal\ (sigma\text{-}app\ \sigma g\ as\ (sizesAPP\ \Sigma\Phi\ f))\ l)\ q)\ td)\}$;
$\models\models_{f\ ,\ n}\ \Sigma b$ ⟧
⟹ $AppE\ g\ as\ rs'\ a :_{f\ (typesAPP\ \Sigma\vartheta\ f)\ (sizesAPP\ \Sigma\Phi\ f)\ td,\ n}\ \{\!|\Delta,\ \mu,\ \sigma|\!\}$
apply (*case-tac* $g{\neq}f$)

apply (*frule lemma-19*,*assumption+*)
apply (*simp add*: *typesAPP-def*)
apply (*simp only*: *SafeResourcesDAssDepth.simps*)

apply (*rule impI*)
apply (*rule conjI*, *subst dom-instance-f*,*simp*)
apply (*rule allI*)+
apply (*rule impI*)
apply (*elim conjE*)

apply (*simp only*: *SafeResourcesDAss.simps*)
apply (*drule mp*,*simp*)

apply (*frule P1-f-n-APP-2*,*simp*,*force*,*force*)
apply (*elim exE*, *elim conjE*)

apply (*rotate-tac 29*)
apply (*erule-tac x*=(*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) **in** *allE*)
apply (*erule-tac x*=(*map-of* (*zip rs* (*map* (*the* ∘ *E2*) *rs'*))(*self* ↦ *Suc k*)) **in** *allE*)
apply (*erule-tac x*=*h* **in** *allE*)
apply (*rotate-tac 38*)
apply (*erule-tac x*=*Suc k* **in** *allE*)
apply (*rotate-tac 38*)
apply (*erule-tac x*=*h'* **in** *allE*)
apply (*erule-tac x*=*v* **in** *allE*)
apply (*rotate-tac 38*)
apply (*erule-tac x*=*δ'* **in** *allE*)
apply (*erule-tac x*=*m* **in** *allE*)
apply (*erule-tac x*=*s'* **in** *allE*)
apply (*erule-tac x*=(*η-ef η ψ* (*R-Args Σt g*) ++ [*ϱself-f g* ↦ *Suc k*]) **in** *allE*)
apply (*erule-tac x*= *build-si* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) *h* (*varsAPP Σd g*) **in** *allE*)

apply (*drule mp*)

apply (*rule conjI*)
apply (*subst eqSemRABound* [**where** *f*=*f*],*force*)

apply (*rule conjI*)
apply (*rule P-dyn-xs-subseteq-dom-E1-g*, *assumption*+)

apply (*rule conjI*)
apply (*rule P-dyn-fvReg-eg-subseteq-dom-E2-g*,*assumption*+,*simp*)

apply (*rule conjI*, *rule P-dyn-dom-η-ef*,*assumption*+)

apply (*rule conjI*, *simp*)

apply (*rule P-η-ef*,*simp*)

apply (*elim conjE*)

apply (*rule conjI*, *rule P-Δ-APP*)
apply (*assumption*+,*simp*,*assumption*+)

apply (*rule conjI*)
apply (*rule P-μ-APP*,*assumption*+)

apply (*rule P-σ-APP*)
apply (*assumption*+,*simp*,*assumption*+,*simp*,*simp*)

apply *simp*
apply (*case-tac n*)

  apply (*simp only*: *typesAPP-def*)
  apply (*simp only*: *SafeResourcesDAssDepth.simps*)
  apply (*rule impI*)
  apply (*rule conjI*, *subst dom-instance-f*,*simp*)
  apply (*rule allI*)+
  apply (*rule impI*)
  apply (*elim conjE*)
  apply (*frule P1-f-n-APP*,*assumption*+,*simp*)

apply (*frule lemma-20*)
apply (*force*,*force*,*simp*,*simp*)

apply (*simp only*: *typesAPP-def*)
apply (*simp only*: *SafeResourcesDAssDepth.simps*)

apply (*rule impI*)
apply (*drule mp*,*simp*)
apply (*elim conjE*)

apply (*rule conjI*, *subst dom-instance-f*, *simp*)
apply (*intro allI*, *rule impI*)
apply (*elim conjE*)

apply (*frule P1-f-n-ge-0-APP*,*simp*,*force*)
apply (*elim exE*, *elim conjE*)

apply (*rotate-tac 25*)
apply (*erule-tac x=*(*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) **in** *allE*)
apply (*erule-tac x=*(*map-of* (*zip rs* (*map* (*the* ∘ *E2*) *rs'*))(*self* ↦ *Suc k*)) **in** *allE*)
apply (*erule-tac x=h* **in** *allE*)
apply (*rotate-tac 40*)
apply (*erule-tac x=Suc k* **in** *allE*)
apply (*rotate-tac 40*)
apply (*erule-tac x=h'* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)
apply (*rotate-tac 40*)
apply (*erule-tac x=δ'* **in** *allE*)
apply (*erule-tac x=m* **in** *allE*)
apply (*erule-tac x=s'* **in** *allE*)
apply (*erule-tac x=*(*η-ef η ψ* (*R-Args* Σ*t g*) ++ [*ϱself-f g* ↦ *Suc k*]) **in** *allE*)
apply (*erule-tac x= build-si* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*))) *h* (*varsAPP* Σ*d f*) **in** *allE*)

apply (*drule mp*)

apply (*rule conjI*,*simp*)

apply (*rule conjI*)
apply (*rule P-dyn-xs-subseteq-dom-E1-g*, *assumption+*)

apply (*rule conjI*)
apply (*rule P-dyn-fvReg-eg-subseteq-dom-E2-g*,*assumption+*,*simp*)

apply (*rule conjI*, *rule P-dyn-dom-η-ef*)
apply (*force*,*simp*,*simp*,*assumption+*,*simp*,*assumption+*,*simp*,*simp*)

apply (*rule conjI*, *simp*)

apply (*frule P-η-ef*,*simp*)

apply (*elim conjE*)

apply (*rule conjI*)
apply (*frule P-Δ-APP*)
apply (*assumption+*,*simp*,*assumption+*,*simp*,*assumption+*,*simp*)

apply (*rule conjI*)
apply (*frule P-μ-APP*,*assumption+*,*simp*)

apply (*frule-tac σg=σg* **in** *P-σ-APP*)
by (*assumption+*,*simp*,*assumption+*,*simp*)

end

20 Proof-rules for certifying memory bounds, and soundness theorem

theory *ProofRulesCostes*
imports *CostesDepth Finite-Set*
begin

constdefs *costs-le*:: *Cost* ⇒ *Cost* ⇒ *bool* (- $\sqsubseteq_c$ - *1000*)
costs-le c1 c2 ≡ (*dom c1* = *dom c2*) ∧ (∀ *x* ∈ *dom c1*. *the* (*c1 x*) ≤ *the* (*c2 x*))

constdefs *abstractDelta-le*:: *AbstractDelta* ⇒ *AbstractDelta* ⇒ *bool*
(- $\sqsubseteq_\Delta$ - *1000*)
abstractDelta-le Δ*1* Δ*2* ≡ (*dom* Δ*1* = *dom* Δ*2*) ∧
(∀ ϱ ∈ *dom* Δ*1*. (*the* (Δ*1* ϱ)) $\sqsubseteq_c$ (*the* (Δ*2* ϱ)))

