Verification of the `UpDown` scheme

Johannes Hölzl

16. Juni 2019

Zusammenfassung

The `UpDown` scheme is a recursive scheme used to compute the stiffness matrix on a special form of sparse grids. Usually, when discretizing a Euclidean space of dimension d we need $O(n^d)$ points, for n points along each dimension. Sparse grids are a hierarchical representation where the number of points is reduced to $O(n \cdot \log(n)^d)$. One disadvantage of such sparse grids is that the algorithm now operate recursively in the dimensions and levels of the sparse grid.

The `UpDown` scheme allows us to compute the stiffness matrix on such a sparse grid. The stiffness matrix represents the influence of each representation function on the L^2 scalar product. For a detailed description see Pflüger's PhD thesis [2]. This formalization was developed as an interdisciplinary project (IDP) at the TU München [1].

Note: This development has two main theories. The correctnes of the UpDown scheme, and a verification of an imperative version of it. Both theories can not be merged, as they use different orders on the product type.

Inhaltsverzeichnis

1 Grid Points

theory *Grid-Point*
imports *HOL−Analysis.Analysis*
begin

type-synonym *grid-point* = (*nat* × *int*) *list*

definition *lv* :: *grid-point* ⇒ *nat* ⇒ *nat*
  where *lv p d* = *fst* (*p* ! *d*)

definition *ix* :: *grid-point* ⇒ *nat* ⇒ *int*
  where *ix p d* = *snd* (*p* ! *d*)

definition *level* :: *grid-point* ⇒ *nat*
  where *level p* = (∑ *i* < *length p*. *lv p i*)

lemma *level-all-eq*:
  assumes ⋀*d*. *d* < *length p* ⟹ *lv p d* = *lv p*′ *d*
  and *length p* = *length p*′
  shows *level p*′ = *level p*
  unfolding *level-def* **using** *assms* **by** *auto*

datatype *dir* = *left* | *right*

fun *sgn* :: *dir* ⇒ *int*
where
    sgn left = −*1*
  | *sgn right* = *1*

fun *inv* :: *dir* ⇒ *dir*
where
    inv left = *right*
  | *inv right* = *left*

lemma *inv-inv*[*simp*]: *inv* (*inv dir*) = *dir*
  by (*cases dir*) *simp-all*

lemma *sgn-inv*[*simp*]: *sgn* (*inv dir*) = − *sgn dir*

by (*cases dir*, *auto*)

definition *child* :: *grid-point* ⇒ *dir* ⇒ *nat* ⇒ *grid-point*
where *child p dir d* = *p*[*d* := (*lv p d* + *1*, *2* ∗ (*ix p d*) + *sgn dir*)]

lemma *child-length*[*simp*]: *length* (*child p dir d*) = *length p*
unfolding *child-def* **by** *simp*

lemma *child-odd*[*simp*]: *d* < *length p* ⟹ *odd* (*ix* (*child p dir d*) *d*)
unfolding *child-def ix-def* **by** (*cases dir*, *auto*)

lemma *child-eq*: *p* ! *d* = (*l*, *i*) ⟹ ∃ *j*. *child p dir d* = *p*[*d* := (*l* + *1*, *j*)]
by (*auto simp add*: *child-def ix-def lv-def*)

lemma *child-other*: *d* ≠ *d'* ⟹ *child p dir d* ! *d'* = *p* ! *d'*
unfolding *child-def lv-def ix-def* **by** (*cases d'* < *length p*, *auto*)

lemma *child-invariant*: **assumes** *d'* < *length p* **shows** (*child p dir d* ! *d'* = *p* ! *d'*) = (*d* ≠ *d'*)
proof −
obtain *l i* **where** *p* ! *d'* = (*l*, *i*) **using** *prod.exhaust* **.**
with *assms* **show** *?thesis*
unfolding *child-def ix-def lv-def* **by** *auto*
qed

lemma *child-single-level*: *d* < *length p* ⟹ *lv* (*child p dir d*) *d* > *lv p d*
unfolding *lv-def child-def* **by** *simp*

lemma *child-lv*: *d* < *length p* ⟹ *lv* (*child p dir d*) *d* = *lv p d* + *1*
unfolding *child-def lv-def* **by** *simp*

lemma *child-lv-other*: **assumes** *d'* ≠ *d* **shows** *lv* (*child p dir d'*) *d* = *lv p d*
using *child-other*[*OF assms*] **unfolding** *lv-def* **by** *simp*

lemma *child-ix-left*: *d* < *length p* ⟹ *ix* (*child p left d*) *d* = *2* ∗ *ix p d* − *1*
unfolding *child-def ix-def* **by** *simp*

lemma *child-ix-right*: *d* < *length p* ⟹ *ix* (*child p right d*) *d* = *2* ∗ *ix p d* + *1*
unfolding *child-def ix-def* **by** *simp*

lemma *child-ix*: *d* < *length p* ⟹ *ix* (*child p dir d*) *d* = *2* ∗ *ix p d* + *sgn dir*
unfolding *child-def ix-def* **by** *simp*

lemma *child-level*[*simp*]: **assumes** *d* < *length p*
shows *level* (*child p dir d*) = *level p* + *1*
proof −
have *inter*: {*0*..<*length p*} ∩ {*d*} = {*d*} **using** *assms* **by** *auto*

have *level* (*child p dir d*) =

```
    (∑ d' = 0..<length p. if d' ∈ {d} then lv p d + 1 else lv p d')
    by (auto intro!: sum.cong simp add: child-lv-other child-lv level-def)
  moreover have level p + 1 =
    (∑ d' = 0..<length p. if d' ∈ {d} then lv p d else lv p d') + 1
    by (auto intro!: sum.cong simp add: child-lv-other child-lv level-def)
  ultimately show ?thesis
    unfolding sum.If-cases[OF finite-atLeastLessThan] inter
    using assms by auto
qed

lemma child-ex-neighbour: ∃ b'. child b dir d = child b' (inv dir) d
proof
  show child b dir d = child (b[d := (lv b d, ix b d + sgn dir)]) (inv dir) d
     unfolding child-def ix-def lv-def by (cases d < length b, auto simp add:
algebra-simps)
qed

lemma child-level-gt[simp]: level (child p dir d) >= level p
  by (cases d < length p, simp, simp add: child-def)

lemma child-estimate-child:
  assumes d < length p
    and l ≤ lv p d
    and i'-range: ix p d < (i + 1) * 2^(lv p d - l) ∧
                  ix p d > (i - 1) * 2^(lv p d - l)
                 (is ?top p ∧ ?bottom p)
    and is-child: p' = child p dir d
  shows ?top p' ∧ ?bottom p'
proof
  from is-child and ‹d < length p›
  have lv p' d = lv p d + 1 by (auto simp add: child-def ix-def lv-def)
  hence lv p' d - l = lv p d - l + 1 using ‹lv p d >= l› by auto
  hence pow-l'': (2^(lv p' d - l) :: int) = 2 * 2^(lv p d - l) by auto

  show ?top p'
  proof -
    from is-child and ‹d < length p›
    have ix p' d ≤ 2 * (ix p d) + 1
      by (cases dir, auto simp add: child-def lv-def ix-def)
    also have ... < (i + 1) * (2 * 2^(lv p d - l)) using i'-range by auto
    finally show ?thesis using pow-l'' by auto
  qed

  show ?bottom p'
  proof -
    have (i - 1) * 2^(lv p' d - l) = 2 * ((i - 1) * 2^(lv p d - l))
      using pow-l'' by simp
    also have ... < 2 * ix p d - 1 using i'-range by auto
    finally show ?thesis using is-child and ‹d < length p›
```

```
    by (cases dir, auto simp add: child-def lv-def ix-def)
  qed
qed

lemma child-neighbour: assumes child p (inv dir) d = child ps dir d (is ?c-p = ?c-ps)
  shows ps = p[d := (lv p d, ix p d − sgn dir)] (is - = ?ps)
proof (rule nth-equalityI)
  have length ?c-ps = length ?c-p using ‹?c-p = ?c-ps› by simp
  hence len-eq: length ps = length p by simp
  thus length ps = length ?ps by simp

  show ps ! i = ?ps ! i if i < length ps for i
  proof −
    have i < length p
      using that len-eq by auto

    show ps ! i = ?ps ! i
    proof (cases d = i)
      case [simp]: True

      have ?c-p ! i = ?c-ps ! i using ‹?c-p = ?c-ps› by auto
      hence ix p i = ix ps d + sgn dir and lv p i = lv ps i
        by (auto simp add: child-def
          nth-list-update-eq[OF ‹i < length p›]
          nth-list-update-eq[OF ‹i < length ps›])
      thus ?thesis by (simp add: lv-def ix-def ‹i < length p›)
    next
      assume d ≠ i
      with child-other[OF this, of ps dir] child-other[OF this, of p inv dir]
      show ?thesis using assms by auto
    qed
  qed
qed

definition start :: nat ⇒ grid-point
where
  start dm = replicate dm (0, 1)

lemma start-lv[simp]: d < dm ⟹ lv (start dm) d = 0
  unfolding start-def lv-def by simp

lemma start-ix[simp]: d < dm ⟹ ix (start dm) d = 1
  unfolding start-def ix-def by simp

lemma start-length[simp]: length (start dm) = dm
  unfolding start-def by auto

lemma level-start-0[simp]: level (start dm) = 0
```

using *level-def* **by** *auto*

end

2 Sparse Grids

theory *Grid*
imports *Grid-Point*
begin

2.1 Vectors

type-synonym *vector* = *grid-point* ⇒ *real*

definition *null-vector* :: *vector*
where *null-vector* ≡ λ *p*. *0*

definition *sum-vector* :: *vector* ⇒ *vector* ⇒ *vector*
where *sum-vector* α β ≡ λ *p*. α *p* + β *p*

2.2 Inductive enumeration of all grid points

inductive-set
  grid :: *grid-point* ⇒ *nat set* ⇒ *grid-point set*
  for *b* :: *grid-point* **and** *ds* :: *nat set*
where
    Start[*intro*!]: *b* ∈ *grid b ds*
  | *Child*[*intro*!]: ⟦ *p* ∈ *grid b ds* ; *d* ∈ *ds* ⟧ ⟹ *child p dir d* ∈ *grid b ds*

lemma *grid-length*[*simp*]: *p'* ∈ *grid p ds* ⟹ *length p'* = *length p*
  by (*erule grid.induct*, *auto*)

lemma *grid-union-dims*: ⟦ *ds* ⊆ *ds'* ; *p* ∈ *grid b ds* ⟧ ⟹ *p* ∈ *grid b ds'*
  by (*erule grid.induct*, *auto*)

lemma *grid-transitive*: ⟦ *a* ∈ *grid b ds* ; *b* ∈ *grid c ds'* ; *ds'* ⊆ *ds''* ; *ds* ⊆ *ds''* ⟧
⟹ *a* ∈ *grid c ds''*
  by (*erule grid.induct*, *auto simp add*: *grid-union-dims*)

lemma *grid-child*[*intro?*]: **assumes** *d* ∈ *ds* **and** *p-grid*: *p* ∈ *grid* (*child b dir d*)
ds
  shows *p* ∈ *grid b ds*
  using ‹*d* ∈ *ds*› *grid-transitive*[*OF p-grid*] **by** *auto*

lemma *grid-single-level*[*simp*]: **assumes** *p* ∈ *grid b ds* **and** *d* < *length b*
  shows *lv b d* ≤ *lv p d*
  using *assms*
proof *induct*
  case (*Child p' d' dir*)

```
  thus ?case by (cases d' = d, auto simp add: child-def ix-def lv-def)
qed auto

lemma grid-child-level:
  assumes d < length b
  and p-grid: p ∈ grid (child b dir d) ds
  shows lv b d < lv p d
proof −
  have lv b d < lv (child b dir d) d using child-lv[OF ‹d < length b›] by auto
  also have ... ≤ lv p d using p-grid assms by (intro grid-single-level) auto
  finally show ?thesis .
qed

lemma child-out: length p ≤ d ⟹ child p dir d = p
  unfolding child-def by auto

lemma grid-dim-remove:
  assumes inset: p ∈ grid b ({d} ∪ ds)
  and eq: d < length b ⟹ p ! d = b ! d
  shows p ∈ grid b ds
  using inset eq
proof induct
  case (Child p' d' dir)
  show ?case
  proof (cases d' ≥ length p')
    case True with child-out[OF this]
    show ?thesis using Child by auto
  next
    case False hence d' < length p' by simp
    show ?thesis
    proof (cases d' = d)
      case True
      hence lv b d ≤ lv p' d and lv p' d < lv (child p' dir d) d
        using child-single-level Child ‹d' < length p'› by auto
      hence False using Child and ‹d' = d› and lv-def and ‹¬ d' ≥ length p'› by
auto
      thus ?thesis ..
    next
      case False
      hence d' ∈ ds using Child by auto
      moreover have d < length b ⟹ p' ! d = b ! d
      proof −
        assume d < length b
        hence d < length p' using Child by auto
        hence child p' dir d' ! d = p' ! d using child-invariant and False by auto
        thus ?thesis using Child and ‹d < length b› by auto
      qed
      hence p' ∈ grid b ds using Child by auto
      ultimately show ?thesis using grid.Child by auto
```

```
    qed
  qed
qed auto

lemma gridgen-dim-restrict:
  assumes inset: p ∈ grid b (ds' ∪ ds)
  and eq: ∀ d ∈ ds'. d ≥ length b
  shows p ∈ grid b ds
  using inset eq
proof induct
  case (Child p' d dir)
  thus ?case
  proof (cases d ∈ ds)
    case False thus ?thesis using Child and child-def by auto
  qed auto
qed auto

lemma grid-dim-remove-outer: grid b ds = grid b {d ∈ ds. d < length b}
proof
  have {d ∈ ds. d < length b} ⊆ ds by auto
  from grid-union-dims[OF this]
  show grid b {d ∈ ds. d < length b} ⊆ grid b ds by auto

  have ds = (ds − {d ∈ ds. d < length b}) ∪ {d ∈ ds. d < length b} by auto
  moreover
  have grid b ((ds − {d ∈ ds. d < length b}) ∪ {d ∈ ds. d < length b}) ⊆ grid b
{d ∈ ds. d < length b}
  proof
    fix p
    assume p ∈ grid b (ds − {d ∈ ds. d < length b} ∪ {d ∈ ds. d < length b})
    moreover have ∀ d ∈ (ds − {d ∈ ds. d < length b}). d ≥ length b by auto
   ultimately show p ∈ grid b {d ∈ ds. d < length b} by (rule gridgen-dim-restrict)
  qed
  ultimately show grid b ds ⊆ grid b {d ∈ ds. d < length b} by auto
qed

lemma grid-level[intro]: assumes p ∈ grid b ds shows level b ≤ level p
proof −
  have ∗: length p = length b using grid-length assms by auto
  { fix i assume i ∈ {0 ..< length p}
    hence lv b i ≤ lv p i using ‹p ∈ grid b ds› and grid-single-level ∗ by auto
  } thus ?thesis unfolding level-def ∗ by (auto intro!: sum-mono)
qed
lemma grid-empty-ds[simp]: grid b {} = { b }
proof −
  have !! z. z ∈ grid b {} ⟹ z = b
    by (erule grid.induct, auto)
  thus ?thesis by auto
qed
```