constdefs *resources-le* :: *AbstractDelta* ⇒ *AbstractMu* ⇒ *AbstractSigma*
⇒ *AbstractDelta* ⇒ *AbstractMu* ⇒ *AbstractSigma* ⇒ *bool*
($'($ - , - , - $')$ $\sqsubseteq_R$ $'($ - , - , - $')$ *1000*)
(Δ′ , μ′ , σ′) $\sqsubseteq_R$ (Δ , μ , σ) ≡
Δ′ $\sqsubseteq_\Delta$ Δ ∧ μ′ $\sqsubseteq_c$ μ ∧ σ′ $\sqsubseteq_c$ σ

inductive
ProofRulesCostes ::
unit Exp ⇒ *FunctionResourcesSignature* ⇒
FunName ⇒ *ThetaMapping* ⇒ *PhiMapping* ⇒ *real* ⇒
AbstractDelta ⇒ *AbstractMu* ⇒ *AbstractSigma* ⇒ *bool*
(- , - ⊢- - - - ′(- , - , - ′) *1000*)
where
litInt : *ConstE* (*LitN i*) *a*, Σ*b*
⊢*f* (ϑ*1*, ϑ*2*) Φ *td* ([]*f* ,
[]*0*, []*1*)

| *litBool*: *ConstE* (*LitB b*) *a*, Σ*b*
⊢*f* (ϑ*1*, ϑ*2*) Φ *td* ([]*f* ,
[]*0*, []*1*)

| *var1* : *VarE x a*, Σ*b*
⊢*f* (ϑ*1*, ϑ*2*) Φ *td* ([]*f* ,
[]*0*, []*1*)

| *var2* : ⟦(*typesRegsAPP* Σϑ *f*) *r* = *Some* ϱ;
(*sizesAPP* ΣΦ *f*) *x* = *Some* η ⟧
⟹ *CopyE x r d*, Σ*b*
⊢*f* (*typesAPP* Σϑ *f*) (*sizesAPP* ΣΦ *f*) *td* ([ϱ ↦ η], η, []*2*)

| *var3* : ⟦ (*typesRegsAPP* Σϑ *f*) *r* = *Some* ϱ;
(*sizesAPP* ΣΦ *f*) *x* = *Some* η ⟧
⟹ *ReuseE x a*, Σ*b*
⊢*f* (*typesAPP* Σϑ *f*) (*sizesAPP* ΣΦ *f*) *td* ([]*f* ,
[]*0*, []*1*)

| *let1* : ⟦ ∀ *C as r a*′. *e1* ≠ *ConstrE C as r a*′;
x1 ∉ *fv e1*;
x1 ∉ *set* (*varsAPP* Σ*d f*);
defined-AbstractDelta Δ*1 f*;
defined-bound |Δ*1*| *f*;
defined-bound μ*1 f*;
defined-bound σ*1 f*;
defined-AbstractDelta Δ*2 f*;
defined-bound μ*2 f*;
defined-bound σ*2 f*;
e1, Σ*b* ⊢*f* (ϑ*1*, ϑ*2*) Φ *0* (Δ*1*, μ*1*, σ*1*);
e2, Σ*b* ⊢*f* (ϑ*1*, ϑ*2*) Φ (*td*+*1*) (Δ*2*, μ*2*, σ*2*);
Δ = Δ*1* +Δ Δ*2*;
μ = ⨆*c* {μ*1*,|Δ*1*|+*c* μ*2*};
σ = ⨆*c*
{[]*2* +*c* σ*1*,
[]*1* +*c* σ*2*} ⟧

$$\Longrightarrow \mathit{Let}\ x1 = e1\ \mathit{In}\ e2\ a, \Sigma b \vdash_f (\vartheta 1, \vartheta 2)\ \Phi\ td\ (\Delta, \mu, \sigma)$$

$$\begin{array}{ll}
| & \mathit{let1c} : [\![(\mathit{typesRegsAPP}\ \Sigma\vartheta\ f)\ r = \mathit{Some}\ \varrho;\ \varrho \notin \mathit{dom}\ \Delta; \\
& \quad e2, \Sigma b \vdash_f (\mathit{typesAPP}\ \Sigma\vartheta\ f)\ (\mathit{sizesAPP}\ \Sigma\Phi\ f)\ td\ (\Delta, \mu, \sigma)\ ]\!] \\
& \quad \Longrightarrow \mathit{Let}\ x1 = \mathit{ConstrE}\ C\ \mathit{as}\ r\ a'\ \mathit{In}\ e2\ a, \Sigma b \\
& \quad \vdash_f (\mathit{typesAPP}\ \Sigma\vartheta\ f)\ (\mathit{sizesAPP}\ \Sigma\Phi\ f)\ td \\
& \qquad (\Delta +\!\!+ [\varrho \mapsto []_1], \\
& \qquad\ \mu +_c []_1, \\
& \qquad\ \sigma 2 +_c []_1)
\end{array}$$

$$\begin{array}{ll}
| & \mathit{case1} : [\![\mathit{length}\ \mathit{alts} = \mathit{length}\ \mathit{assert}; \\
& \quad \mathit{length}\ \mathit{alts} > 0; \\
& \quad (\forall\ i < \mathit{length}\ \mathit{assert}. \\
& \qquad \mathit{defined\text{-}AbstractDelta}\ (\mathit{AbstractDeltaSpaceCost}\ (\mathit{assert}\ !\ i))\ f); \\
& \quad (\forall\ i < \mathit{length}\ \mathit{assert}. \\
& \qquad \mathit{monotonic\text{-}AbstractDelta}\ (\mathit{AbstractDeltaSpaceCost}\ (\mathit{assert}\ !\ i))); \\
& \quad (\forall\ i < \mathit{length}\ \mathit{assert}. \\
& \qquad \mathit{defined\text{-}bound}\ (\mathit{AbstractMuSpaceCost}\ (\mathit{assert}\ !\ i))\ f); \\
& \quad (\forall\ i < \mathit{length}\ \mathit{assert}. \\
& \qquad \mathit{monotonic\text{-}bound}\ (\mathit{AbstractMuSpaceCost}\ (\mathit{assert}\ !\ i))); \\
& \quad (\forall\ i < \mathit{length}\ \mathit{assert}. \\
& \qquad \mathit{defined\text{-}bound}\ (\mathit{AbstractSigmaSpaceCost}\ (\mathit{assert}\ !\ i))\ f); \\
& \quad (\forall\ i < \mathit{length}\ \mathit{assert}. \\
& \qquad \mathit{monotonic\text{-}bound}\ (\mathit{AbstractSigmaSpaceCost}\ (\mathit{assert}\ !\ i))); \\
& \quad \forall\ i < \mathit{length}\ \mathit{alts}. \\
& \quad \mathit{snd}\ (\mathit{alts}\ !\ i), \Sigma b \\
& \qquad \vdash_f (\vartheta 1, \vartheta 2)\ \Phi\ (td +\!\!+ \mathit{real}\ (\mathit{length}\ (\mathit{snd}\ (\mathit{extractP}\ (\mathit{fst}\ (\mathit{alts}\ !\ i))))))) \\
& \qquad\ (\mathit{AbstractDeltaSpaceCost}\ (\mathit{assert}!i), \\
& \qquad\ \mathit{AbstractMuSpaceCost}\ (\mathit{assert}!i), \\
& \qquad\ \mathit{AbstractSigmaSpaceCost}\ (\mathit{assert}!i)); \\
& \quad \Delta = \bigsqcup_\Delta f\ (\mathit{map}\ \mathit{AbstractDeltaSpaceCost}\ \mathit{assert}); \\
& \quad \mu = \bigsqcup_c (\mathit{map}\ \mathit{AbstractMuSpaceCost}\ \mathit{assert}); \\
& \quad \sigma = \bigsqcup_c (\mathit{map}\ (\lambda\ (\Delta,n).\ \mathit{addCostReal}\ \Delta\ n) \\
& \qquad (\mathit{zip}\ (\mathit{map}\ \mathit{AbstractSigmaSpaceCost}\ \mathit{assert}) \\
& \qquad\ (\mathit{map}\ \mathit{num\text{-}r}\ \mathit{alts})))]\!] \\
& \Longrightarrow \mathit{Case}\ (\mathit{VarE}\ x\ a)\ \mathit{Of}\ \mathit{alts}\ a', \Sigma b \\
& \vdash_f (\vartheta 1, \vartheta 2)\ \Phi\ td\ (\Delta,\mu,\sigma)
\end{array}$$

$$\begin{array}{ll}
| & \mathit{app\text{-}primop} : [\![\mathit{primops}\ g = \mathit{Some}\ \mathit{oper}]\!] \\
& \quad \Longrightarrow \mathit{AppE}\ g\ [a1,a2]\ []\ a, \Sigma b \\
& \quad \vdash_f (\vartheta 1, \vartheta 2)\ \Phi\ td \\
& \quad ([]_g, \\
& \quad\ []_0, []_2)
\end{array}$$

$$\begin{array}{ll}
| & \mathit{app} : [\![\Sigma d\ g = \mathit{Some}\ (xs,rs,eg); \Sigma b\ g = \mathit{Some}\ (\mathit{projection}\ g\ \Delta g,\mu g,\sigma g); \\
& \quad \mathit{primops}\ g = \mathit{None};
\end{array}$$

$l = real\ (length\ as)$; $q = real\ (length\ rs')$;
$argP\ \psi\ (R\text{-}Args\ \Sigma t\ g)\ (typesRegsAPP\ \Sigma\vartheta\ f)\ rs'$;
$\forall\, i{<}length\ as.\ atom\ (as\ !\ i)$;
$monotonic\text{-}bound\ \mu g$; $defined\text{-}bound\ \mu g\ g$;
$monotonic\text{-}bound\ \sigma g$; $defined\text{-}bound\ \sigma g\ g$;
$monotonic\text{-}AbstractDelta\ \Delta g$; $defined\text{-}AbstractDelta\ \Delta g\ g$;
$f \in dom\ \Sigma d$;
$\Sigma t\ g = Some(ti,\varrho s,t)$;
$fv\ eg \subseteq set\ xs$; $fvReg\ eg \subseteq set\ rs$;
$fv\ eg \subseteq dom\ (sizesAPP\ \Sigma\Phi\ f)$;
$length\ xs = length\ ti$;
$\Delta = instance\text{-}f\ f\ g\ \Delta g\ \psi\ (sizesAPP\ \Sigma\Phi\ f)\ as$;
$\mu = mu\text{-}app\ \mu g\ as\ (sizesAPP\ \Sigma\Phi\ f)$;
$\sigma = \bigsqcup c\ \ \{[]_l + q,$
$(substCostReal\ (addCostReal\ (addCostReal$
$(sigma\text{-}app\ \sigma g\ as\ (sizesAPP\ \Sigma\Phi\ f))\ \ l)\ \ q)\ td)\}[\![$
$\Longrightarrow AppE\ g\ as\ rs'\ a, \Sigma b \vdash_f (typesAPP\ \Sigma\vartheta\ f)\ \ (sizesAPP\ \Sigma\Phi\ f)\ td$
$(\Delta,\ \mu,\ \sigma)$

$|\ \ rec\ \ : [\![\ \Sigma d\ f = Some\ (xs,rs,ef)$;
$f \notin dom\ \Sigma b$;
$(\varrho self\text{-}f\ f) \notin dom\ \Delta$;
$finite\ (dom\ \Delta)$;
$ef,\ \Sigma b(f{\mapsto}(\Delta,\mu,\sigma))$
$\vdash_f (typesAPP\ \Sigma\vartheta\ f)\ (sizesAPP\ \Sigma\Phi\ f)\ real(length\ \ (varsAPP\ \Sigma d\ f) + length\ (regionsAPP\ \Sigma d\ f$
$(\Delta',\ \mu',\ \sigma')$;
$(projection\ f\ \ \Delta',\ \mu',\ \sigma')\ \ \sqsubseteq_R\ \ (\Delta,\ \mu,\ \sigma)[\![$
$\Longrightarrow\ ef, \Sigma b \vdash_f (typesAPP\ \Sigma\vartheta\ f)\ (sizesAPP\ \Sigma\Phi\ f)\ \ real(length\ (varsAPP\ \Sigma d\ f){+}length(region$
$\mu',\ \sigma')$

lemma *equiv-all-n-SafeResourcesDAssDepth-SafeResourcesDAss*:
$\forall\, n.\ SafeResourcesDAssDepth\ e\ f\ (typesAPP\ \Sigma\vartheta\ f)\ (sizesAPP\ \Sigma\Phi\ f)\ td\ n\ \Delta\ \mu$
σ
$\Longrightarrow e :_f (typesAPP\ \Sigma\vartheta\ f)\ (sizesAPP\ \Sigma\Phi\ f)\ td\ \{\!|\Delta,\ \mu,\ \sigma|\!\}$
apply (*simp only*: *typesAPP-def*)
apply (*simp only*: *SafeResourcesDAss.simps*)
apply (*simp only*: *SafeResourcesDAssDepth.simps*)
apply (*rule impI*)
apply (*rule conjI*,*simp*)+
apply (*intro allI*, *rule impI*)
apply (*elim conjE*)
apply (*frule-tac f*=*f* **in** *eqSemRADepth*)
apply (*simp only*: *SafeBoundSem-def*)
apply (*elim exE*)
apply (*rename-tac n*)

apply (*erule-tac x=n* **in** *allE*)
apply (*drule mp,simp*)
apply (*elim conjE*)
apply (*erule-tac x=E1* **in** *allE*)
apply (*erule-tac x=E2* **in** *allE*)
apply (*erule-tac x=h* **in** *allE*)
apply (*erule-tac x=k* **in** *allE*)
apply (*erule-tac x=hh* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)
apply (*erule-tac x=δ* **in** *allE*)
apply (*erule-tac x=m* **in** *allE*)
apply (*erule-tac x=s* **in** *allE*)
apply (*erule-tac x=η* **in** *allE*)
apply (*erule-tac x=si* **in** *allE*)
apply (*drule mp*)
apply (*rule conjI*, *simp add*: *Let-def*, *force*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *simp*)
apply *simp*
by *simp*

lemma *equiv-SafeResourcesDAss-all-n-SafeResourcesDAssDepth*:
$e :_f (typesAPP\ \Sigma\vartheta\ g)\ (sizesAPP\ \Sigma\Phi\ g)\ td\ \{\!|\Delta, \mu, \sigma|\!\}$
$\Longrightarrow \forall n.\ SafeResourcesDAssDepth\ e\ f\ (typesAPP\ \Sigma\vartheta\ g)\ (sizesAPP\ \Sigma\Phi\ g)\ td\ n\ \Delta\ \mu\ \sigma$
apply (*simp only*: *typesAPP-def*)
apply (*simp only*: *SafeResourcesDAss.simps*)
apply (*simp only*: *SafeResourcesDAssDepth.simps*)
apply *clarsimp*
apply (*simp only*: *SafeBoundSem-def*)
apply (*simp add*: *Let-def*)
apply (*elim exE*)
apply (*elim conjE*)
apply (*rotate-tac 7*)
apply (*erule-tac x=E1* **in** *allE*)
apply (*erule-tac x=E2* **in** *allE*)
apply (*erule-tac x=h* **in** *allE*)
apply (*rotate-tac 9*)
apply (*erule-tac x=k* **in** *allE*)
apply (*erule-tac x=hh* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)
apply (*erule-tac x=δ* **in** *allE*)
apply (*erule-tac x=m* **in** *allE*)
apply (*erule-tac x=s* **in** *allE*)
apply (*erule-tac x=η* **in** *allE*)
apply (*frule-tac td=td* **in** *eqSemDepthRA*)