lemma *grid-Start*: **assumes** *inset*: $p \in grid\ b\ ds$ **and** *eq*: $level\ p = level\ b$ **shows** $p = b$
using *inset eq*
proof *induct*
case (*Child p d dir*)
show *?case*
proof (*cases* $d < length\ b$)
case *True*
from *Child*
have $level\ p \geq level\ b$ **by** *auto*
moreover
have $level\ p \leq level\ (child\ p\ dir\ d)$ **by** (*rule child-level-gt*)
hence $level\ p \leq level\ b$ **using** *Child* **by** *auto*
ultimately have $level\ p = level\ b$ **by** *auto*
hence $p = b$ **using** *Child*(*2*) **by** *auto*
with *Child*(*4*) **have** $level\ (child\ b\ dir\ d) = level\ b$ **by** *auto*
moreover have $level\ (child\ b\ dir\ d) \neq level\ b$ **using** *child-level* **and** ‹$d < length\ b$› **by** *auto*
ultimately show *?thesis* **by** *auto*
next
case *False*
with *Child* **have** $length\ p = length\ b$ **by** *auto*
with *False* **have** $child\ p\ dir\ d = p$ **using** *child-def* **by** *auto*
moreover with *Child* **have** $level\ p = level\ b$ **by** *auto*
with *Child*(*2*) **have** $p = b$ **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
qed *auto*
lemma *grid-estimate*:
assumes $d < length\ b$ **and** *p-grid*: $p \in grid\ b\ ds$
shows $ix\ p\ d < (ix\ b\ d + 1) * 2\hat{}(lv\ p\ d - lv\ b\ d) \wedge ix\ p\ d > (ix\ b\ d - 1) * 2\hat{}(lv\ p\ d - lv\ b\ d)$
using *p-grid*
proof *induct*
case (*Child p d' dir*)
show *?case*
proof (*cases* $d = d'$)
case *False* **with** *Child* **show** *?thesis* **unfolding** *child-def lv-def ix-def* **by** *auto*
next
case *True* **with** *child-estimate-child* **and** *Child* **and** ‹$d < length\ b$›
show *?thesis* **using** *grid-single-level* **by** *auto*
qed
qed *auto*
lemma *grid-odd*: **assumes** $d < length\ b$ **and** *p-diff*: $p\ !\ d \neq b\ !\ d$ **and** *p-grid*: $p \in grid\ b\ ds$
shows $odd\ (ix\ p\ d)$
using *p-grid* **and** *p-diff*
proof *induct*
case (*Child p d' dir*)

show *?case*
proof (*cases* $d = d'$)
case *True* **with** *child-odd* **and** ‹$d < length\ b$› **and** *Child* **show** *?thesis* **by** *auto*
next
case *False* **with** *Child* **and** ‹$d < length\ b$› **show** *?thesis* **using** *child-def* **and** *ix-def* **and** *lv-def* **by** *auto*
qed
qed *auto*
lemma *grid-invariant*: **assumes** $d < length\ b$ **and** $d \notin ds$ **and** *p-grid*: $p \in grid\ b\ ds$
shows $p\ !\ d = b\ !\ d$
using *p-grid*
proof (*induct*)
case (*Child p d' dir*) **hence** $d' \neq d$ **using** ‹$d \notin ds$› **by** *auto*
thus *?case* **using** *child-def* **and** *Child* **by** *auto*
qed *auto*
lemma *grid-part*: **assumes** $d < length\ b$ **and** *p-valid*: $p \in grid\ b\ \{d\}$ **and** *p'-valid*: $p' \in grid\ b\ \{d\}$
and *level*: $lv\ p'\ d \geq lv\ p\ d$
and *right*: $ix\ p'\ d \leq (ix\ p\ d + 1) * 2\hat{}(lv\ p'\ d - lv\ p\ d)$ (**is** *?right p p' d*)
and *left*: $ix\ p'\ d \geq (ix\ p\ d - 1) * 2\hat{}(lv\ p'\ d - lv\ p\ d)$ (**is** *?left p p' d*)
shows $p' \in grid\ p\ \{d\}$
using *p'-valid left right level* **and** *p-valid*
proof *induct*
case (*Child p' d' dir*)
hence $d = d'$ **by** *auto*
let *?child* = *child p' dir d'*

show *?case*
proof (*cases lv p d = lv ?child d*)
case *False*
moreover have *lv ?child d* = *lv p' d + 1* **using** *child-lv* **and** ‹$d < length\ b$› **and** *Child* **and** ‹$d = d'$› **by** *auto*
ultimately have *lv p d < lv p' d + 1* **using** *Child* **by** *auto*
hence *lv*: *Suc* (*lv p' d*) − *lv p d* = *Suc* (*lv p' d* − *lv p d*) **by** *auto*

have *?left p p' d* ∧ *?right p p' d*
proof (*cases dir*)
case *left*
with *Child* **have** $2 * ix\ p'\ d - 1 \leq (ix\ p\ d + 1) * 2\hat{}(Suc\ (lv\ p'\ d) - lv\ p\ d)$
using ‹$d = d'$› **and** ‹$d < length\ b$› **by** (*auto simp add*: *child-def ix-def lv-def*)
also have ... = $2 * (ix\ p\ d + 1) * 2\hat{}(lv\ p'\ d - lv\ p\ d)$ **using** *lv* **by** *auto*
finally have $2 * ix\ p'\ d - 2 < 2 * (ix\ p\ d + 1) * 2\hat{}(lv\ p'\ d - lv\ p\ d)$ **by** *auto*
also have ... = $2 * ((ix\ p\ d + 1) * 2\hat{}(lv\ p'\ d - lv\ p\ d))$ **by** *auto*
finally have *left-r*: $ix\ p'\ d \leq (ix\ p\ d + 1) * 2\hat{}(lv\ p'\ d - lv\ p\ d)$ **by** *auto*

have $2 * ((ix\ p\ d - 1) * 2\hat{}(lv\ p'\ d - lv\ p\ d)) = 2 * (ix\ p\ d - 1) * 2\hat{}(lv\ p'\ d - lv\ p\ d)$ **by** *auto*

```
    also have ... = (ix p d − 1) * 2^(Suc (lv p' d) − lv p d) using lv by auto
    also have ... ≤ 2 * ix p' d − 1
      using left and Child and ‹d = d'› and ‹d < length b› by (auto simp add:
child-def ix-def lv-def)
    finally have right-r: ((ix p d − 1) * 2^(lv p' d − lv p d)) ≤ ix p' d by auto

    show ?thesis using left-r and right-r by auto
  next
    case right
    with Child have 2 * ix p' d + 1 ≤ (ix p d + 1) * 2^(Suc (lv p' d) − lv p d)
     using ‹d = d'› and ‹d < length b› by (auto simp add: child-def ix-def lv-def)
    also have ... = 2 * (ix p d + 1) * 2^(lv p' d − lv p d) using lv by auto
    finally have 2 * ix p' d < 2 * (ix p d + 1) * 2^(lv p' d − lv p d) by auto
    also have ... = 2 * ((ix p d + 1) * 2^(lv p' d − lv p d)) by auto
    finally have left-r: ix p' d ≤ (ix p d + 1) * 2^(lv p' d − lv p d) by auto

    have 2 * ((ix p d − 1) * 2^(lv p' d − lv p d)) = 2 * (ix p d − 1) * 2^(lv p'
d − lv p d) by auto
    also have ... = (ix p d − 1) * 2^(Suc (lv p' d) − lv p d) using lv by auto
    also have ... ≤ 2 * ix p' d + 1
     using right and Child and ‹d = d'› and ‹d < length b› by (auto simp add:
child-def ix-def lv-def)
    also have ... < 2 * (ix p' d + 1) by auto
    finally have right-r: ((ix p d − 1) * 2^(lv p' d − lv p d)) ≤ ix p' d by auto

    show ?thesis using left-r and right-r by auto
  qed
  with Child and lv have p' ∈ grid p {d} by auto
  thus ?thesis using ‹d = d'› by auto
next
  case True
  moreover with Child have ?left p ?child d ∧ ?right p ?child d by auto
  ultimately have range: ix p d − 1 ≤ ix ?child d ∧ ix ?child d ≤ ix p d + 1
by auto

  have p ! d ≠ b ! d
  proof (rule ccontr)
    assume ¬ (p ! d ≠ b ! d)
     with ‹lv p d = lv ?child d› have lv b d = lv ?child d by (auto simp add:
lv-def)
    hence lv b d = lv p' d + 1 using ‹d = d'› and Child and ‹d < length b›
and child-lv by auto
    moreover have lv b d ≤ lv p' d using ‹d = d'› and Child and ‹d < length
b› and grid-single-level by auto
    ultimately show False by auto
  qed
  hence odd (ix p d) using grid-odd and ‹p ∈ grid b {d}› and ‹d < length b›
by auto
  hence ¬ odd (ix p d + 1) and ¬ odd (ix p d − 1) by auto
```

```
    have d < length p' using ‹p' ∈ grid b {d}› and ‹d < length b› by auto
    hence odd-child: odd (ix ?child d) using child-odd and ‹d = d'› by auto

    have ix p d − 1 ≠ ix ?child d
    proof (rule ccontr)
      assume ¬ (ix p d − 1 ≠ ix ?child d)
      hence odd (ix p d − 1) using odd-child by auto
      thus False using ‹¬ odd (ix p d − 1)› by auto
    qed
    moreover
    have ix p d + 1 ≠ ix ?child d
    proof (rule ccontr)
      assume ¬ (ix p d + 1 ≠ ix ?child d)
      hence odd (ix p d + 1) using odd-child by auto
      thus False using ‹¬ odd (ix p d + 1)› by auto
    qed
    ultimately have ix p d = ix ?child d using range by auto
    with True have d-eq: p ! d = (?child) ! d by (auto simp add: prod-eqI ix-def
lv-def)

    have length p = length ?child using ‹p ∈ grid b {d}› and ‹p' ∈ grid b {d}› by
auto
    moreover have p ! d'' = ?child ! d'' if d'' < length p for d''
    proof −
      have d'' < length b using that ‹p ∈ grid b {d}› by auto
      show p ! d'' = ?child ! d''
      proof (cases d = d'')
        case True with d-eq show ?thesis by auto
      next
        case False hence d'' ∉ {d} by auto
        from ‹d'' < length b› and this and ‹p ∈ grid b {d}›
        have p ! d'' = b ! d'' by (rule grid-invariant)
        also have ... = p' ! d'' using ‹d'' < length b› and ‹d'' ∉ {d}› and ‹p' ∈
grid b {d}›
          by (rule grid-invariant[symmetric])
        also have ... = ?child ! d''
        proof −
          have d'' < length p' using ‹d'' < length b› and ‹p' ∈ grid b {d}› by auto
          hence ?child ! d'' = p' ! d'' using child-invariant and ‹d ≠ d''› and ‹d
= d'› by auto
          thus ?thesis by auto
        qed
        finally show ?thesis .
      qed
    qed
    ultimately have p = ?child by (rule nth-equalityI)
    thus ?child ∈ grid p {d} by auto
  qed
```

```
next
  case Start
  moreover hence lv b d ≤ lv p d using grid-single-level and ‹d < length b› by
auto
  ultimately have lv b d = lv p d by auto

  have level p = level b
  proof −
    { fix d'
      assume d' < length b
      have lv b d' = lv p d'
      proof (cases d = d')
        case True with ‹lv b d = lv p d› show ?thesis by auto
      next
        case False hence d' ∉ {d} by auto
        from ‹d' < length b› and this and ‹p ∈ grid b {d}›
        have p ! d' = b ! d' by (rule grid-invariant)
        thus ?thesis by (auto simp add: lv-def)
      qed }
    moreover have length b = length p using ‹p ∈ grid b {d}› by auto
    ultimately show ?thesis by (rule level-all-eq)
  qed
  hence p = b using grid-Start and ‹p ∈ grid b {d}› by auto
  thus ?case by auto
qed
lemma grid-disjunct: assumes d < length p
  shows grid (child p left d) ds ∩ grid (child p right d) ds = {}
  (is grid ?l ds ∩ grid ?r ds = {})
proof (intro set-eqI iffI)
  fix x
  assume x ∈ grid ?l ds ∩ grid ?r ds
  hence ix x d < (ix ?l d + 1) * 2^(lv x d − lv ?l d)
    and ix x d > (ix ?r d − 1) * 2^(lv x d − lv ?r d)
    using grid-estimate ‹d < length p› by auto
  thus x ∈ {} using ‹d < length p› and child-lv and child-ix by auto
qed auto

lemma grid-level-eq: assumes eq: ∀ d ∈ ds. lv p d = lv b d and grid: p ∈ grid b
ds
  shows level p = level b
proof (rule level-all-eq)
  { fix i assume i < length b
    show lv b i = lv p i
    proof (cases i ∈ ds)
      case True with eq show ?thesis by auto
    next case False with ‹i < length b› and grid show ?thesis
        using lv-def ix-def grid-invariant by auto
    qed }
  show length b = length p using grid by auto
```

qed

lemma *grid-partition*:
grid p {d} = {p} ∪ grid (child p left d) {d} ∪ grid (child p right d) {d}
(**is** - = - ∪ *grid ?l {d} ∪ grid ?r {d}*)
proof −
have !! *x*. ⟦ *x ∈ grid p {d}* ; *x ≠ p* ; *x ∉ grid ?r {d}* ⟧ ⟹ *x ∈ grid ?l {d}*
proof (*cases d < length p*)
case *True*
fix *x*

let *?nr-r p = ix x d > (ix p d + 1) * 2 ^ (lv x d − lv p d)*
let *?nr-l p = (ix p d − 1) * 2 ^ (lv x d − lv p d) > ix x d*

have *ix-r-eq*: *ix ?r d = 2 * ix p d + 1* **using** ‹*d < length p*› **and** *child-ix* **by** *auto*
have *lv-r-eq*: *lv ?r d = lv p d + 1* **using** ‹*d < length p*› **and** *child-lv* **by** *auto*

have *ix-l-eq*: *ix ?l d = 2 * ix p d − 1* **using** ‹*d < length p*› **and** *child-ix* **by** *auto*
have *lv-l-eq*: *lv ?l d = lv p d + 1* **using** ‹*d < length p*› **and** *child-lv* **by** *auto*