apply (*drule mp*,*force*)
by *simp*

lemma *lemma-5*:
∀ *n*. *SafeResourcesDAssDepth e f* (*typesAPP* Σϑ *f*) (*sizesAPP* ΣΦ *f*) *td n* Δ *μ*
σ ≡
e :*f* (*typesAPP* Σϑ *f*) (*sizesAPP* ΣΦ *f*) *td* {|Δ, *μ*, *σ*|}
apply (*rule eq-reflection*)
apply (*rule iffI*)

apply (*rule equiv-all-n-SafeResourcesDAssDepth-SafeResourcesDAss*,*force*)

by (*rule equiv-SafeResourcesDAss-all-n-SafeResourcesDAssDepth*,*force*)

declare *fun-upd-apply* [*simp del*]
declare *SafeResourcesDAss.simps* [*simp del*]
declare *SafeResourcesDAssDepth.simps* [*simp del*]

lemma *imp-ValidGlobalResourcesEnv-all-n-ValidGlobalResourcesEnvDepth*:
ValidGlobalResourcesEnv Σ*b*
⟹ ∀ *n*. ⊨⊨$_{f,n}$ Σ*b*
apply (*erule ValidGlobalResourcesEnv.induct*)
apply (*rule allI*)
apply (*rule ValidGlobalResourcesEnvDepth.base*)
apply (*rule ValidGlobalResourcesEnv.base*)
apply *simp*
apply (*rule allI*)
apply (*case-tac fa=f*,*simp*)
apply (*induct-tac n*)
apply (*rule ValidGlobalResourcesEnvDepth.depth0*,*simp*,*simp*)
apply (*rule-tac ValidGlobalResourcesEnvDepth.step*)
apply (*simp*,*simp*,*simp*)
apply (*frule-tac f=f* **in** *equiv-SafeResourcesDAss-all-n-SafeResourcesDAssDepth*,*force*)
apply (*rule ValidGlobalResourcesEnvDepth.g*)
by (*simp*,*simp*,*simp*,*simp*)

lemma *imp-ValidGlobalResourcesEnv-all-n-ValidGlobalResourcesEnvDepth*:
ValidGlobalResourcesEnv Σ*b*
⟹ ∀ *n*. ⊨⊨$_{f,n}$ Σ*b*
apply (*erule ValidGlobalResourcesEnv.induct*)
apply (*rule allI*)

apply (*rule ValidGlobalResourcesEnvDepth.base*)
apply (*rule ValidGlobalResourcesEnv.base*)
apply *simp*
apply (*rule allI*)
apply (*case-tac fa=f*,*simp*)
apply (*induct-tac n*)
apply (*rule ValidGlobalResourcesEnvDepth.depth0*,*simp*,*simp*)
apply (*rule-tac ValidGlobalResourcesEnvDepth.step*)
apply (*simp*,*simp*,*simp*)
apply (*frule-tac f=f* **in** *equiv-SafeResourcesDAss-all-n-SafeResourcesDAssDepth*,*force*)
apply (*rule ValidGlobalResourcesEnvDepth.g*)
by (*simp*,*simp*,*simp*,*simp*)

lemma *imp-ValidResourcesDepth-n-SigmaResources-Valid-Sigma* [*rule-format*]:
$\models\models_{f,n} \Sigma b$
$\longrightarrow f \notin dom\ \Sigma b$
$\longrightarrow ValidGlobalResourcesEnv\ \Sigma b$
apply (*rule impI*)
apply (*erule ValidGlobalResourcesEnvDepth.induct*,*simp-all*)
apply (*rule impI*)
by (*rule ValidGlobalResourcesEnv.step*,*simp-all*)

lemma *imp-f-notin-SigmaResources-ValiddDepth-n-SigmaResources-Valid-Sigma*:
$[\![f \notin dom\ \Sigma b; \forall\ n.\ \models\models_{f,n} \Sigma b]\!]$
$\Longrightarrow ValidGlobalResourcesEnv\ \Sigma b$
apply (*erule-tac x=n* **in** *allE*)
by (*rule imp-ValidResourcesDepth-n-SigmaResources-Valid-Sigma*,*assumption+*)

lemma *the-AbstractDeltaSpaceCost-upd*:
$AbstractDeltaSpaceCost\ (the\ ((\Sigma(f \mapsto (\Delta,\mu,\sigma)))\ f)) = \Delta$
by (*simp add*: *fun-upd-apply add*: *dom-def*)

lemma *the-AbstractMuSpaceCost-upd*:
$AbstractMuSpaceCost\ (the\ ((\Sigma(f \mapsto (\Delta,\mu,\sigma)))\ f)) = \mu$
by (*simp add*: *fun-upd-apply add*: *dom-def*)

lemma *the-AbstractSigmaSpaceCost-upd*:
$AbstractSigmaSpaceCost\ (the\ ((\Sigma(f \mapsto (\Delta,\mu,\sigma)))\ f)) = \sigma$
by (*simp add*: *fun-upd-apply add*: *dom-def*)

lemma *the-AbstractDeltaSpaceCost-other-upd*:
$g \neq f$
$\Longrightarrow AbstractDeltaSpaceCost\ (the\ ((\Sigma(g \mapsto (\Delta,\mu,\sigma)))\ f)) = AbstractDeltaSpaceCost\ (the\ (\Sigma\ f))$
by (*simp add*: *fun-upd-apply add*: *dom-def*)

lemma *the-AbstractMuSpaceCost-other-upd*:
$g \neq f$
$\Longrightarrow$ *AbstractMuSpaceCost* (*the* ((Σ($g \mapsto (\Delta,\mu,\sigma)$)) f)) = *AbstractMuSpaceCost* (*the* (Σ f))
by (*simp add*: *fun-upd-apply add*: *dom-def*)

lemma *the-AbstractSigmaSpaceCost-other-upd*:
$g \neq f$
$\Longrightarrow$ *AbstractSigmaSpaceCost* (*the* ((Σ($g \mapsto (\Delta,\mu,\sigma)$)) f)) = *AbstractSigmaSpaceCost* (*the* (Σ f))
by (*simp add*: *fun-upd-apply add*: *dom-def*)

lemma *the-AbstractDeltaSpaceCost-other-projection-upd*:
$g \neq f$
$\Longrightarrow$ *AbstractDeltaSpaceCost* (*the* ((Σ($g \mapsto$ (*projection* g Δ,μ,σ))) f)) = *AbstractDeltaSpaceCost* (*the* (Σ f))
by (*simp add*: *fun-upd-apply add*: *dom-def*)

lemma *the-AbstractMuSpaceCost-other-projection-upd*:
$g \neq f$
$\Longrightarrow$ *AbstractMuSpaceCost* (*the* ((Σ($g \mapsto$ (*projection* g Δ,μ,σ))) f)) = *AbstractMuSpaceCost* (*the* (Σ f))
by (*simp add*: *fun-upd-apply add*: *dom-def*)