assume *x ∈ grid p {d}* **and** *x ≠ p* **and** *x ∉ grid ?r {d}*
hence *lv p d ≤ lv x d* **using** *grid-single-level* **and** ‹*d < length p*› **by** *auto*
moreover have *lv p d ≠ lv x d*
proof (*rule ccontr*)
assume ¬ *lv p d ≠ lv x d*
hence *level x = level p* **using** ‹*x ∈ grid p {d}*› **and** *grid-level-eq*[**where** *ds*={*d*}] **by** *auto*
hence *x = p* **using** *grid-Start* **and** ‹*x ∈ grid p {d}*› **by** *auto*
thus *False* **using** ‹*x ≠ p*› **by** *auto*
qed
ultimately have *lv p d < lv x d* **by** *auto*
hence *lv ?r d ≤ lv x d* **and** *?r ∈ grid p {d}* **using** *child-lv* **and** ‹*d < length p*› **by** *auto*
with ‹*d < length p*› **and** ‹*x ∈ grid p {d}*›
have *r-range*: ¬ *?nr-r ?r* ∧ ¬ *?nr-l ?r* ⟹ *x ∈ grid ?r {d}*
using *grid-part*[**where** *p*=*?r* **and** *p′*=*x* **and** *b*=*p* **and** *d*=*d*] **by** *auto*
have *x ∉ grid ?r {d}* ⟹ *?nr-l ?r* ∨ *?nr-r ?r* **by** (*rule ccontr*, *auto simp add*: *r-range*)
hence *?nr-l ?r* ∨ *?nr-r ?r* **using** ‹*x ∉ grid ?r {d}*› **by** *auto*

have *gt0*: *lv x d − lv p d > 0* **using** ‹*lv p d < lv x d*› **by** *auto*

have *ix-shift*: *ix ?r d = ix ?l d + 2* **and** *lv-lr*: *lv ?r d = lv ?l d* **and** *right1*: !! *x* :: *int*. *x + 2 − 1 = x + 1*
using ‹*d < length p*› **and** *child-ix* **and** *child-lv* **by** *auto*

have *lv x d − lv p d = Suc (lv x d − (lv p d + 1))*

```
    using gt0 by auto
  hence lv-shift: !! y :: int. y * 2 ^ (lv x d - lv p d) = y * 2 * 2 ^ (lv x d - (lv
p d + 1))
    by auto

  have ix x d < (ix p d + 1) * 2 ^ (lv x d - lv p d)
    using ⟨x ∈ grid p {d}⟩ grid-estimate and ⟨d < length p⟩ by auto
  also have ... = (ix ?r d + 1) * 2 ^ (lv x d - lv ?r d)
    using ⟨lv p d < lv x d⟩ and ix-r-eq and lv-r-eq lv-shift[where y=ix p d + 1]
by auto
  finally have ?nr-l ?r using ⟨?nr-l ?r ∨ ?nr-r ?r⟩ by auto
  hence r-bound: (ix ?l d + 1) * 2 ^ (lv x d - lv ?l d) > ix x d
    unfolding ix-shift lv-lr using right1 by auto

  have (ix ?l d - 1) * 2 ^ (lv x d - lv ?l d) = (ix p d - 1) * 2 * 2 ^ (lv x d -
(lv p d + 1))
    unfolding ix-l-eq lv-l-eq by auto
  also have ... = (ix p d - 1) * 2 ^ (lv x d - lv p d)
    using lv-shift[where y=ix p d - 1] by auto
  also have  ... < ix x d
    using ⟨x ∈ grid p {d}⟩ grid-estimate and ⟨d < length p⟩ by auto
  finally have l-bound: (ix ?l d - 1) * 2 ^ (lv x d - lv ?l d) < ix x d .

  from l-bound r-bound ⟨d < length p⟩ and ⟨x ∈ grid p {d}⟩ ⟨lv ?r d ≤ lv x d⟩
and lv-lr
  show x ∈ grid ?l {d} using grid-part[where p=?l and p'=x and d=d] by
auto
qed (auto simp add: child-def)
thus ?thesis by (auto intro: grid-child)
qed
lemma grid-change-dim: assumes grid: p ∈ grid b ds
shows p[d := X] ∈ grid (b[d := X]) ds
using grid
proof induct
case (Child p d' dir)
show ?case
proof (cases d ≠ d')
  case True
  have (child p dir d')[d := X] = child (p[d := X]) dir d'
    unfolding child-def and ix-def and lv-def
    unfolding list-update-swap[OF ⟨d ≠ d'⟩] and nth-list-update-neq[OF ⟨d ≠
d'⟩] ..
  thus ?thesis using Child by auto
next
  case False hence d = d' by auto
  with Child show ?thesis unfolding child-def ⟨d = d'⟩ list-update-overwrite
by auto
qed
qed auto
```

lemma *grid-change-dim-child*: **assumes** *grid*: $p \in grid\ b\ ds$ **and** $d \notin ds$
shows *child p dir d* $\in$ *grid* (*child b dir d*) *ds*
proof (*cases d* $<$ *length b*)
case *True* **thus** *?thesis* **using** *grid-change-dim*[*OF grid*]
unfolding *child-def lv-def ix-def grid-invariant*[*OF True* ‹$d \notin ds$› *grid*] **by** *auto*
next
case *False* **hence** *length b* $\leq$ *d* **and** *length p* $\leq$ *d* **using** *grid* **by** *auto*
thus *?thesis* **unfolding** *child-def* **using** *list-update-beyond assms* **by** *auto*
qed
lemma *grid-split*: **assumes** *grid*: $p \in grid\ b\ (ds' \cup ds)$ **shows** $\exists\ x \in grid\ b\ ds.\ p \in grid\ x\ ds'$
using *grid*
proof *induct*
case (*Child p d dir*)
show *?case*
proof (*cases d* $\in$ *ds'*)
case *True* **with** *Child* **show** *?thesis* **by** *auto*
next
case *False*
hence *d* $\in$ *ds* **using** *Child* **by** *auto*
obtain *x* **where** $x \in grid\ b\ ds$ **and** $p \in grid\ x\ ds'$ **using** *Child* **by** *auto*
hence *child x dir d* $\in$ *grid b ds* **using** ‹$d \in ds$› **by** *auto*
moreover have *child p dir d* $\in$ *grid* (*child x dir d*) *ds'*
using ‹$p \in grid\ x\ ds'$› *False* **and** *grid-change-dim-child* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
qed *auto*
lemma *grid-union-eq*: $(\bigcup\ p \in grid\ b\ ds.\ grid\ p\ ds') = grid\ b\ (ds' \cup ds)$
using *grid-split* **and** *grid-transitive*[**where** *ds''*=*ds'* $\cup$ *ds* **and** *ds*=*ds'* **and** *ds'*=*ds*, *OF - - Un-upper2 Un-upper1*] **by** *auto*
lemma *grid-onedim-split*:
grid b (*ds* $\cup$ {*d*}) = *grid b ds* $\cup$ *grid* (*child b left d*) (*ds* $\cup$ {*d*}) $\cup$ *grid* (*child b right d*) (*ds* $\cup$ {*d*})
(**is** *- = ?g* $\cup$ *?l* (*ds* $\cup$ {*d*}) $\cup$ *?r* (*ds* $\cup$ {*d*}))
proof $-$
have *?g* $\cup$ *?l* (*ds* $\cup$ {*d*}) $\cup$ *?r* (*ds* $\cup$ {*d*}) = *?g* $\cup$ ($\bigcup\ p \in ?l\ \{d\}.\ grid\ p\ ds$) $\cup$ ($\bigcup\ p \in ?r\ \{d\}.\ grid\ p\ ds$)
unfolding *grid-union-eq* **..**
also have ... = ($\bigcup\ p \in (\{b\} \cup ?l\ \{d\} \cup ?r\ \{d\}).\ grid\ p\ ds$) **by** *auto*
also have ... = ($\bigcup\ p \in grid\ b\ \{d\}.\ grid\ p\ ds$) **unfolding** *grid-partition*[**where** *p*=*b*] **..**
finally show *?thesis* **unfolding** *grid-union-eq* **by** *auto*
qed
lemma *grid-child-without-parent*: **assumes** *grid*: $p \in grid\ (child\ b\ dir\ d)\ ds$ (**is** $p \in grid\ ?c\ ds$) **and** $d < length\ b$
shows $p \neq b$
proof $-$
have *level ?c* $\leq$ *level p* **using** *grid* **by** (*rule grid-level*)
hence *level b* $<$ *level p* **using** *child-level* **and** ‹$d < length\ b$› **by** *auto*

thus *?thesis* **by** *auto*
qed
lemma *grid-disjunct'*:
assumes $p \in grid\ b\ ds$ **and** $p' \in grid\ b\ ds$ **and** $x \in grid\ p\ ds'$ **and** $p \neq p'$ **and** $ds \cap ds' = \{\}$
shows $x \notin grid\ p'\ ds'$
proof (*rule ccontr*)
assume $\neg\ x \notin grid\ p'\ ds'$ **hence** $x \in grid\ p'\ ds'$ **by** *auto*
have *l*: $length\ b = length\ p$ **and** *l'*: $length\ b = length\ p'$ **using** ‹$p \in grid\ b\ ds$› **and** ‹$p' \in grid\ b\ ds$› **by** *auto*
hence $length\ p' = length\ p$ **by** *auto*
moreover have $\forall\ d < length\ p'.\ p'\ !\ d = p\ !\ d$
proof (*rule allI*, *rule impI*)
fix *d* **assume** *dl'*: $d < length\ p'$ **hence** $d < length\ b$ **using** *l'* **by** *auto*
hence *dl*: $d < length\ p$ **using** *l* **by** *auto*
show $p'\ !\ d = p\ !\ d$
proof (*cases* $d \in ds'$)
case *True* **with** ‹$ds \cap ds' = \{\}$› **have** $d \notin ds$ **by** *auto*
hence $p'\ !\ d = b\ !\ d$ **and** $p\ !\ d = b\ !\ d$
using ‹$d < length\ b$› ‹$p' \in grid\ b\ ds$› **and** ‹$p \in grid\ b\ ds$› **and** *grid-invariant* **by** *auto*
thus *?thesis* **by** *auto*
next
case *False*
show *?thesis*
using *grid-invariant*[*OF dl' False* ‹$x \in grid\ p'\ ds'$›]
and *grid-invariant*[*OF dl False* ‹$x \in grid\ p\ ds'$›] **by** *auto*
qed
qed
ultimately have $p' = p$ **by** (*metis nth-equalityI*)
thus *False* **using** ‹$p \neq p'$› **by** *auto*
qed
lemma *grid-split1*: **assumes** *grid*: $p \in grid\ b\ (ds' \cup ds)$ **and** $ds \cap ds' = \{\}$
shows $\exists!\ x \in grid\ b\ ds.\ p \in grid\ x\ ds'$
proof (*rule ex-ex1I*)
obtain *x* **where** $x \in grid\ b\ ds$ **and** $p \in grid\ x\ ds'$ **using** *grid-split*[*OF grid*] **by** *auto*
thus $\exists\ x.\ x \in grid\ b\ ds \wedge p \in grid\ x\ ds'$ **by** *auto*
next
fix *x y*
assume $x \in grid\ b\ ds \wedge p \in grid\ x\ ds'$ **and** $y \in grid\ b\ ds \wedge p \in grid\ y\ ds'$
hence $x \in grid\ b\ ds$ **and** $p \in grid\ x\ ds'$ **and** $y \in grid\ b\ ds$ **and** $p \in grid\ y\ ds'$ **by** *auto*
show $x = y$
proof (*rule ccontr*)
assume $x \neq y$
from *grid-disjunct'*[*OF* ‹$x \in grid\ b\ ds$› ‹$y \in grid\ b\ ds$› ‹$p \in grid\ x\ ds'$› *this* ‹$ds \cap ds' = \{\}$›]
show *False* **using** ‹$p \in grid\ y\ ds'$› **by** *auto*

qed
qed

2.3 Grid Restricted to a Level

definition *lgrid* :: *grid-point* ⇒ *nat set* ⇒ *nat* ⇒ *grid-point set*
where *lgrid b ds lm* = { *p* ∈ *grid b ds*. *level p* < *lm* }

lemma *lgridI*[*intro*]:
⟦ *p* ∈ *grid b ds* ; *level p* < *lm* **⟧** ⟹ *p* ∈ *lgrid b ds lm*
unfolding *lgrid-def* **by** *simp*

lemma *lgridE*[*elim*]:
assumes *p* ∈ *lgrid b ds lm*
assumes **⟦** *p* ∈ *grid b ds* ; *level p* < *lm* **⟧** ⟹ *P*
shows *P*
using *assms* **unfolding** *lgrid-def* **by** *auto*

lemma *lgridI-child*[*intro*]:
d ∈ *ds* ⟹ *p* ∈ *lgrid* (*child b dir d*) *ds lm* ⟹ *p* ∈ *lgrid b ds lm*
by (*auto intro*: *grid-child*)

lemma *lgrid-empty*[*simp*]: *lgrid p ds* (*level p*) = {}
proof (*rule equals0I*)
fix *p'* **assume** *p'* ∈ *lgrid p ds* (*level p*)
hence *level p'* < *level p* **and** *level p* ≤ *level p'* **by** *auto*
thus *False* **by** *auto*
qed

lemma *lgrid-empty'*: **assumes** *lm* ≤ *level p* **shows** *lgrid p ds lm* = {}
proof (*rule equals0I*)
fix *p'* **assume** *p'* ∈ *lgrid p ds lm*
hence *level p'* < *lm* **and** *level p* ≤ *level p'* **by** *auto*
thus *False* **using** ‹*lm* ≤ *level p*› **by** *auto*
qed

lemma *grid-not-child*:
assumes [*simp*]: *d* < *length p*
shows *p* ∉ *grid* (*child p dir d*) *ds*
proof (*rule ccontr*)
assume ¬ *?thesis*
have *level p* < *level* (*child p dir d*) **by** *auto*
with *grid-level*[*OF* ‹¬ *?thesis*›[*unfolded not-not*]]
show *False* **by** *auto*
qed

2.4 Unbounded Sparse Grid

definition *sparsegrid'* :: *nat* ⇒ *grid-point set*
where

sparsegrid' dm = grid (start dm) { 0 ..< dm }

lemma *grid-subset-alldim*:
assumes *p*: *p ∈ grid b ds*
defines *dm ≡ length b*
shows *p ∈ grid b {0..<dm}*
proof −
have *ds ∩ {dm..} ∪ ds ∩ {0..<dm} = ds* **by** *auto*
from *gridgen-dim-restrict*[**where** *ds=ds ∩ {0..<dm}* **and** *ds'=ds ∩ {dm..}*] *this*
have *ds ∩ {0..<dm} ⊆ {0..<dm}*
and *p ∈ grid b (ds ∩ {0..<dm})* **using** *p* **unfolding** *dm-def* **by** *auto*
thus *?thesis* **by** (*rule grid-union-dims*)
qed

lemma *sparsegrid'-length*[*simp*]:
b ∈ sparsegrid' dm ⟹ length b = dm **unfolding** *sparsegrid'-def* **by** *auto*

lemma *sparsegrid'I*[*intro*]:
assumes *b*: *b ∈ sparsegrid' dm* **and** *p*: *p ∈ grid b ds*
shows *p ∈ sparsegrid' dm*
using *sparsegrid'-length*[*OF b*] *b*
grid-transitive[*OF grid-subset-alldim*[*OF p*], **where** *c=start dm* **and** *ds''={0..<dm}*]
unfolding *sparsegrid'-def* **by** *auto*

lemma *sparsegrid'-start*:
assumes *b ∈ grid (start dm) ds*
shows *b ∈ sparsegrid' dm*
unfolding *sparsegrid'-def*
using *grid-subset-alldim*[*OF assms*] **by** *simp*

2.5 Sparse Grid

definition *sparsegrid* :: *nat ⇒ nat ⇒ grid-point set*
where
sparsegrid dm lm = lgrid (start dm) { 0 ..< dm } lm

lemma *sparsegrid-length*: *p ∈ sparsegrid dm lm ⟹ length p = dm*
by (*auto simp*: *sparsegrid-def*)

lemma *sparsegrid-subset*[*intro*]: *p ∈ sparsegrid dm lm ⟹ p ∈ sparsegrid' dm*
unfolding *sparsegrid-def sparsegrid'-def lgrid-def* **by** *auto*

lemma *sparsegridI*[*intro*]:
assumes *p ∈ sparsegrid' dm* **and** *level p < lm*
shows *p ∈ sparsegrid dm lm*
using *assms* **unfolding** *sparsegrid'-def sparsegrid-def lgrid-def* **by** *auto*

lemma *sparsegrid-start*:
assumes *b ∈ lgrid (start dm) ds lm*

shows $b \in sparsegrid\ dm\ lm$
proof
have $b \in grid\ (start\ dm)\ ds$ **using** *assms* **by** *auto*
thus $b \in sparsegrid'\ dm$ **by** (*rule sparsegrid'-start*)
qed (*insert assms*, *auto*)

lemma *sparsegridE*[*elim*]:
assumes $p \in sparsegrid\ dm\ lm$
shows $p \in sparsegrid'\ dm$ **and** $level\ p < lm$
using *assms* **unfolding** *sparsegrid'-def sparsegrid-def lgrid-def* **by** *auto*

2.6 Compute Sparse Grid Points

fun *gridgen* :: *grid-point* ⇒ *nat set* ⇒ *nat* ⇒ *grid-point list*
where
gridgen p ds 0 = []
| *gridgen p ds* (*Suc l*) = (*let*
sub = λ *d*. *gridgen* (*child p left d*) { $d' \in ds\ .\ d' \leq d$ } *l* @
gridgen (*child p right d*) { $d' \in ds\ .\ d' \leq d$ } *l*
in p # *concat* (*map sub* [$d \leftarrow [0\ ..<\ length\ p].\ d \in ds$]))

lemma *gridgen-lgrid-eq*: *set* (*gridgen p ds l*) = *lgrid p ds* (*level p* + *l*)
proof (*induct l arbitrary*: *p ds*)
case (*Suc l*)
let *?subg dir d* = *set* (*gridgen* (*child p dir d*) { $d' \in ds\ .\ d' \leq\ d$ } *l*)
let *?sub dir d* = *lgrid* (*child p dir d*) { $d' \in ds\ .\ d' \leq\ d$ } (*level p* + *Suc l*)
let *?union F dm* = {*p*} ∪ (⋃ *d* ∈ { $d \in ds.\ d < dm$ }. *F left d* ∪ *F right d*)

have *hyp*: !! *dir d*. $d < length\ p \Longrightarrow$ *?subg dir d* = *?sub dir d*
using *Suc.hyps* **using** *child-level* **by** *auto*

{ fix *dm* **assume** $dm \leq length\ p$
hence *?union ?sub dm* = *lgrid p* {$d \in ds.\ d < dm$} (*level p* + *Suc l*)
proof (*induct dm*)
case (*Suc dm*)
hence $dm \leq length\ p$ **by** *auto*

let *?l* = *child p left dm* **and** *?r* = *child p right dm*

have *p-lgrid*: $p \in lgrid\ p$ {$d \in ds.\ d < dm$} (*level p* + *Suc l*) **by** *auto*

show *?case*
proof (*cases* $dm \in ds$)
case *True*
let *?ds* = {$d \in ds.\ d < dm$} ∪ {*dm*}
have *ds-eq*: {$d' \in ds.\ d' \leq dm$} = *?ds* **using** *True* **by** *auto*
have *ds-eq'*: {$d \in ds.\ d < Suc\ dm$} = {$d \in ds.\ d < dm$ } ∪ {*dm*} **using** *True* **by** *auto*

```
      have ?union ?sub (Suc dm) = ?union ?sub dm ∪ ({p} ∪ ?sub left dm ∪
?sub right dm)
        unfolding ds-eq' by auto
      also have ... = lgrid p {d ∈ ds. d < dm} (level p + Suc l) ∪ ?sub left dm
∪ ?sub right dm
        unfolding Suc.hyps[OF ‹dm ≤ length p›] using p-lgrid by auto
      also have ... = {p' ∈ grid p {d ∈ ds. d<dm} ∪ (grid ?l ?ds) ∪ (grid ?r
?ds).
        level p' < level p + Suc l} unfolding lgrid-def ds-eq by auto
      also have ... = lgrid p {d ∈ ds. d < Suc dm} (level p + Suc l)
        unfolding lgrid-def ds-eq' unfolding grid-onedim-split[where b=p] ..
      finally show ?thesis .
    next
      case False hence {d ∈ ds. d < Suc dm} = {d ∈ ds. d < dm ∨ d = dm}
by auto
      hence ds-eq: {d ∈ ds. d < Suc dm} = {d ∈ ds. d < dm} using ‹dm ∉ ds›
by auto
      show ?thesis unfolding ds-eq Suc.hyps[OF ‹dm ≤ length p›] ..
    qed
  next case 0 thus ?case unfolding lgrid-def by auto
  qed }
 hence ?union ?sub (length p) = lgrid p {d ∈ ds. d < length p} (level p + Suc
l) by auto
 hence union-lgrid-eq: ?union ?sub (length p) = lgrid p ds (level p + Suc l)
  unfolding lgrid-def using grid-dim-remove-outer by auto

 have set (gridgen p ds (Suc l)) = ?union ?subg (length p)
  unfolding gridgen.simps and Let-def by auto
 hence set (gridgen p ds (Suc l)) = ?union ?sub (length p)
  using hyp by auto
 also have ... = lgrid p ds (level p + Suc l)
  using union-lgrid-eq .
 finally show ?case .
qed auto

lemma gridgen-distinct: distinct (gridgen p ds l)
proof (induct l arbitrary: p ds)
 case (Suc l)
 let ?ds = [d ← [0..<length p]. d ∈ ds]
 let ?left d = gridgen (child p left d) { d' ∈ ds . d' ≤ d } l
 and ?right d = gridgen (child p right d) { d' ∈ ds . d' ≤ d } l
 let ?sub d = ?left d @ ?right d

 have distinct (concat (map ?sub ?ds))
 proof (cases l)
  case (Suc l')

  have inj-on: inj-on ?sub (set ?ds)
  proof (rule inj-onI, rule ccontr)
```

```
  fix d d' assume d ∈ set ?ds and d' ∈ set ?ds
  hence d < length p and d ∈ set ?ds and d' < length p by auto
  assume *: ?sub d = ?sub d'
  have in-d: child p left d ∈ set (?sub d)
    using ‹d ∈ set ?ds› Suc
    by (auto simp add: gridgen-lgrid-eq lgrid-def grid-Start)

  have in-d': child p left d' ∈ set (?sub d')
    using ‹d ∈ set ?ds› Suc
    by (auto simp add: gridgen-lgrid-eq lgrid-def grid-Start)

  { fix p' d assume d ∈ set ?ds and p' ∈ set (?sub d)
    hence lv p d < lv p' d
      using grid-child-level
      by (auto simp add: gridgen-lgrid-eq lgrid-def grid-child-level) }
  note level-less = this

  assume d ≠ d'
  show False
  proof (cases d' < d)
    case True
    with in-d' ‹?sub d = ?sub d'› level-less[OF ‹d ∈ set ?ds›]
    have lv p d < lv (child p left d') d by simp
    thus False unfolding lv-def
      using child-invariant[OF ‹d < length p›, of left d'] ‹d ≠ d'›
      by auto
  next
    case False hence d < d' using ‹d ≠ d'› by auto
    with in-d ‹?sub d = ?sub d'› level-less[OF ‹d' ∈ set ?ds›]
    have lv p d' < lv (child p left d) d' by simp
    thus False unfolding lv-def
      using child-invariant[OF ‹d' < length p›, of left d] ‹d ≠ d'›
      by auto
  qed
qed

show ?thesis
proof (rule distinct-concat)
  show distinct (map ?sub ?ds)
    unfolding distinct-map using inj-on by simp
next
  fix ys assume ys ∈ set (map ?sub ?ds)
  then obtain d where d ∈ ds and d < length p
    and *: ys = ?sub d by auto

  show distinct ys unfolding *
    using grid-disjunct[OF ‹d < length p›, of {d' ∈ ds. d' ≤ d}]
      gridgen-lgrid-eq lgrid-def ‹distinct (?left d)› ‹distinct (?right d)›
    by auto
```

```
  next
    fix ys zs
    assume ys ∈ set (map ?sub ?ds)
    then obtain d where ys: ys = ?sub d and d ∈ set ?ds by auto
    hence d < length p by auto

    assume zs ∈ set (map ?sub ?ds)
    then obtain d' where zs: zs = ?sub d' and d' ∈ set ?ds by auto
    hence d' < length p by auto

    assume ys ≠ zs
    hence d' ≠ d unfolding ys zs by auto

    show set ys ∩ set zs = {}
    proof (rule ccontr)
      assume ¬ ?thesis
      then obtain p' where p' ∈ set (?sub d) and p' ∈ set (?sub d')
        unfolding ys zs by auto

      hence lv p d < lv p' d lv p d' < lv p' d'
        using grid-child-level ‹d ∈ set ?ds› ‹d' ∈ set ?ds›
        by (auto simp add: gridgen-lgrid-eq lgrid-def grid-child-level)

      show False
      proof (cases d < d')
        case True
        from ‹p' ∈ set (?sub d)›
        have p ! d' = p' ! d'
          using grid-invariant[of d' child p right d {d' ∈ ds. d' ≤ d}]
          using grid-invariant[of d' child p left d {d' ∈ ds. d' ≤ d}]
          using child-invariant[of d' - - d] ‹d < d'› ‹d' < length p›
          using gridgen-lgrid-eq lgrid-def by auto
        thus False using ‹lv p d' < lv p' d'› unfolding lv-def by auto
      next
        case False hence d' < d using ‹d' ≠ d› by simp
        from ‹p' ∈ set (?sub d')›
        have p ! d = p' ! d
          using grid-invariant[of d child p right d' {d ∈ ds. d ≤ d'}]
          using grid-invariant[of d child p left d' {d ∈ ds. d ≤ d'}]
          using child-invariant[of d - - d'] ‹d' < d› ‹d < length p›
          using gridgen-lgrid-eq lgrid-def by auto
        thus False using ‹lv p d < lv p' d› unfolding lv-def by auto
      qed
    qed
  qed
qed (simp add: map-replicate-const)
moreover
have p ∉ set (concat (map ?sub ?ds))
  using gridgen-lgrid-eq lgrid-def grid-not-child[of - p] by simp
```

ultimately show *?case*
unfolding *gridgen.simps Let-def distinct.simps* **by** *simp*
qed *auto*

lemma *lgrid-finite*: *finite* (*lgrid b ds lm*)
proof (*cases level b* $\leq$ *lm*)
case *True* **from** *iffD1*[*OF le-iff-add True*]
obtain *l* **where** *l*: *lm* = *level b* + *l* **by** *auto*
show *?thesis* **unfolding** *l gridgen-lgrid-eq*[*symmetric*] **by** *auto*
next
case *False* **hence** !! *x*. *x* $\in$ *grid b ds* $\Longrightarrow$ ($\neg$ *level x* < *lm*)
proof −
fix *x* **assume** *x* $\in$ *grid b ds*
from *grid-level*[*OF this*] **show** $\neg$ *level x* < *lm* **using** *False* **by** *auto*
qed
hence *lgrid b ds lm* = {} **unfolding** *lgrid-def* **by** *auto*
thus *?thesis* **by** *auto*
qed

lemma *lgrid-sum*:
fixes *F* :: *grid-point* $\Rightarrow$ *real*
assumes *d* < *length b* **and** *level b* < *lm*
shows ($\sum$ *p* $\in$ *lgrid b* {*d*} *lm*. *F p*) =
($\sum$ *p* $\in$ *lgrid* (*child b left d*) {*d*} *lm*. *F p*) + ($\sum$ *p* $\in$ *lgrid* (*child b right d*) {*d*} *lm*. *F p*) + *F b*
(**is** ($\sum$ *p* $\in$ *?grid b*. *F p*) = ($\sum$ *p* $\in$ *?grid ?l* . *F p*) + (*?sum* (*?grid ?r*)) + *F b*)
proof −
have !! *dir*. *b* $\notin$ *?grid* (*child b dir d*)
using *grid-child-without-parent*[**where** *ds*={*d*}] **and** ‹*d* < *length b*› **and** *lgrid-def* **by** *auto*
hence *b-distinct*: *b* $\notin$ (*?grid ?l* $\cup$ *?grid ?r*) **by** *auto*

have *?grid ?l* $\cap$ *?grid ?r* = {}
unfolding *lgrid-def* **using** *grid-disjunct* **and** ‹*d* < *length b*› **by** *auto*
from *lgrid-finite lgrid-finite* **and** *this*
have *child-eq*: *?sum* ((*?grid ?l*) $\cup$ (*?grid ?r*)) = *?sum* (*?grid ?l*) + *?sum* (*?grid ?r*)
by (*rule sum.union-disjoint*)

have *?grid b* = {*b*} $\cup$ (*?grid ?l*) $\cup$ (*?grid ?r*) **unfolding** *lgrid-def grid-partition*[**where** *p*=*b*] **using** *assms* **by** *auto*
hence *?sum* (*?grid b*) = *F b* + *?sum* ((*?grid ?l*) $\cup$ (*?grid ?r*)) **using** *b-distinct* **and** *lgrid-finite* **by** *auto*
thus *?thesis* **using** *child-eq* **by** *auto*
qed