lemma *the-AbstractSigmaSpaceCost-other-projection-upd*:
$g \neq f$
$\Longrightarrow$ *AbstractSigmaSpaceCost* (*the* ((Σ($g \mapsto$ (*projection* g Δ,μ,σ))) f)) = *AbstractSigmaSpaceCost* (*the* (Σ f))
by (*simp add*: *fun-upd-apply add*: *dom-def*)

axioms *equals-projection*:
projection f $\Delta' =$ *projection* f Δ
$\Longrightarrow \Delta' = \Delta$

lemma *Theorem-4-aux* [*rule-format*]:
$\models\models_{f,n} \Sigma b$
$\longrightarrow n =$ *Suc* n'
$\longrightarrow f \in$ *dom* Σb
$\longrightarrow \Sigma b\, f =$ *Some* (*projection* f Δ, μ ,σ)
$\longrightarrow$ (*bodyAPP* $\Sigma d\, f$) :$_f$ (*typesAPP* $\Sigma\vartheta$ f) (*sizesAPP* $\Sigma\Phi$ f) *real*(*length* (*varsAPP* Σd f) + *length* (*regionsAPP* Σ
{| Δ, μ, σ |}
apply (*rule impI*)
apply (*erule ValidGlobalResourcesEnvDepth.induct*)
apply *simp* **apply** *simp*
apply (*rule impI*)+

apply (*subst* (*asm*) *map-upd-Some-unfold*,*simp*)
apply (*elim conjE*, *frule equals-projection*,*simp*)
apply *clarsimp*
by (*simp add*: *fun-upd-apply add*: *dom-def*)

lemma *Theorem-4*:
⟦ ∀ *n* > *0*. ⊨⊨$_{f,n}$ Σ*b*; *f* ∈ *dom* Σ*b*; Σ*b f* = *Some* (*projection f* Δ, μ ,σ) ⟧
⟹ ∀ *n*. (*bodyAPP* Σ*d f*) :$_f$ (*typesAPP* Σϑ *f*) (*sizesAPP* ΣΦ *f*) *real*(*length* (*varsAPP* Σ*d f*) + *length* (*regionsAP*
{⦃ Δ , μ, σ ⦄}
apply (*rule allI*)
apply (*rule-tac n*=*Suc n* **in** *Theorem-4-aux*)
by (*force*,*simp*,*simp*,*simp*)

lemma *Theorem-5-aux* [*rule-format*]:
⊨⊨$_{f,n}$ Σ*b*
⟶ *n* = *Suc n'*
⟶ *f* ∈ *dom* Σ*b*
⟶ Σ*b f* = *Some* (*projection f* Δ, μ ,σ)
⟶ (*bodyAPP* Σ*d f*) :$_f$ (*typesAPP* Σϑ *f*) (*sizesAPP* ΣΦ *f*) *real*(*length* (*varsAPP* Σ*d f*) + *length* (*regionsAPP* Σ
{⦃ Δ, μ, σ ⦄}
⟶ ⊨⊨ Σ*b*
apply (*rule impI*)
apply (*erule ValidGlobalResourcesEnvDepth.induct*,*simp-all*)
apply (*rule impI*)+
apply (*rule ValidGlobalResourcesEnv.step*)
apply (*simp*,*simp*)
apply (*subst* (*asm*) *map-upd-Some-unfold*,*simp*)
apply (*elim conjE*, *frule equals-projection*,*simp*)
apply (*rule impI*)+
apply (*case-tac g*=*f*,*simp-all*)
apply (*rule ValidGlobalResourcesEnv.step*,*simp-all*)
apply (*subgoal-tac f* ∈ *dom* Σ*b*,*simp*)
prefer *2* **apply** (*simp add*: *fun-upd-apply add*: *dom-def*)
by (*simp add*: *fun-upd-apply add*: *dom-def*)+

lemma *Theorem-5*:
⟦ ∀ *n* > *0*. ⊨⊨$_{f,n}$ Σ*b*; *f* ∈ *dom* Σ*b*; Σ*b f* = *Some* (*projection f* Δ, μ ,σ);
(*bodyAPP* Σ*d f*) :$_f$ (*typesAPP* Σϑ *f*) (*sizesAPP* ΣΦ *f*) *real*(*length* (*varsAPP* Σ*d f*) + *length* (*regionsAPP* Σ*d f*)
{⦃ Δ, μ ,σ ⦄}⟧
⟹ ⊨⊨ Σ*b*
apply (*rule-tac n*=*Suc n* **in** *Theorem-5-aux*)
by *simp-all*

lemma *signature-correct* [*rule-format*]:
$\models\models_{f,n} \Sigma b$
$\longrightarrow f \in dom\ \Sigma b$
$\longrightarrow (\exists\ \Delta.\ (AbstractDeltaSpaceCost\ (the\ (\Sigma b\ f))) = projection\ f\ \Delta)$
apply (*rule impI*)
apply (*erule ValidGlobalResourcesEnvDepth.induct*,*simp-all*)
apply (*simp add*: *fun-upd-apply add*: *dom-def*,*force*)
apply (*rule-tac* $x{=}\Delta$ **in** *exI*)
apply (*simp add*: *fun-upd-apply add*: *dom-def*)
apply (*rule impI*)
apply (*drule mp*,*simp*)
apply (*elim exE*)
apply (*rule-tac* $x{=}\Delta'$ **in** *exI*)
by (*simp add*: *fun-upd-apply add*: *dom-def*)

lemma *imp-f-in-SigmaResources-ValidDepth-n-SigmaResources-Valid-Sigma*:
$[\![\ \forall\ n.\ \models\models_{f,n} \Sigma b;\ f \in dom\ \Sigma b\ ]\!]$
$\Longrightarrow ValidGlobalResourcesEnv\ \Sigma b$
apply (*subgoal-tac* $\models\models_{f,n} \Sigma b$)
prefer *2* **apply** *simp*
apply (*frule signature-correct*,*assumption*)
apply (*elim exE*)
apply (*frule domD*)
apply (*elim exE*, *case-tac b*, *case-tac ba*,*simp*,*clarsimp*)
apply (*subgoal-tac f* $\in$ *dom* Σb)
apply (*subgoal-tac* $\models\models_{f,0} \Sigma b \wedge (\forall n > 0.\ \models\models_{f,n} \Sigma b)$,*elim conjE*)
prefer *2* **apply** *simp*
apply (*frule Theorem-4*,*simp-all*,*force*)
apply (*subgoal-tac* $\models\models_{f,0} \Sigma b \wedge (\forall n > 0.\ \models\models_{f,n} \Sigma b)$,*elim conjE*)
prefer *2* **apply** *simp*
apply (*frule Theorem-5*,*simp-all*,*force*)
apply (*rule equiv-all-n-SafeResourcesDAssDepth-SafeResourcesDAss*,*simp*)
by (*simp add*: *dom-def*)

lemma *imp-all-n-ValidGlobalResourcesEnvDepth-ValidGlobalResourcesEnv*:
$[\![\ \forall\ n.\ \models\models_{f,n} \Sigma b\ ]\!]$
$\Longrightarrow ValidGlobalResourcesEnv\ \Sigma b$
apply (*case-tac* $f \notin dom\ \Sigma b$,*simp-all*)
apply (*rule imp-f-notin-SigmaResources-ValiddDepth-n-SigmaResources-Valid-Sigma*,*assumption*+)
by (*rule imp-f-in-SigmaResources-ValidDepth-n-SigmaResources-Valid-Sigma*,*assumption*+)

lemma *lemma-6*:

$\forall\ n.\ \models\models_{f,n} \Sigma b \equiv$
$\quad ValidGlobalResourcesEnv\ \Sigma b$
apply (*rule eq-reflection*)
apply (*rule iffI*)

apply (*rule-tac f=f* **in** *imp-all-n-ValidGlobalResourcesEnvDepth-ValidGlobalResourcesEnv,force*)

by (*rule imp-ValidGlobalResourcesEnv-all-n-ValidGlobalResourcesEnvDepth,force*)

lemma *lemma-7*:
$[\![\forall\ n.\ e, \Sigma b :_f (typesAPP\ \Sigma\vartheta\ f)\ (sizesAPP\ \Sigma\Phi\ f)\ td, n \{\!|\ \Delta\ ,\ \mu\ ,\ \sigma\ |\!\}]\!]$
$\quad \Longrightarrow SafeResourcesDAssCntxt\ e\ \Sigma b\ f\ (typesAPP\ \Sigma\vartheta\ f)\ (sizesAPP\ \Sigma\Phi\ f)\ td\ \Delta$
$\mu\ \sigma$
apply (*simp only*: *typesAPP-def*)
apply (*simp only*: *SafeResourcesDAssDepthCntxt.simps*)
apply (*subgoal-tac*
$(\forall n.\ \models\models_{f,n} \Sigma b) \longrightarrow$
$(\forall n.\ \ e :_f (typesAPP\ \Sigma\vartheta\ f)\ (sizesAPP\ \Sigma\Phi\ f)\ td, n \{\!|\ \Delta\ ,\ \mu\ ,\ \sigma\ |\!\}))$
apply (*erule thin-rl*)
apply (*subst* (*asm*) *lemma-5*)
apply (*subst* (*asm*) *lemma-6*)
apply (*simp add*: *SafeResourcesDAssCntxt.simps*)
apply (*simp only*: *typesAPP-def*)
apply (*rule impI*)
apply (*simp only*: *typesAPP-def*)
by *force*

lemma *fold-op-plus-le* [*rule-format*]:
$finite\ (\{\varrho.\ \eta\ \varrho = Some\ j\})$
$\quad \longrightarrow \Delta' \sqsubseteq_\Delta \Delta$
$\quad \longrightarrow fold\ op\ +\ (\lambda\varrho.\ the\ (the\ (\Delta'\ \varrho)\ si))\ (0::real)\ (\{\varrho.\ \eta\ \varrho = Some\ j\}) \leq$
$\quad\ fold\ op\ +\ (\lambda\varrho.\ the\ (the\ (\Delta\ \varrho)\ si))\ (0::real)\ (\{\varrho.\ \eta\ \varrho = Some\ j\})$
apply (*rule impI*)
apply (*induct rule*: *finite-induct*)
apply (*rule impI*)
apply *simp*
apply (*rule impI*)+
apply (*simp add*: *ran-def*)
apply (*simp add*: *abstractDelta-le-def*)
apply (*subgoal-tac the* (*the* ($\Delta'\ x$) *si*) $\leq$ *the* (*the* ($\Delta\ x$) *si*))

apply *clarsimp*
apply (*elim conjE*)
apply (*case-tac* $x \in dom\ \Delta$)
apply (*simp add*: *costs-le-def*)
apply (*case-tac si* $\in$*dom* (*the* ($\Delta'\ x$)),*force*,*force*)
apply (*subgoal-tac* $\Delta\ x = None$,*simp*)
apply (*subgoal-tac* $\Delta'\ x = None$,*simp*,*force*)
by *force*

lemma *sizeAbstractDelta-si-le*:
$[\![\ finite\ (\{\varrho.\ \eta\ \varrho = Some\ j \wedge \varrho \in dom\ \Delta\});$
$\Delta' \sqsubseteq_\Delta\ \Delta\ ]\!]$
$\Longrightarrow sizeAbstractDelta\text{-}si\ \Delta'\ si\ j\ \eta \leq$
$sizeAbstractDelta\text{-}si\ \Delta\ si\ j\ \eta$
apply (*simp add*: *sizeAbstractDelta-si-def*)
apply (*simp add*: *setsum-def*)
apply (*rule impI*)
by (*rule fold-op-plus-le*,*assumption*+)

lemma *resources-le-Delta-ge*:
$[\![\ \forall j{\in}\{0..k\}.\ finite\ (\{\varrho.\ \eta\ \varrho = Some\ j \wedge \varrho \in dom\ \Delta\});$
$Delta\text{-}ge\ \Delta'\ si\ k\ \eta\ \delta;\ \Delta' \sqsubseteq_\Delta\ \Delta]\!]$
$\Longrightarrow Delta\text{-}ge\ \Delta\ si\ k\ \eta\ \delta$
apply (*simp add*: *Delta-ge-def*)
apply (*rule ballI*)
apply (*subgoal-tac dom* $\Delta' = dom\ \Delta$)
prefer *2* **apply** (*simp add*: *abstractDelta-le-def*)
apply (*erule-tac* $x{=}j$ **in** *ballE*,*simp*)
apply (*frule-tac* $\eta{=}\eta$ **and** $j{=}j$ **and** *si*=*si* **in** *sizeAbstractDelta-si-le*,*assumption*+)

apply *force*
by *simp*

lemma *resources-le-Mu-ge*:
$[\![\ mu\text{-}ge\ \mu'\ si\ s;\ \mu' \sqsubseteq_c \mu\ ]\!]$
$\Longrightarrow mu\text{-}ge\ \mu\ si\ s$
apply (*simp add*: *mu-ge-def*)
apply (*simp add*: *costs-le-def*)
apply (*erule real-le-trans*)
apply (*elim conjE*)
by (*erule-tac* $x{=}si$ **in** *ballE*,*simp*,*force*)

lemma *resources-le-Sigma-ge*:
$[\![\ sigma\text{-}ge\ \sigma'\ si\ s;\ \sigma' \sqsubseteq_c \sigma\ ]\!]$

$\Longrightarrow$ *sigma-ge* σ *si s*
apply (*simp add*: *sigma-ge-def*)
apply (*simp add*: *costs-le-def*)
apply (*erule real-le-trans*)
apply (*elim conjE*)
by (*erule-tac x=si* **in** *ballE*,*simp*,*force*)

lemma *dom-projection-Delta*:
$dom\ (projection\ f\ \Delta) = dom\ \Delta - \{(\varrho self\text{-}f\ f)\}$
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp add*: *projection-def*,*clarsimp*)
apply (*split split-if-asm*,*simp*,*force*)
apply (*rule subsetI*)
apply (*simp add*: *projection-def*)
by *clarsimp*

lemma *dom-delta'-equals-dom-delta-*ϱ*self*:
$[\![\ insert\ (\varrho self\text{-}f\ f)\ (set\ (R\text{-}Args\ \Sigma t\ f)) = dom\ \Delta';$
$projection\ f\ \Delta' \sqsubseteq_{\Delta}\ \Delta\ ]\!]$
$\Longrightarrow dom\ \Delta' = insert\ (\varrho self\text{-}f\ f)\ (dom\ \Delta)$
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp add*: *abstractDelta-le-def*)
apply (*subst* (*asm*) *dom-projection-Delta*)
apply *blast*
apply (*rule subsetI*)
apply (*simp add*: *abstractDelta-le-def*)
apply (*subst* (*asm*) *dom-projection-Delta*)
by *blast*