2.7 Base Points

definition *base* :: *nat set* $\Rightarrow$ *grid-point* $\Rightarrow$ *grid-point*

where *base ds p* = (*THE b*. *b* ∈ *grid* (*start* (*length p*)) ({*0* ..< *length p*} − *ds*) ∧ *p* ∈ *grid b ds*)

lemma *baseE*: **assumes** *p-grid*: *p* ∈ *sparsegrid′ dm*
shows *base ds p* ∈ *grid* (*start dm*) ({*0*..<*dm*} − *ds*)
and *p* ∈ *grid* (*base ds p*) *ds*
proof −
from *p-grid*[*unfolded sparsegrid′-def*]
have ∗: ∃! *x* ∈ *grid* (*start dm*) ({*0*..<*dm*} − *ds*). *p* ∈ *grid x ds*
by (*intro grid-split1*) (*auto intro*: *grid-union-dims*)
then obtain *x* **where** *x-eq*: *x* ∈ *grid* (*start dm*) ({*0*..<*dm*} − *ds*) ∧ *p* ∈ *grid x ds*
by *auto*
with ∗ **have** *base ds p* = *x* **unfolding** *base-def* **by** *auto*
thus *base ds p* ∈ *grid* (*start dm*) ({*0*..<*dm*} − *ds*) **and** *p* ∈ *grid* (*base ds p*) *ds*
using *x-eq* **by** *auto*
qed

lemma *baseI*: **assumes** *x-grid*: *x* ∈ *grid* (*start dm*) ({*0*..<*dm*} − *ds*) **and** *p-xgrid*: *p* ∈ *grid x ds*
shows *base ds p* = *x*
proof −
have *p* ∈ *grid* (*start dm*) (*ds* ∪ ({*0*..<*dm*} − *ds*))
using *grid-transitive*[*OF p-xgrid x-grid*, **where** *ds′′*=*ds* ∪ ({*0*..<*dm*} − *ds*)]
by *auto*
moreover have *ds* ∩ ({*0*..<*dm*} − *ds*) = {} **by** *auto*
ultimately have ∃! *x* ∈ *grid* (*start dm*) ({*0*..<*dm*} − *ds*). *p* ∈ *grid x ds*
using *grid-split1*[**where** *p*=*p* **and** *b*=*start dm* **and** *ds′*=*ds* **and** *ds*={*0*..<*dm*} − *ds*] **by** *auto*
thus *base ds p* = *x* **using** *x-grid p-xgrid* **unfolding** *base-def* **by** *auto*
qed

lemma *base-empty*: **assumes** *p-grid*: *p* ∈ *sparsegrid′ dm* **shows** *base* {} *p* = *p*
using *grid-empty-ds* **and** *p-grid* **and** *grid-split1*[**where** *ds*={*0*..<*dm*} **and** *ds′*={}]
unfolding *base-def sparsegrid′-def* **by** *auto*

lemma *base-start-eq*: **assumes** *p-spg*: *p* ∈ *sparsegrid dm lm*
shows *start dm* = *base* {*0*..<*dm*} *p*
proof −
from *p-spg*
have *start dm* ∈ *grid* (*start dm*) ({*0*..<*dm*} − {*0*..<*dm*})
and *p* ∈ *grid* (*start dm*) {*0*..<*dm*} **using** *sparsegrid′-def* **by** *auto*
from *baseI*[*OF this*(*1*) *this*(*2*)] **show** *?thesis* **by** *auto*
qed

lemma *base-in-grid*: **assumes** *p-grid*: *p* ∈ *sparsegrid′ dm* **shows** *base ds p* ∈ *grid* (*start dm*) {*0*..<*dm*}
proof −
let *?ds* = *ds* ∪ {*0*..<*dm*}

have *ds-eq*: { $d \in ?ds.\ d < length\ (start\ dm)$ } = { $0..< dm$}
unfolding *start-def* **by** *auto*
have *base ds p* ∈ *grid* (*start dm*) *?ds*
using *grid-union-dims*[*OF - baseE*(*1*)[*OF p-grid*, **where** *ds*=*ds*], **where** *ds'*=*?ds*]
by *auto*
thus *?thesis* **using** *grid-dim-remove-outer*[**where** *b*=*start dm* **and** *ds*=*?ds*] **unfolding** *ds-eq* **by** *auto*
qed

lemma *base-grid*: **assumes** *p-grid*: $p \in sparsegrid'\ dm$ **shows** *grid* (*base ds p*) *ds* ⊆ *sparsegrid' dm*
proof
fix *x* **assume** *xgrid*: $x \in grid\ (base\ ds\ p)\ ds$
have *ds-eq*: { $d \in \{0..<dm\} \cup ds.\ d < length\ (start\ dm)$ } = {*0..<dm*} **by** *auto*
from *grid-transitive*[*OF xgrid base-in-grid*[*OF p-grid*], **where** *ds''*={*0..<dm*} ∪ *ds*]
show *x* ∈ *sparsegrid' dm* **unfolding** *sparsegrid'-def*
using *grid-dim-remove-outer*[**where** *b*=*start dm* **and** *ds*={*0..<dm*} ∪ *ds*] **unfolding** *ds-eq* **unfolding** *Un-ac*(*3*)[*of* {*0..<dm*}]
by *auto*
qed
lemma *base-length*[*simp*]: **assumes** *p-grid*: $p \in sparsegrid'\ dm$ **shows** *length* (*base ds p*) = *dm*
proof −
from *baseE*[*OF p-grid*] **have** *base ds p* ∈ *grid* (*start dm*) ({*0..<dm*} − *ds*) **by** *auto*
thus *?thesis* **by** *auto*
qed
lemma *base-in*[*simp*]: **assumes** *d* < *dm* **and** *d* ∈ *ds* **and** *p-grid*: $p \in sparsegrid'$ *dm* **shows** *base ds p* ! *d* = *start dm* ! *d*
proof −
have *ds*: *d* ∉ {*0..<dm*} − *ds* **using** ‹*d* ∈ *ds*› **by** *auto*
have *d* < *length* (*start dm*) **using** ‹*d* < *dm*› **by** *auto*
with *grid-invariant*[*OF this ds*] *baseE*(*1*)[*OF p-grid*] **show** *?thesis* **by** *auto*
qed
lemma *base-out*[*simp*]: **assumes** *d* < *dm* **and** *d* ∉ *ds* **and** *p-grid*: $p \in sparsegrid'$ *dm* **shows** *base ds p* ! *d* = *p* ! *d*
proof −
have *d* < *length* (*base ds p*) **using** *base-length*[*OF p-grid*] ‹*d* < *dm*› **by** *auto*
with *grid-invariant*[*OF this* ‹*d* ∉ *ds*›] *baseE*(*2*)[*OF p-grid*] **show** *?thesis* **by** *auto*
qed
lemma *base-base*: **assumes** *p-grid*: $p \in sparsegrid'\ dm$ **shows** *base ds* (*base ds'* *p*) = *base* (*ds* ∪ *ds'*) *p*
proof (*rule nth-equalityI*)
have *b-spg*: *base ds' p* ∈ *sparsegrid' dm* **unfolding** *sparsegrid'-def*
using *grid-union-dims*[*OF Diff-subset*[**where** *A*={*0..<dm*} **and** *B*=*ds'*] *baseE*(*1*)[*OF p-grid*]] .
from *base-length*[*OF b-spg*] *base-length*[*OF p-grid*] **show** *length* (*base ds* (*base ds' p*)) = *length* (*base* (*ds* ∪ *ds'*) *p*) **by** *auto*

show *base ds* (*base ds′ p*) ! *i* = *base* (*ds* ∪ *ds′*) *p* ! *i* **if** *i* < *length* (*base ds* (*base ds′ p*)) **for** *i*
proof −
have *i* < *dm* **using** *that base-length*[*OF b-spg*] **by** *auto*
show *base ds* (*base ds′ p*) ! *i* = *base* (*ds* ∪ *ds′*) *p* ! *i*
proof (*cases i* ∈ *ds* ∪ *ds′*)
case *True*
show *?thesis*
proof (*cases i* ∈ *ds*)
case *True* **from** *base-in*[*OF* ‹*i* < *dm*› ‹*i* ∈ *ds* ∪ *ds′*› *p-grid*] *base-in*[*OF* ‹*i* < *dm*› *this b-spg*] **show** *?thesis* **by** *auto*
next
case *False* **hence** *i* ∈ *ds′* **using** ‹*i* ∈ *ds* ∪ *ds′*› **by** *auto*
from *base-in*[*OF* ‹*i* < *dm*› ‹*i* ∈ *ds* ∪ *ds′*› *p-grid*] *base-out*[*OF* ‹*i* < *dm*› ‹*i* ∉ *ds*› *b-spg*] *base-in*[*OF* ‹*i* < *dm*› ‹*i* ∈ *ds′*› *p-grid*] **show** *?thesis* **by** *auto*
qed
next
case *False* **hence** *i* ∉ *ds* **and** *i* ∉ *ds′* **by** *auto*
from *base-out*[*OF* ‹*i* < *dm*› ‹*i* ∉ *ds* ∪ *ds′*› *p-grid*] *base-out*[*OF* ‹*i* < *dm*› ‹*i* ∉ *ds*› *b-spg*] *base-out*[*OF* ‹*i* < *dm*› ‹*i* ∉ *ds′*› *p-grid*] **show** *?thesis* **by** *auto*
qed
qed
qed
lemma *grid-base-out*: **assumes** *d* < *dm* **and** *d* ∉ *ds* **and** *p-grid*: *b* ∈ *sparsegrid′ dm* **and** *p* ∈ *grid* (*base ds b*) *ds*
shows *p* ! *d* = *b* ! *d*
proof −
have *base ds b* ! *d* = *b* ! *d* **using** *assms* **by** *auto*
moreover have *d* < *length* (*base ds b*) **using** *assms* **by** *auto*
from *grid-invariant*[*OF this*]
have *p* ! *d* = *base ds b* ! *d* **using** *assms* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *grid-grid-inj-on*: **assumes** *ds* ∩ *ds′* = {} **shows** *inj-on snd* (⋃*p′*∈*grid b ds*. ⋃*p′′*∈*grid p′ ds′*. {(*p′*, *p′′*)})
proof (*rule inj-onI*)
fix *x y*
assume *x* ∈ (⋃*p′*∈*grid b ds*. ⋃*p′′*∈*grid p′ ds′*. {(*p′*, *p′′*)})
hence *snd x* ∈ *grid* (*fst x*) *ds′* **and** *fst x* ∈ *grid b ds* **by** *auto*

assume *y* ∈ (⋃*p′*∈*grid b ds*. ⋃*p′′*∈*grid p′ ds′*. {(*p′*, *p′′*)})
hence *snd y* ∈ *grid* (*fst y*) *ds′* **and** *fst y* ∈ *grid b ds* **by** *auto*

assume *snd x* = *snd y*
have *fst x* = *fst y*
proof (*rule ccontr*)
assume *fst x* ≠ *fst y*

from *grid-disjunct'*[*OF* ‹*fst x* ∈ *grid b ds*› ‹*fst y* ∈ *grid b ds*› ‹*snd x* ∈ *grid* (*fst x*) *ds'*› *this* ‹*ds* ∩ *ds'* = {}›]
show *False* **using** ‹*snd y* ∈ *grid* (*fst y*) *ds'*› **unfolding** ‹*snd x* = *snd y*› **by** *auto*
qed
show *x* = *y* **using** *prod-eqI*[*OF* ‹*fst x* = *fst y*› ‹*snd x* = *snd y*›] **.**
qed

lemma *grid-level-d*: **assumes** *d* < *length b* **and** *p-grid*: *p* ∈ *grid b* {*d*} **and** *p* ≠ *b* **shows** *lv p d* > *lv b d*
proof −
from *p-grid*[*unfolded grid-partition*[**where** *p*=*b*]]
show *?thesis* **using** *grid-child-level* **using** *assms* **by** *auto*
qed

lemma *grid-base-base*: **assumes** *b* ∈ *sparsegrid' dm*
shows *base ds' b* ∈ *grid* (*base ds* (*base ds' b*)) (*ds* ∪ *ds'*)
proof −
from *base-grid*[*OF* ‹*b* ∈ *sparsegrid' dm*›] **have** *base ds' b* ∈ *sparsegrid' dm* **by** *auto*
from *grid-union-dims*[*OF* - *baseE*(*2*)[*OF this*], *of ds ds* ∪ *ds'*] **show** *?thesis* **by** *auto*
qed

lemma *grid-base-union*: **assumes** *b-spg*: *b* ∈ *sparsegrid' dm* **and** *p-grid*: *p* ∈ *grid* (*base ds b*) *ds* **and** *x-grid*: *x* ∈ *grid* (*base ds' p*) *ds'*
shows *x* ∈ *grid* (*base* (*ds* ∪ *ds'*) *b*) (*ds* ∪ *ds'*)
proof −
have *ds-union*: *ds* ∪ *ds'* = *ds'* ∪ (*ds* ∪ *ds'*) **by** *auto*

from *base-grid*[*OF b-spg*] *p-grid* **have** *p-spg*: *p* ∈ *sparsegrid' dm* **by** *auto*
with *assms* **and** *grid-base-base* **have** *base-b'*: *base ds' p* ∈ *grid* (*base ds* (*base ds' p*)) (*ds* ∪ *ds'*) **by** *auto*
moreover have *base ds'* (*base ds b*) = *base ds'* (*base ds p*) (**is** *?b* = *?p*)
proof (*rule nth-equalityI*)
have *bb-spg*: *base ds b* ∈ *sparsegrid' dm* **using** *base-grid*[*OF b-spg*] *grid.Start* **by** *auto*
hence *dm* = *length* (*base ds b*) **by** *auto*
have *bp-spg*: *base ds p* ∈ *sparsegrid' dm* **using** *base-grid*[*OF p-spg*] *grid.Start* **by** *auto*

show *length ?b* = *length ?p* **using** *base-length*[*OF bp-spg*] *base-length*[*OF bb-spg*] **by** *auto*
show *?b* ! *i* = *?p* ! *i* **if** *i* < *length ?b* **for** *i*
proof −
have *i* < *dm* **and** *i* < *length* (*base ds b*) **using** *that base-length*[*OF bb-spg*] ‹*dm* = *length* (*base ds b*)› **by** *auto*
show *?b* ! *i* = *?p* ! *i*
proof (*cases i* ∈ *ds* ∪ *ds'*)

```
      case True
     hence !! x. base ds x ∈ sparsegrid' dm ⟹ x ∈ sparsegrid' dm ⟹ base ds'
(base ds x) ! i = (start dm) ! i
      proof − fix x assume x-spg: x ∈ sparsegrid' dm and xb-spg: base ds x ∈
sparsegrid' dm
        show base ds' (base ds x) ! i = (start dm) ! i
        proof (cases i ∈ ds')
          case True from base-in[OF ‹i < dm› this xb-spg] show ?thesis .
        next
          case False hence i ∈ ds using ‹i ∈ ds ∪ ds'› by auto
         from base-out[OF ‹i < dm› False xb-spg] base-in[OF ‹i < dm› this x-spg]
show ?thesis by auto
        qed
      qed
      from this[OF bp-spg p-spg] this[OF bb-spg b-spg] show ?thesis by auto
    next
      case False hence i ∉ ds and i ∉ ds' by auto
      from grid-invariant[OF ‹i < length (base ds b)› ‹i ∉ ds› p-grid]
        base-out[OF ‹i < dm› ‹i ∉ ds'› bp-spg] base-out[OF ‹i < dm› ‹i ∉ ds›
p-spg] base-out[OF ‹i < dm› ‹i ∉ ds'› bb-spg]
      show ?thesis by auto
    qed
  qed
 qed
 ultimately have base ds' p ∈ grid (base (ds ∪ ds') b) (ds ∪ ds')
  by (simp only: base-base[OF p-spg] base-base[OF b-spg] Un-ac(3))
 from grid-transitive[OF x-grid this] show ?thesis using ds-union by auto
qed
lemma grid-base-dim-add: assumes ds' ⊆ ds and b-spg: b ∈ sparsegrid' dm and
p-grid: p ∈ grid (base ds' b) ds'
 shows p ∈ grid (base ds b) ds
proof −
 have ds-eq: ds' ∪ ds = ds using assms by auto

 have p ∈ sparsegrid' dm using base-grid[OF b-spg] p-grid by auto
 hence p ∈ grid (base ds p) ds using baseE by auto
 from grid-base-union[OF b-spg p-grid this]
 show ?thesis using ds-eq by auto
qed
lemma grid-replace-dim: assumes d < length b' and d < length b and p-grid: p
∈ grid b ds and p'-grid: p' ∈ grid b' ds
 shows p[d := p' ! d] ∈ grid (b[d := b' ! d]) ds (is - ∈ grid ?b ds)
 using p'-grid and p-grid
proof induct
 case (Child p'' d' dir)
 hence p''-grid: p[d := p'' ! d] ∈ grid ?b ds and d < length p'' using assms by
auto
 have d < length p using p-grid assms by auto
 thus ?case
```

proof (*cases* $d' = d$)
case *True*
from *grid.Child*[*OF* p''*-grid* ‹$d' \in ds$›]
show *?thesis* **unfolding** *child-def ix-def lv-def list-update-overwrite* ‹$d' = d$› *nth-list-update-eq*[*OF* ‹$d <$ *length* p''›] *nth-list-update-eq*[*OF* ‹$d <$ *length* p›] **.**
next
case *False*
show *?thesis* **unfolding** *child-def nth-list-update-neq*[*OF False*] **using** *Child* **by** *auto*
qed
qed (*rule grid-change-dim*)
lemma *grid-shift-base*:
assumes *ds-dj*: $ds \cap ds' = \{\}$ **and** *b-spg*: $b \in$ *sparsegrid'* dm **and** *p-grid*: $p \in$ *grid* (*base* ($ds' \cup ds$) b) ($ds' \cup ds$)
shows *base* ds' $p \in$ *grid* (*base* ($ds \cup ds'$) b) ds
proof −
from *grid-split*[*OF p-grid*]
obtain x **where** *x-grid*: $x \in$ *grid* (*base* ($ds' \cup ds$) b) ds **and** *p-xgrid*: $p \in$ *grid* x ds' **by** *auto*
from *grid-union-dims*[*OF - this*(*1*)]
have *x-spg*: $x \in$ *sparsegrid'* dm **using** *base-grid*[*OF b-spg*] **by** *auto*

have *b-len*: *length* (*base* ($ds' \cup ds$) b) = dm **using** *base-length*[*OF b-spg*] **by** *auto*

define d' **where** $d' = dm$
moreover have $d' \leq dm \Longrightarrow x \in$ *grid* (*start* dm) ($\{0..<dm\} - \{d \in ds'.\ d < d'\}$)
proof (*induct* d')
case (*Suc* d')
with *b-len* **have** d'*-b*: $d' <$ *length* (*base* ($ds' \cup ds$) b) **by** *auto*
show *?case*
proof (*cases* $d' \in ds'$)
case *True* **hence** $d' \notin ds$ **and** $d' \in ds' \cup ds$ **using** *ds-dj* **by** *auto*
have $\{0..<dm\} - \{d \in ds'.\ d < d'\} = (\{0..<dm\} - \{d \in ds'.\ d < d'\}) - \{d'\} \cup \{d'\}$ **using** ‹*Suc* $d' \leq dm$› **by** *auto*
also have $\ldots = (\{0..<dm\} - \{d \in ds'.\ d <$ *Suc* $d'\}) \cup \{d'\}$ **by** *auto*
finally have *x-g*: $x \in$ *grid* (*start* dm) ($\{d'\} \cup (\{0..<dm\} - \{d \in ds'.\ d <$ *Suc* $d'\})$) **using** *Suc* **by** *auto*
from *grid-invariant*[*OF* d'*-b* ‹$d' \notin ds$› *x-grid*] *base-in*[*OF* - ‹$d' \in ds' \cup ds$› *b-spg*] ‹*Suc* $d' \leq dm$›
have x ! d' = *start* dm ! d' **by** *auto*
from *grid-dim-remove*[*OF x-g this*] **show** *?thesis* **.**
next
case *False*
hence $\{d \in ds'.\ d <$ *Suc* $d'\} = \{d \in ds'.\ d < d' \vee d = d'\}$ **by** *auto*
also have $\ldots = \{d \in ds'.\ d < d'\}$ **using** *False* **by** *auto*
finally show *?thesis* **using** *Suc* **by** *auto*
qed
next

case *0* **show** *?case* **using** *x-spg*[*unfolded sparsegrid'-def*] **by** *auto*
qed
moreover have $\{0..<dm\} - ds' = \{0..<dm\} - \{d \in ds'.\ d < dm\}$ **by** *auto*
ultimately have $x \in grid\ (start\ dm)\ (\{0..<dm\} - ds')$ **by** *auto*
from *baseI*[*OF this p-xgrid*] **and** *x-grid*
show *?thesis* **by** (*auto simp*: *Un-ac*(*3*))
qed

2.8 Lift Operation over all Grid Points

definition $lift :: (nat \Rightarrow nat \Rightarrow grid\text{-}point \Rightarrow vector \Rightarrow vector) \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow vector \Rightarrow vector$
where $lift\ f\ dm\ lm\ d = foldr\ (\lambda\ p.\ f\ d\ (lm - level\ p)\ p)\ (gridgen\ (start\ dm)\ (\{\ 0\ ..<\ dm\ \} - \{\ d\ \})\ lm)$

lemma *lift*:
assumes $d < dm$ **and** $p \in sparsegrid\ dm\ lm$
and *Fintro*: $\bigwedge l\ b\ p\ \alpha.\ [\![\ b \in lgrid\ (start\ dm)\ (\{0..<dm\} - \{d\})\ lm\ ;\ l + level\ b = lm\ ;\ p \in sparsegrid\ dm\ lm\ ]\!] \Longrightarrow F\ d\ l\ b\ \alpha\ p = (if\ b = base\ \{d\}\ p\ then\ (\sum\ p' \in lgrid\ b\ \{d\}\ lm.\ S\ (\alpha\ p')\ p\ p')\ else\ \alpha\ p)$
shows $lift\ F\ dm\ lm\ d\ \alpha\ p = (\sum\ p' \in lgrid\ (base\ \{d\}\ p)\ \{d\}\ lm.\ S\ (\alpha\ p')\ p\ p')$
(**is** *?lift* = *?S p* α)
proof −
let *?gridgen* = $gridgen\ (start\ dm)\ (\{0..<dm\} - \{d\})\ lm$
let *?f p* = $F\ d\ (lm - level\ p)\ p$

{ fix *bs* β *b*
assume *set bs* $\subseteq$ *set ?gridgen* **and** *distinct bs* **and** $p \in sparsegrid\ dm\ lm$
hence *foldr ?f bs* β *p* = (*if base* $\{d\}$ *p* $\in$ *set bs then ?S p* β *else* β *p*)
proof (*induct bs arbitrary*: *p*)
case (*Cons b bs*)
hence $b \in lgrid\ (start\ dm)\ (\{0..<dm\} - \{d\})\ lm$
and $(lm - level\ b) + level\ b = lm$
and *b-grid*: $b \in grid\ (start\ dm)\ (\{0..<dm\} - \{d\})$
using *lgrid-def gridgen-lgrid-eq* **by** *auto*
note *F* = *Fintro*[*OF this*(*1*,*2*) ‹*p* $\in$ *sparsegrid dm lm*›]

have *b* $\notin$ *set bs* **using** ‹*distinct* (*b*#*bs*)› **by** *auto*

show *?case*
proof (*cases base* $\{d\}$ *p* $\in$ *set* (*b*#*bs*))
case *True* **note** *base-in-set* = *this*

show *?thesis*
proof (*cases b* = *base* $\{d\}$ *p*)
case *True*
moreover

```
        { fix p' assume p' ∈ lgrid b {d} lm
          hence p' ∈ grid b {d} and level p' < lm unfolding lgrid-def by auto
          from grid-transitive[OF this(1) b-grid, of {0..<dm}] ⟨d < dm⟩
            baseI[OF b-grid ⟨p' ∈ grid b {d}⟩] ⟨b ∉ set bs⟩
            Cons.prems Cons.hyps[of p'] this(2)
          have foldr ?f bs β p' = β p' unfolding sparsegrid-def lgrid-def by auto
}
        ultimately show ?thesis
          using F base-in-set by auto
      next
        case False
        with base-in-set have base {d} p ∈ set bs by auto
        with Cons.hyps[of p] Cons.prems
        have foldr ?f bs β p = ?S p β by auto
        thus ?thesis using F base-in-set False by auto
      qed
    next
      case False
      hence b ≠ base {d} p by auto
      from False Cons.hyps[of p] Cons.prems
      have foldr ?f bs β p = β p by auto
      thus ?thesis using False F ⟨b ≠ base {d} p⟩ by auto
    qed
  qed auto
 }
 moreover have base {d} p ∈ set ?gridgen
 proof −
  have p ∈ grid (base {d} p) {d}
    using ⟨p ∈ sparsegrid dm lm⟩[THEN sparsegrid-subset] by (rule baseE)
  from grid-level[OF this] baseE(1)[OF sparsegrid-subset[OF ⟨p ∈ sparsegrid dm
lm⟩]]
  show ?thesis using ⟨p ∈ sparsegrid dm lm⟩
    unfolding gridgen-lgrid-eq sparsegrid'-def lgrid-def sparsegrid-def
    by auto
 qed
 ultimately show ?thesis unfolding lift-def
  using gridgen-distinct ⟨p ∈ sparsegrid dm lm⟩ by auto
qed
```

2.9 Parent Points

```
definition parents :: nat ⇒ grid-point ⇒ grid-point ⇒ grid-point set
where parents d b p = { x ∈ grid b {d}. p ∈ grid x {d} }

lemma parents-split: assumes p-grid: p ∈ grid (child b dir d) {d}
 shows parents d b p = { b } ∪ parents d (child b dir d) p
proof (intro set-eqI iffI)
 let ?chd = child b dir d and ?chid = child b (inv dir) d
 fix x assume x ∈ parents d b p
```

```
  hence x ∈ grid b {d} and p ∈ grid x {d} unfolding parents-def by auto
 hence x-split: x ∈ {b} ∪ grid ?chd {d} ∪ grid ?chid {d} using grid-onedim-split[where
ds={} and b=b] and grid-empty-ds
    by (cases dir, auto)
  thus x ∈ {b} ∪ parents d (child b dir d) p
  proof (cases x = b)
    case False
    have d < length b
    proof (rule ccontr)
      assume ¬ d < length b hence empty: {d' ∈ {d}. d' < length b} = {} by
auto
      have x = b using ‹x ∈ grid b {d}›
        unfolding grid-dim-remove-outer[where ds={d} and b=b] empty
        using grid-empty-ds by auto
      thus False using ‹¬ x = b› by auto
    qed
    have x ∉ grid ?chid {d}
    proof (rule ccontr)
      assume ¬ x ∉ grid ?chid {d}
     hence p ∈ grid ?chid {d} using grid-transitive[OF ‹p ∈ grid x {d}›, where
ds'={d}]
        by auto
      hence p ∉ grid ?chd {d} using grid-disjunct[OF ‹d < length b›] by (cases
dir, auto)
      thus False using ‹p ∈ grid ?chd {d}› ..
    qed
    with False and x-split
    have x ∈ grid ?chd {d} by auto
    thus ?thesis unfolding parents-def using ‹p ∈ grid x {d}› by auto
  qed auto
next
  let ?chd = child b dir d and ?chid = child b (inv dir) d
  fix x assume x-in: x ∈ {b} ∪ parents d ?chd p
  thus x ∈ parents d b p
  proof (cases x = b)
    case False
    hence x ∈ parents d ?chd p using x-in by auto
    thus ?thesis unfolding parents-def using grid-child[where b=b] by auto
  next
    from p-grid have p ∈ grid b {d} using grid-child[where b=b] by auto
   case True thus ?thesis unfolding parents-def using ‹p ∈ grid b {d}› by auto
  qed
qed

lemma parents-no-parent: assumes d < length b shows b ∉ parents d (child b
dir d) p (is - ∉ parents - ?ch -)
proof
  assume b ∈ parents d ?ch p hence b ∈ grid ?ch {d} unfolding parents-def by
auto
```

from *grid-level*[*OF this*]
have *level b + 1 ≤ level b* **unfolding** *child-level*[*OF ‹d < length b›*] **.**
thus *False* **by** *auto*
qed

lemma *parents-subset-lgrid*: *parents d b p ⊆ lgrid b {d} (level p + 1)*
proof
fix *x* **assume** *x ∈ parents d b p*
hence *x ∈ grid b {d}* **and** *p ∈ grid x {d}* **unfolding** *parents-def* **by** *auto*
moreover hence *level x ≤ level p* **using** *grid-level* **by** *auto*
hence *level x < level p + 1* **by** *auto*
ultimately show *x ∈ lgrid b {d} (level p + 1)* **unfolding** *lgrid-def* **by** *auto*
qed

lemma *parents-finite*: *finite (parents d b p)*
using *finite-subset*[*OF parents-subset-lgrid lgrid-finite*] **.**

lemma *parent-sum*: **assumes** *p-grid*: *p ∈ grid (child b dir d) {d}* **and** *d < length b*
shows *(∑ x ∈ parents d b p. F x) = F b + (∑ x ∈ parents d (child b dir d) p. F x)*
unfolding *parents-split*[*OF p-grid*] **using** *parents-no-parent*[*OF ‹d < length b›*, **where** *dir=dir* **and** *p=p*] **using** *parents-finite*
by *auto*

lemma *parents-single*: *parents d b b = { b }*
proof
have *parents d b b ⊆ lgrid b {d} (level b + (Suc 0))* **using** *parents-subset-lgrid* **by** *auto*
also have *... = {b}* **unfolding** *gridgen-lgrid-eq*[*symmetric*] *gridgen.simps Let-def* **by** *auto*
finally show *parents d b b ⊆ { b }* **.**
next
have *b ∈ parents d b b* **unfolding** *parents-def* **by** *auto*
thus *{ b } ⊆ parents d b b* **by** *auto*
qed