lemma *projection-prop*:
$[\![\ insert\ (\varrho self\text{-}f\ f)\ (set\ (R\text{-}Args\ \Sigma t\ f)) = dom\ \Delta';$
$projection\ f\ \Delta' \sqsubseteq_{\Delta}\ \Delta\ ]\!]$
$\Longrightarrow \Delta' \sqsubseteq_{\Delta} \Delta(\varrho self\text{-}f\ f \mapsto the\ (\Delta'\ (\varrho self\text{-}f\ f)))$
apply (*simp* (*no-asm*) *add*: *abstractDelta-le-def*)
apply (*rule conjI*)
apply (*rule dom-delta'-equals-dom-delta-*ϱ*self*,*simp*,*simp*)
apply (*subst dom-delta'-equals-dom-delta-*ϱ*self*,*simp*,*simp*)
apply (*rule ballI*)
apply (*simp only*: *abstractDelta-le-def*)
apply (*elim conjE*,*clarsimp*,*erule disjE*)
apply (*simp add*: *fun-upd-apply add*: *dom-def*)
apply (*simp add*: *costs-le-def*)

apply (*erule-tac x=ϱ* **in** *ballE*)
prefer *2* **apply** *simp*
apply (*subst* (*asm*) *dom-projection-Delta*)
apply (*subgoal-tac* $\varrho \neq (\varrho\text{self-}f\ f)$)
apply (*simp add*: *projection-def*)
apply (*simp add*: *fun-upd-apply add*: *dom-def*)
by *blast*

lemma *sizeAbstractDelta-si-le*:
$[\![$ *insert* $(\varrho\text{self-}f\ f)$ (*set* (*R-Args* $\Sigma t\ f$)) = *dom* Δ';
finite $(\{\varrho.\ \eta\ \varrho = \text{Some}\ j \wedge \varrho \in \text{dom}\ \Delta\})$;
projection $f\ \Delta' \sqsubseteq_\Delta\ \Delta\ ]\!]$
$\Longrightarrow$ *sizeAbstractDelta-si* Δ' *si j* $\eta \leq$
sizeAbstractDelta-si $(\Delta(\varrho\text{self-}f\ f \mapsto \text{the}\ (\Delta'\ (\varrho\text{self-}f\ f))))$ *si j* η
apply (*simp add*: *sizeAbstractDelta-si-def*)
apply (*simp add*: *setsum-def*)
apply (*rule impI*)
apply (*rule fold-op-plus-le*,*assumption*+)
by (*rule projection-prop*,*assumption*+)

lemma *resources-le-Delta-ge*:
$[\![$ *insert* $(\varrho\text{self-}f\ f)$ (*set* (*R-Args* $\Sigma t\ f$)) = *dom* Δ';
$\forall j \in \{0..k\}$. *finite* $(\{\varrho.\ \eta\ \varrho = \text{Some}\ j \wedge \varrho \in \text{dom}\ \Delta\})$;
Delta-ge Δ' *si k* $\eta\ \delta$; *projection* $f\ \Delta' \sqsubseteq_\Delta\ \Delta]\!]$
$\Longrightarrow$ *Delta-ge* $(\Delta(\varrho\text{self-}f\ f \mapsto \text{the}\ (\Delta'\ (\varrho\text{self-}f\ f))))$ *si k* $\eta\ \delta$
apply (*simp add*: *Delta-ge-def*)
apply (*rule ballI*)
apply (*erule-tac x=j* **in** *ballE*,*simp*)
apply (*frule-tac* $\eta=\eta$ **and** *j=j* **and** *si=si* **in** *sizeAbstractDelta-si-le*,*assumption*+)

apply *force*
by *simp*

lemma *finite-dom-Δ-finite-η*:
finite (*dom* Δ)
$\Longrightarrow$ $(\forall j \in \{0..k\}$. *finite* $(\{\varrho.\ \eta\ \varrho = \text{Some}\ j \wedge \varrho \in \text{dom}\ \Delta\}))$
apply (*rule ballI*)
apply (*rule-tac B=*$\{\varrho.\ \varrho \in \text{dom}\ \Delta\}$ **in** *finite-subset*)
by (*blast*,*simp*)

lemma *resources-le-SafeResourcesDAssDepth*:
$[\![$ (*projection* $f\ \Delta'$, μ' , σ') $\sqsubseteq_R$ (Δ , μ , σ);
finite (*dom* Δ);
bodyAPP $\Sigma d\ f :_{f\ (\text{typesAPP}\ \Sigma\vartheta\ f)\ (\text{sizesAPP}\ \Sigma\Phi\ f)\ td,\ n}$ $\{\!|\ \Delta'$, μ' , $\sigma'\ |\!\}\ ]\!]$
$\Longrightarrow$ *bodyAPP* $\Sigma d\ f :_{f\ (\text{typesAPP}\ \Sigma\vartheta\ f)\ (\text{sizesAPP}\ \Sigma\Phi\ f)\ td,\ n}$ $\{\!|\ \Delta((\varrho\text{self-}f\ f) \mapsto$

the (Δ' (ϱ*self-f f*))), μ , σ $\|$
apply (*simp only*: *resources-le-def*)
apply (*simp only*: *typesAPP-def*)
apply (*simp only*: *SafeResourcesDAssDepth.simps*)
apply (*elim conjE*)
apply (*rule impI*)
apply (*rule conjI*,*simp*)+
apply (*rule dom-delta'-equals-dom-delta-*ϱ*self*,*simp*,*simp*)
apply (*drule mp*,*simp*)
apply (*elim conjE*)
apply (*rule allI*)+
apply (*erule-tac x*=*E1* **in** *allE*)
apply (*erule-tac x*=*E2* **in** *allE*)
apply (*erule-tac x*=*h* **in** *allE*)
apply (*erule-tac x*=*k* **in** *allE*)
apply (*erule-tac x*=*hh* **in** *allE*)
apply (*erule-tac x*=*v* **in** *allE*)
apply (*erule-tac x*=δ **in** *allE*)
apply (*erule-tac x*=*m* **in** *allE*)
apply (*erule-tac x*=*s* **in** *allE*)
apply (*erule-tac x*=η **in** *allE*)
apply (*erule-tac x*=*si* **in** *allE*)
apply (*rule impI*)
apply (*elim conjE*)
apply (*drule mp*)
apply (*rule conjI*,*simp*)+
apply (*subst dom-delta'-equals-dom-delta-*ϱ*self*,*simp*,*simp*,*simp*)
apply (*rule conjI*,*simp*)
apply *simp*
apply (*elim conjE*)
apply (*frule-tac* η=η **and** *k*=*k* **in** *finite-dom-*Δ*-finite-*η)
apply (*rule conjI*, *rule resources-le-Delta-ge*,*simp*,*simp*,*simp*,*simp*)
apply (*rule conjI*, *rule resources-le-Mu-ge*,*simp*,*simp*)
by (*rule resources-le-Sigma-ge*,*simp*,*simp*)

lemma *projection-prop2*:
(ϱ*self-f f*) $\notin$ *dom* Δ
$\Longrightarrow$ Δ = *projection f* (Δ(ϱ*self-f f* $\mapsto$ *the* (Δ' (ϱ*self-f f*))))
apply (*rule map-le-antisym*)
apply (*simp add*: *projection-def*)
apply (*simp add*: *map-le-def*,*clarsimp*)
apply (*rule sym*)
apply (*subst map-upd-Some-unfold*,*simp*)
apply (*simp add*: *projection-def*)
apply (*simp add*: *map-le-def*,*clarsimp*)
by (*subst* (*asm*) *map-upd-Some-unfold*,*simp*)