lemma *grid-single-dimensional-specification*:
assumes *d < length b*
and *odd i*
and *lv b d + l' = l*
and *i < (ix b d + 1) * 2^l'*
and *i > (ix b d − 1) * 2^l'*
shows *b[d := (l,i)] ∈ grid b {d}*
using *assms* **proof** *(induct l' arbitrary: b)*
case *0*
hence *i = ix b d* **and** *l = lv b d* **by** *auto*
thus *?case* **unfolding** *ix-def lv-def* **by** *auto*
next

```
  case (Suc l')

  have d ∈ {d} by auto

  show ?case
  proof (rule linorder-cases)
    assume i = ix b d * 2^(Suc l')
    hence even i by auto
    thus ?thesis using ‹odd i› by blast
  next
    assume *: i < ix b d * 2^(Suc l')

    let ?b = child b left d

    have d < length ?b using Suc by auto
    moreover note ‹odd i›
    moreover have lv ?b d + l' = l
      and i < (ix ?b d + 1) * 2^l'
      and (ix ?b d - 1) * 2^l' < i
      unfolding child-ix-left[OF Suc.prems(1)]
      using Suc.prems * child-lv by (auto simp add: field-simps)
    ultimately have ?b[d := (l,i)] ∈ grid ?b {d}
      by (rule Suc.hyps)
    thus ?thesis
      by (auto intro!: grid-child[OF ‹d ∈ {d}›, of - b left]
        simp add: child-def)
  next
    assume *: ix b d * 2^(Suc l') < i

    let ?b = child b right d

    have d < length ?b using Suc by auto
    moreover note ‹odd i›
    moreover have lv ?b d + l' = l
      and i < (ix ?b d + 1) * 2^l'
      and (ix ?b d - 1) * 2^l' < i
      unfolding child-ix-right[OF Suc.prems(1)]
      using Suc.prems * child-lv by (auto simp add: field-simps)
    ultimately have ?b[d := (l,i)] ∈ grid ?b {d}
      by (rule Suc.hyps)
    thus ?thesis
      by (auto intro!: grid-child[OF ‹d ∈ {d}›, of - b right]
        simp add: child-def)
  qed
qed

lemma grid-multi-dimensional-specification:
  assumes dm ≤ length b and length p = length b
  and ⋀ d. d < dm ⟹
```

```
    odd (ix p d) ∧
    lv b d ≤ lv p d ∧
    ix p d < (ix b d + 1) * 2^(lv p d - lv b d) ∧
    ix p d > (ix b d - 1) * 2^(lv p d - lv b d)
    (is ⋀ d. d < dm ⟹ ?bounded p d)
  and ⋀ d. ⟦ dm ≤ d ; d < length b ⟧ ⟹ p ! d = b ! d
  shows p ∈ grid b {0..<dm}
using assms proof (induct dm arbitrary: p)
  case 0
  hence p = b by (auto intro!: nth-equalityI)
  thus ?case by auto
next
  case (Suc dm)
  hence dm ≤ length b
    and dm < length p by auto

  let ?p = p[dm := b ! dm]

  note ‹dm ≤ length b›
  moreover have length ?p = length b using ‹length p = length b› by simp
  moreover
  {
    fix d assume d < dm
    hence *: d < Suc dm and dm ≠ d by auto
    have ?p ! d = p ! d
      by (rule nth-list-update-neq[OF ‹dm ≠ d›])
    hence ?bounded ?p d
      using Suc.prems(3)[OF *] lv-def ix-def
      by simp
  }
  moreover
  {
    fix d assume dm ≤ d and d < length b
    have ?p ! d = b ! d
    proof (cases d = dm)
      case True thus ?thesis using ‹d < length b› ‹length p = length b› by auto
    next
      case False
      hence Suc dm ≤ d using ‹dm ≤ d› by auto
      thus ?thesis using Suc.prems(4) ‹d < length b› by auto
    qed
  }
  ultimately
  have *: ?p ∈ grid b {0..<dm}
    by (auto intro!: Suc.hyps)

  have lv b dm ≤ lv p dm using Suc.prems(3)[OF lessI] by simp

  have [simp]: lv ?p dm = lv b dm using lv-def ‹dm < length p› by auto
```

```
  have [simp]: ix ?p dm = ix b dm using ix-def ‹dm < length p› by auto
  have [simp]: p[dm := (lv p dm, ix p dm)] = p
    using lv-def ix-def ‹dm < length p› by auto
  have dm < length ?p and
    [simp]: lv b dm + (lv p dm − lv b dm) = lv p dm
    using ‹dm < length p› ‹lv b dm ≤ lv p dm› by auto
  from grid-single-dimensional-specification[OF this(1),
    where l=lv p dm and i=ix p dm and l'=lv p dm − lv b dm, simplified]
  have p ∈ grid ?p {dm}
    using Suc.prems(3)[OF lessI] by blast
  from grid-transitive[OF this ∗]
  show ?case by auto
qed

lemma sparsegrid:
  sparsegrid dm lm = {p.
    length p = dm ∧ level p < lm ∧
    (∀ d < dm. odd (ix p d) ∧ 0 < ix p d ∧ ix p d < 2^(lv p d + 1))}
  (is - = ?set)
proof (rule equalityI[OF subsetI subsetI])
  fix p
  assume ∗: p ∈ sparsegrid dm lm
  hence length p = dm and level p < lm unfolding sparsegrid-def by auto
  moreover
  { fix d assume d < dm
    hence ∗∗: p ∈ grid (start dm) {0..<dm} and d < length (start dm)
      using ∗ unfolding sparsegrid-def by auto
    have odd (ix p d)
    proof (cases p ! d = start dm ! d)
      case True
      thus ?thesis unfolding start-def using ‹d < dm› ix-def by auto
    next
      case False
      from grid-odd[OF - this ∗∗]
      show ?thesis using ‹d < dm› by auto
    qed
    hence odd (ix p d) ∧ 0 < ix p d ∧ ix p d < 2^(lv p d + 1)
      using grid-estimate[OF ‹d < length (start dm)› ∗∗]
      unfolding ix-def lv-def start-def using ‹d < dm› by auto
  }
  ultimately show p ∈ ?set
    using sparsegrid-def lgrid-def by auto
next
  fix p
  assume p ∈ ?set
  with grid-multi-dimensional-specification[of dm start dm p]
  have p ∈ grid (start dm) {0..<dm} and level p < lm
    by auto
  thus p ∈ sparsegrid dm lm
```

unfolding *sparsegrid-def lgrid-def* **by** *auto*
qed

end

3 Hat Functions

theory *Triangular-Function*
imports *HOL−Analysis.Analysis Grid*
begin

lemma *continuous-on-max*[*continuous-intros*]:
fixes f :: - ⇒ $'a$::*linorder-topology*
shows *continuous-on S f* ⟹ *continuous-on S g* ⟹ *continuous-on S* (λx. *max* (*f x*) (*g x*))
by (*auto simp*: *continuous-on-def intro*: *tendsto-max*)

definition φ :: (*nat* × *int*) ⇒ *real* ⇒ *real* **where**
$\varphi \equiv (\lambda(l,i)\ x.\ max\ 0\ (1 - |\ x * 2\hat{}(l + 1) - real\text{-}of\text{-}int\ i\ |))$

definition Φ :: (*nat* × *int*) *list* ⇒ (*nat* ⇒ *real*) ⇒ *real* **where**
Φ *p x* = (∏ *d*<*length p*. φ (*p* ! *d*) (*x d*))

definition *l2-*φ **where**
l2-φ *p1 p2* = (∫ x. φ *p1 x* ∗ φ *p2 x* ∂*lborel*)

definition *l2* **where**
l2 a b = (∫ x. Φ *a x* ∗ Φ *b x* ∂(Π$_M$ *d*∈{..<*length a*}. *lborel*))

lemma *measurable-*φ[*measurable*]: φ *p* ∈ *borel-measurable borel*
by (*cases p*) (*simp add*: φ*-def*)

lemma φ*-nonneg*: *0* ≤ φ *p x*
by (*simp add*: φ*-def split*: *prod.split*)

lemma φ*-zero-iff*:
φ (*l*,*i*) *x* = *0* ⟷ *x* ∉ {*real-of-int* (*i* − *1*) / *2*^(*l* + *1*) <..< *real-of-int* (*i* + *1*) / *2*^(*l* + *1*)}
by (*auto simp*: φ*-def field-simps split*: *split-max*)

lemma φ*-zero*: *x* ∉ {*real-of-int* (*i* − *1*) / *2*^(*l* + *1*) <..< *real-of-int* (*i* + *1*) / *2*^(*l* + *1*)} ⟹ φ (*l*,*i*) *x* = *0*
unfolding φ*-zero-iff* **by** *simp*

lemma φ*-eq-0*: **assumes** *x*: *x* < *0* ∨ *1* < *x* **and** *i*: *0* < *i i* < *2*^*Suc l* **shows** φ (*l*, *i*) *x* = *0*
using *x*
proof
assume *x* < *0*

also have $0 \leq$ *real-of-int* $(i - 1)$ / $2\hat{}(l + 1)$
using i **by** (*auto simp*: *field-simps*)
finally show *?thesis*
by (*auto intro*!: φ*-zero simp*: *field-simps*)
next
have *real-of-int* $(i + 1)$ / $2\hat{}(l + 1) \leq 1$
using i **by** (*subst divide-le-eq-1-pos*) (*auto simp del*: *of-int-add power-Suc*)
also assume $1 < x$
finally show *?thesis*
by (*auto intro*!: φ*-zero simp*: *field-simps*)
qed

lemma *ix-lt*: $p \in$ *sparsegrid dm lm* $\Longrightarrow d < dm \Longrightarrow$ *ix p d* $<$ $2\hat{}$(*lv p d* + 1)
unfolding *sparsegrid-def lgrid-def*
using *grid-estimate*[*of d start dm p* {0 ..< *dm*}] **by** *auto*

lemma *ix-gt*: $p \in$ *sparsegrid dm lm* $\Longrightarrow d < dm \Longrightarrow 0 <$ *ix p d*
unfolding *sparsegrid-def lgrid-def*
using *grid-estimate*[*of d start dm p* {0 ..< *dm*}] **by** *auto*

lemma Φ*-eq-0*: **assumes** *x*: $\exists\, d$<*length p*. *x d* $< 0 \vee 1 <$ *x d* **and** *p*: $p \in$ *sparsegrid dm lm* **shows** Φ *p x* = 0
unfolding Φ*-def*
proof (*rule prod-zero*)
from *x* **guess** *d* **..**
with *p*[*THEN ix-lt*, *of d*] *p*[*THEN ix-gt*, *of d*] *p*
show $\exists\, a \in$ {..<*length p*}. φ (*p* ! *a*) (*x a*) = 0
apply (*cases p*!*d*)
apply (*intro bexI*[*of* - *d*])
apply (*auto intro*!: φ*-eq-0 simp*: *sparsegrid-length ix-def lv-def*)
done
qed *simp*

lemma φ*-left-support'*:
x $\in$ {*real-of-int* $(i - 1)$ / $2\hat{}(l + 1)$ .. *real-of-int i* / $2\hat{}(l + 1)$} $\Longrightarrow \varphi$ (*l*,*i*) *x* = 1 + *x* $*$ $2\hat{}(l + 1)$ − *real-of-int i*
by (*auto simp*: φ*-def field-simps split*: *split-max*)

lemma φ*-left-support*: *x* $\in$ {−1 .. 0::*real*} $\Longrightarrow \varphi$ (*l*,*i*) ((*x* + *real-of-int i*) / $2\hat{}$(*l* + 1)) = 1 + *x*
by (*auto simp*: φ*-def field-simps split*: *split-max*)

lemma φ*-right-support'*:
x $\in$ {*real-of-int i* / $2\hat{}(l + 1)$ .. *real-of-int* $(i + 1)$ / $2\hat{}(l + 1)$} $\Longrightarrow \varphi$ (*l*,*i*) *x* = 1 − *x* $*$ $2\hat{}(l + 1)$ + *real-of-int i*
by (*auto simp*: φ*-def field-simps split*: *split-max*)

lemma φ*-right-support*:
x $\in$ {0 .. 1::*real*} $\Longrightarrow \varphi$ (*l*,*i*) ((*x* + *real i*) / $2\hat{}(l + 1)$) = 1 − *x*

by (*auto simp*: φ-*def field-simps split*: *split-max*)

lemma *integrable-*φ: *integrable lborel* (φ *p*)
proof (*induct p*)
case (*Pair l i*)
have *integrable lborel* (λx. *indicator* {*real-of-int* (*i* − *1*) / *2*^(*l* + *1*) .. *real-of-int* (*i* + *1*) / *2*^(*l* + *1*)} *x* $*_R$ φ (*l*, *i*) *x*)
unfolding φ-*def* **by** (*intro borel-integrable-compact*) (*auto intro*!: *continuous-intros*)
then show *?case*
by (*rule integrable-cong*[*THEN iffD1*, *rotated* −*1*]) (*auto simp*: φ-*zero-iff*)
qed

lemma *integrable-*φ*2*: *integrable lborel* (λx. φ *p x* * φ *q x*)
proof (*cases p q rule*: *prod.exhaust*[*case-product prod.exhaust*])
case (*Pair-Pair l i l' i'*)
have *integrable lborel*
(λx. *indicator* {*real-of-int* (*i* − *1*) / *2*^(*l* + *1*) .. *real-of-int* (*i* + *1*) / *2*^(*l* + *1*)} *x* $*_R$ (φ (*l*, *i*) *x* * φ (*l'*, *i'*) *x*))
unfolding φ-*def* **by** (*intro borel-integrable-compact*) (*auto intro*!: *continuous-intros*)
then show *?thesis* **unfolding** *Pair-Pair*
by (*rule integrable-cong*[*THEN iffD1*, *rotated* −*1*]) (*auto simp*: φ-*zero-iff*)
qed

lemma *l2-*φ*I-DERIV*:
assumes *n*: $\bigwedge$ *x*. *x* ∈ { (*real-of-int i'* − *1*) / *2*^(*l'* + *1*) .. *real-of-int i'* / *2*^(*l'* + *1*) } ⟹
DERIV Φ-*n x* :> (φ (*l'*, *i'*) *x* * φ (*l*, *i*) *x*) (**is** $\bigwedge$ *x*. *x* ∈ {*?a*..*?b*} ⟹ *DERIV* - - :> *?P x*)
and *p*: $\bigwedge$ *x*. *x* ∈ { *real-of-int i'* / *2*^(*l'* + *1*) .. (*real-of-int i'* + *1*) / *2*^(*l'* + *1*) } ⟹
DERIV Φ-*p x* :> (φ (*l'*, *i'*) *x* * φ (*l*, *i*) *x*) (**is** $\bigwedge$ *x*. *x* ∈ {*?b*..*?c*} ⟹ -)
shows *l2-*φ (*l'*, *i'*) (*l*, *i*) = (Φ-*n ?b* − Φ-*n ?a*) + (Φ-*p ?c* − Φ-*p ?b*)
proof −
have *has-bochner-integral lborel*
(λ*x*. *?P x* * *indicator* {*?a*..*?b*} *x* + *?P x* * *indicator* {*?b*..*?c*} *x*)
((Φ-*n ?b* − Φ-*n ?a*) + (Φ-*p ?c* − Φ-*p ?b*))
by (*intro has-bochner-integral-add has-bochner-integral-FTC-Icc-nonneg n p*)
(*auto simp*: φ-*nonneg field-simps*)
then have *has-bochner-integral lborel?P* ((Φ-*n ?b* − Φ-*n ?a*) + (Φ-*p ?c* − Φ-*p ?b*))
by (*rule has-bochner-integral-discrete-difference*[**where** *X*={*?b*}, *THEN iffD1*, *rotated* −*1*])
(*auto simp*: *power-add intro*!: φ-*zero integral-cong split*: *split-indicator*)
then show *?thesis* **by** (*simp add*: *has-bochner-integral-iff l2-*φ-*def*)
qed