lemma *lemma-8-REC* [*rule-format*]:
(∀ *n*. (*ValidGlobalResourcesEnvDepth f n* (Σ*b*(*f* ↦ (Δ, *μ*, *σ*))))
⟶ *SafeResourcesDAssDepth* (*bodyAPP* Σ*d f*) *f* (*typesAPP* Σ*ϑ f*) (*sizesAPP* Σ*Φ f*)
(*real* (*length* (*varsAPP* Σ*d f*)) + *real* (*length* (*regionsAPP* Σ*d f*))) *n* Δ′ *μ*′ *σ*′)
⟶ *f* ∉ *dom* Σ*b*
⟶ (*ϱself-f f*) ∉ *dom* Δ
⟶ *finite* (*dom* Δ)
⟶ (*projection f* Δ′ , *μ*′ , *σ*′) ⊑$_R$ (Δ , *μ* , *σ*)
⟶ *ValidGlobalResourcesEnvDepth f n* Σ*b*
⟶ *SafeResourcesDAssDepth* (*bodyAPP* Σ*d f*) *f* (*typesAPP* Σ*ϑ f*) (*sizesAPP* Σ*Φ f*)
(*real* (*length* (*varsAPP* Σ*d f*)) + *real* (*length* (*regionsAPP* Σ*d f*))) *n* Δ′ *μ*′ *σ*′
apply (*rule impI*)
apply (*induct-tac n*)

apply (*rule impI*)+
apply (*erule-tac x=0* **in** *allE*)
apply (*frule imp-ValidResourcesDepth-n-SigmaResources-Valid-Sigma*,*assumption*+)
apply (*subgoal-tac* ⊨⊨$_{f,0}$ Σ*b*(*f* ↦ (Δ, *μ*, *σ*)),*simp*)
apply (*subst projection-prop2*,*assumption*)
apply (*rule ValidGlobalResourcesEnvDepth.depth0*,*assumption*+)

apply (*erule-tac x=Suc n* **in** *allE*)
apply (*rule impI*)+
apply (*drule mp*,*simp*)
apply (*frule imp-ValidResourcesDepth-n-SigmaResources-Valid-Sigma*,*assumption*+)
apply (*frule-tac f=f* **in** *imp-ValidGlobalResourcesEnv-all-n-ValidGlobalResourcesEnvDepth*)
apply (*erule-tac x=n* **in** *allE*,*simp*)
apply (*frule resources-le-SafeResourcesDAssDepth*,*assumption*+)
apply (*subgoal-tac* ⊨⊨$_{f,Suc\ n}$ Σ*b*(*f* ↦ (Δ, *μ*, *σ*)),*simp*)
apply (*subst projection-prop2*,*assumption*)
by (*rule ValidGlobalResourcesEnvDepth.step*,*simp-all*)

lemma *lemma-8*:
e, Σ*b* ⊢$_f$ (*typesAPP* Σ*ϑ f*) (*sizesAPP* Σ*Φ f*) *td* (Δ,*μ*,*σ*)
⟹ ∀ *n*. *e* , Σ*b* :$_f$ (*typesAPP* Σ*ϑ f*) (*sizesAPP* Σ*Φ f*) *td*,*n* {| Δ , *μ* , *σ* |}
apply (*simp only*: *typesAPP-def*)
apply (*erule ProofRulesCostes.induct*)

apply (*simp only*: *SafeResourcesDAssDepthCntxt.simps*)
apply (*rule allI*, *rule impI*)
apply (*rule SafeResourcesDADepth-LitInt*)

apply (*simp only*: *SafeResourcesDAssDepthCntxt.simps*)
apply (*rule allI*, *rule impI*)
apply (*rule SafeResourcesDADepth-LitBool*)

apply (*simp only*: *SafeResourcesDAssDepthCntxt.simps*)
apply (*rule allI*, *rule impI*)
apply (*rule SafeResourcesDADepth-Var1*)

apply (*simp only*: *typesAPP-def*)
apply (*simp only*: *SafeResourcesDAssDepthCntxt.simps*)
apply (*rule allI*, *rule impI*)
apply (*rule SafeResourcesDADepth-Var2*,*force*,*force*)

apply (*simp only*: *typesAPP-def*)
apply (*simp only*: *SafeResourcesDAssDepthCntxt.simps*)
apply (*rule allI*, *rule impI*)
apply (*rule SafeResourcesDADepth-Var3*,*force*,*force*)

apply (*rule allI*)
apply (*simp only*: *SafeResourcesDAssDepthCntxt.simps*)
apply (*rule impI*)
apply (*erule-tac x*=*n* **in** *allE*)+
apply (*drule mp*, *simp*)+
apply (*rule SafeResourcesDADepth-Let1*)
apply (*simp*,*assumption*+,*simp*,*simp*,*simp*)

apply (*rule allI*)
apply (*simp only*: *typesAPP-def*)
apply (*simp only*: *SafeResourcesDAssDepthCntxt.simps*)
apply (*rule impI*)
apply (*erule-tac x*=*n* **in** *allE*)+
apply (*drule mp*, *simp*)+
apply (*rule SafeResourcesDADepth-Let2*)
apply *assumption*+

apply (*rule allI*)
apply (*simp only*: *SafeResourcesDAssDepthCntxt.simps*)
apply (*rule impI*)
apply (*rule SafeResourcesDADepth-CASE*)
apply (*assumption*+,*simp*,*assumption*+,*simp*,*simp*,*simp*,*simp*)

apply (*rule allI*)
apply (*simp only*: *SafeResourcesDAssDepthCntxt.simps*)
apply (*rule impI*)
apply (*rule SafeResourcesDADepth-APP-PRIMOP*)
apply (*assumption+*)

apply (*rule allI*)
apply (*simp only*: *typesAPP-def*)
apply (*simp only*: *SafeResourcesDAssDepthCntxt.simps*)
apply (*rule impI*)
apply (*fold typesAPP-def*)
apply (*frule-tac* $\sigma g{=}\sigma g$ **in** *SafeResourcesDADepth-APP*)
apply (*assumption+*,*simp*,*assumption+*,*simp*)

apply (*simp only*: *typesAPP-def*)
apply (*simp only*: *SafeResourcesDAssDepthCntxt.simps*)
apply (*rule allI*)
apply (*rule impI*)
apply (*subgoal-tac*
$ef = (bodyAPP\ \Sigma d\ f) \wedge$
$xs = (varsAPP\ \Sigma d\ f) \wedge$
$rs = (regionsAPP\ \Sigma d\ f)$,*simp*)
apply (*fold typesAPP-def*)
apply (*rule-tac* $f{=}f$ **and** $\Sigma b{=}\Sigma b$ **in** *lemma-8-REC*)
apply (*simp only*: *typesAPP-def*)
apply (*force*,*simp*,*simp*,*simp*,*simp*,*simp*)
apply (*simp add*: *bodyAPP-def*)
apply (*simp add*: *varsAPP-def*)
apply (*simp add*: *regionsAPP-def*)
done

lemma *lemma-2*:
$e, \Sigma b \vdash_f (typesAPP\ \Sigma\vartheta\ f)\ (sizesAPP\ \Sigma\Phi\ f)\ \ td\ (\Delta,\mu,\sigma)$
$\Longrightarrow SafeResourcesDAssCntxt\ e\ \Sigma b\ f\ \ (typesAPP\ \Sigma\vartheta\ f)\ (sizesAPP\ \Sigma\Phi\ f)\ \ td\ \Delta\ \mu\ \sigma$
apply (*rule lemma-7*)
by (*rule lemma-8*,*assumption*)

end