lemma *l2-eq*: *length a* = *length b* ⟹ *l2 a b* = ($\prod$ *d*<*length a*. *l2-*φ (*a*!*d*) (*b*!*d*))
unfolding *l2-def l2-*φ-*def* Φ-*def*
apply (*simp add*: *prod.distrib*[*symmetric*])

proof (*rule product-sigma-finite.product-integral-prod*)
show *product-sigma-finite* (λ*d*. *lborel*) **..**
qed (*auto intro*: *integrable-φ2*)

lemma *l2-when-disjoint*:
assumes $l \leq l'$
defines $d == l' - l$
assumes $(i + 1) * 2\hat{}d < i' \vee i' < (i - 1) * 2\hat{}d$ (**is** *?right* ∨ *?left*)
shows *l2-φ* (l', i') $(l, i) = 0$
proof −
let *?sup* = λ*l i*. {*real-of-int* $(i - 1)$ / $2\hat{}(l + 1)$ <..< *real-of-int* $(i + 1)$ / $2\hat{}(l + 1)$}

have l': $l' = l + d$
using *assms* **by** *simp*
have ∗: ⋀*i l*. $2 \hat{} l$ = *real-of-int* $(2 \hat{} l::int)$
by *simp*
have [*arith*]: $0 < (2\hat{}d::int)$
by *simp*

from ‹*?right* ∨ *?left*› ‹$l \leq l'$› **have** *empty-support*: *?sup l i* ∩ *?sup l′ i′* = {}
by (*auto simp add*: *min-def max-def divide-simps l′ power-add* ∗ *of-int-mult*[*symmetric*]
simp del: *of-int-diff of-int-add of-int-mult of-int-power*)
(*simp-all add*: *field-simps*)
then have ⋀*x*. *φ* (l', i') *x* ∗ *φ* (l, i) *x* = 0
unfolding *φ-zero-iff mult-eq-0-iff* **by** *blast*
then show *?thesis*
by (*simp add*: *l2-φ-def del*: *mult-eq-0-iff vector-space-over-itself.scale-eq-0-iff*)
qed

lemma *l2-commutative*: *l2-φ p q* = *l2-φ q p*
by (*simp add*: *l2-φ-def mult.commute*)

lemma *l2-when-same*: *l2-φ* (l, i) (l, i) = *1/3* / $2\hat{}l$
proof (*subst l2-φI-DERIV*)
let *?l* = $(2 :: real)\hat{}(l + 1)$
let *?in* = *real-of-int i* − *1*
let *?ip* = *real-of-int i* + *1*
let *?φ* = *φ* (l,i)
let *?φ2* = λ*x*. *?φ x* ∗ *?φ x*

{ fix *x* **assume** $x \in \{$*?in* / *?l* .. *real-of-int i* / *?l*$\}$
hence *φ-eq*: *?φ x* = *?l* ∗ *x* − *?in* **using** *φ-left-support′* **by** *auto*
show *DERIV* (λ*x*. $x\hat{}3$ / *3* ∗ *?l*$\hat{}2$ + *x* ∗ *?in*$\hat{}2$ − $x\hat{}2$/*2* ∗ *2* ∗ *?l* ∗ *?in*) *x* :> *?φ2 x*
by (*auto intro*!: *derivative-eq-intros simp add*: *power2-eq-square field-simps φ-eq*) **}**

{ fix *x* **assume** $x \in \{$*real-of-int i* / *?l* .. *?ip* / *?l*$\}$

hence *φ-eq*: *?φ x = ?ip − ?l * x* **using** *φ-right-support′* **by** *auto*
show *DERIV* (*λx. x^3 / 3 * ?l^2 + x * ?ip^2 − x^2/2 * 2 * ?l * ?ip*) *x :> ?φ2 x*
by (*auto intro*!: *derivative-eq-intros simp add*: *power2-eq-square field-simps φ-eq*) **}**
qed (*simp-all add*: *field-simps power-eq-if*[*of - 2*] *power-eq-if*[*of - 3*])

lemma *l2-when-left-child*:
assumes *l < l′*
and *i′-bot*: *i′ > (i − 1) * 2^(l′ − l)*
and *i′-top*: *i′ < i * 2^(l′ − l)*
shows *l2-φ* (*l′*, *i′*) (*l*, *i*) = (*1 + real-of-int i′ / 2^(l′ − l) − real-of-int i*) / *2^(l′ + 1)*
proof (*subst l2-φI-DERIV*)
let *?l′ = (2 :: real)^(l′ + 1)*
let *?in′ = real-of-int i′ − 1*
let *?ip′ = real-of-int i′ + 1*
let *?l = (2 :: real)^(l + 1)*
let *?i = real-of-int i − 1*
let *?φ′ = φ* (*l′*,*i′*)
let *?φ = φ* (*l*, *i*)
let *?φ2 x = ?φ′ x * ?φ x*
define *Φ-n* **where** *Φ-n x = x^3 / 3 * ?l′ * ?l + x * ?i * ?in′ − x^2 / 2 * (?in′ * ?l + ?i * ?l′)* **for** *x*
define *Φ-p* **where** *Φ-p x = x^2 / 2 * (?ip′ * ?l + ?i * ?l′) − x^3 / 3 * ?l′ * ?l − x * ?i * ?ip′* **for** *x*

have *level-diff*: *2^(l′ − l) = 2^l′ / (2^l :: real)* **using** *power-diff*[*of 2*::*real l l′*] ‹*l < l′*› **by** *auto*

{ fix *x* **assume** *x*: *x ∈ {?in′ / ?l′ .. ?ip′ / ?l′}*
have *?i * 2^(l′ − l) ≤ ?in′*
using *i′-bot int-less-real-le* **by** *auto*
hence *?i / ?l ≤ ?in′ / ?l′*
using *level-diff* **by** (*auto simp*: *field-simps*)
hence *?i / ?l ≤ x* **using** *x* **by** *auto*
moreover
have *?ip′ ≤ real-of-int i * 2^(l′ − l)*
using *i′-top int-less-real-le* **by** *auto*
hence *ip′-le-i*: *?ip′ / ?l′ ≤ real-of-int i / ?l*
using *level-diff* **by** (*auto simp*: *field-simps*)
hence *x ≤ real-of-int i / ?l* **using** *x* **by** *auto*
ultimately have *?φ x = ?l * x − ?i* **using** *φ-left-support′* **by** *auto*
} note *φ-eq = this*

{ fix *x* **assume** *x*: *x ∈ {?in′ / ?l′ .. real-of-int i′ / ?l′}*
hence *φ′-eq*: *?φ′ x = ?l′ * x − ?in′* **using** *φ-left-support′* **by** *auto*
from *x* **have** *x′*: *x ∈ {?in′ / ?l′ .. ?ip′ / ?l′}* **by** (*auto simp add*: *field-simps*)
show *DERIV Φ-n x :> ?φ2 x* **unfolding** *φ-eq*[*OF x′*] *φ′-eq Φ-n-def*

by (*auto intro*!: *derivative-eq-intros simp add*: *power2-eq-square algebra-simps*)
}

{ fix x **assume** x: $x \in \{real\text{-}of\text{-}int\ i' \mathbin{/} ?l' .. ?ip' \mathbin{/} ?l'\}$
hence φ'-*eq*: $?\varphi'\ x = ?ip' - ?l' * x$ **using** φ-*right-support'* **by** *auto*
from x **have** x': $x \in \{?in' \mathbin{/} ?l' .. ?ip' \mathbin{/} ?l'\}$ **by** (*simp add*: *field-simps*)
show *DERIV* Φ-$p\ x :> ?\varphi 2\ x$ **unfolding** φ-*eq*[*OF x'*] φ'-*eq* Φ-*p-def*
by (*auto intro*!: *derivative-eq-intros simp add*: *power2-eq-square algebra-simps*)
}
qed (*simp-all add*: *field-simps power-eq-if*[*of - 2*] *power-eq-if*[*of - 3*] *power-diff*[*of 2*::*real*, *OF - ‹l < l'›*[*THEN less-imp-le*]])

lemma *l2-when-right-child*:
assumes $l < l'$
and i'-*bot*: $i' > i * 2\hat{}(l' - l)$
and i'-*top*: $i' < (i + 1) * 2\hat{}(l' - l)$
shows $l2\text{-}\varphi\ (l', i')\ (l, i) = (1 - real\text{-}of\text{-}int\ i' \mathbin{/} 2\hat{}(l' - l) + real\text{-}of\text{-}int\ i) \mathbin{/} 2\hat{}(l' + 1)$
proof (*subst l2-φI-DERIV*)
let $?l' = (2 :: real)\hat{}(l' + 1)$
let $?in' = real\text{-}of\text{-}int\ i' - 1$
let $?ip' = real\text{-}of\text{-}int\ i' + 1$
let $?l = (2 :: real)\hat{}(l + 1)$
let $?i = real\text{-}of\text{-}int\ i + 1$
let $?\varphi' = \varphi\ (l',i')$
let $?\varphi = \varphi\ (l, i)$
let $?\varphi 2 = \lambda x.\ ?\varphi'\ x * ?\varphi\ x$
define Φ-n **where** $\Phi\text{-}n\ x = x\hat{}2 \mathbin{/} 2 * (?in' * ?l + ?i * ?l') - x\hat{}3 \mathbin{/} 3 * ?l' * ?l - x * ?i * ?in'$ **for** x
define Φ-p **where** $\Phi\text{-}p\ x = x\hat{}3 \mathbin{/} 3 * ?l' * ?l + x * ?i * ?ip' - x\hat{}2 \mathbin{/} 2 * (?ip' * ?l + ?i * ?l')$ **for** x

have *level-diff*: $2\hat{}(l' - l) = 2\hat{}l' \mathbin{/} (2\hat{}l :: real)$ **using** *power-diff*[*of 2*::*real l l'*] ‹$l < l'$› **by** *auto*

{ fix x **assume** x: $x \in \{?in' \mathbin{/} ?l' .. ?ip' \mathbin{/} ?l'\}$
have $real\text{-}of\text{-}int\ i * 2\hat{}(l' - l) \leq ?in'$
using i'-*bot int-less-real-le* **by** *auto*
hence $real\text{-}of\text{-}int\ i \mathbin{/} ?l \leq ?in' \mathbin{/} ?l'$
using *level-diff* **by** (*auto simp*: *field-simps*)
hence $real\text{-}of\text{-}int\ i \mathbin{/} ?l \leq x$ **using** x **by** *auto*
moreover
have $?ip' \leq ?i * 2\hat{}(l' - l)$
using i'-*top int-less-real-le* **by** *auto*
hence *ip'-le-i*: $?ip' \mathbin{/} ?l' \leq ?i \mathbin{/} ?l$
using *level-diff* **by** (*auto simp*: *field-simps*)
hence $x \leq ?i \mathbin{/} ?l$ **using** x **by** *auto*
ultimately have $?\varphi\ x = ?i - ?l * x$ **using** φ-*right-support'* **by** *auto*
} **note** φ-*eq* = *this*

```
  { fix x assume x: x ∈ {?in' / ?l' .. real-of-int i' / ?l'}
    hence φ'-eq: ?φ' x = ?l' * x  - ?in' using φ-left-support' by auto

    from x have x': x ∈ {?in' / ?l' .. ?ip' / ?l'} by (simp add: field-simps)
    show DERIV Φ-n x :> ?φ2 x unfolding Φ-n-def φ-eq[OF x'] φ'-eq
       by (auto intro!: derivative-eq-intros simp add: simp add: power2-eq-square
algebra-simps) }

  { fix x assume x: x ∈ {real-of-int i' / ?l' .. ?ip' / ?l'}
    hence φ'-eq: ?φ' x = ?ip' - ?l' * x using φ-right-support' by auto
    from x have x': x ∈ {?in' / ?l' .. ?ip' / ?l'} by (auto simp: field-simps)
    show DERIV Φ-p x :> ?φ2 x unfolding φ-eq[OF x'] φ'-eq Φ-p-def
     by (auto intro!: derivative-eq-intros simp add: power2-eq-square algebra-simps)
}
qed (simp-all add: field-simps power-eq-if[of - 2] power-eq-if[of - 3] power-diff[of
2::real, OF - ‹l < l'›[THEN less-imp-le]] )

lemma level-shift: lc > l ⟹ (x :: real) / 2 ^ (lc - Suc l) = x * 2 / 2 ^ (lc - l)
  by (auto simp add: power-diff)

lemma l2-child: assumes d < length b
  and p-grid: p ∈ grid (child b dir d) ds (is p ∈ grid ?child ds)
  shows l2-φ (p ! d) (b ! d) = (1 - real-of-int (sgn dir) * (real-of-int (ix p d) /
2^(lv p d - lv b d) - real-of-int (ix b d))) /
                                2^(lv p d + 1)
proof -
  have lv ?child d ≤ lv p d using ‹d < length b› and p-grid
    using grid-single-level by auto
  hence lv b d < lv p d using ‹d < length b› and p-grid
    using child-lv by auto

  let ?i-c = ix ?child d and ?l-c = lv ?child d
  let ?i-p = ix p d and ?l-p = lv p d
  let ?i-b = ix b d and ?l-b = lv b d

  have (2::int) * 2^(?l-p - ?l-c) = 2^Suc (?l-p - ?l-c) by auto
  also have ... = 2^(Suc ?l-p - ?l-c)
  proof -
    have Suc (?l-p - ?l-c) = Suc ?l-p - ?l-c
      using ‹lv ?child d ≤ lv p d› by auto
    thus ?thesis by auto
  qed
  also have ... = 2^(?l-p - ?l-b)
    using ‹d < length b› and ‹lv b d < lv p d›
    by (auto simp add: child-def lv-def)
  finally have level: 2^(?l-p - ?l-b) = (2::int) * 2^(?l-p - ?l-c) ..

  from ‹d < length b› and p-grid
```

```
  have range-left: ?i-p > (?i-c − 1) * 2^(?l-p − ?l-c) and
      range-right: ?i-p < (?i-c + 1) * 2^(?l-p − ?l-c)
    using grid-estimate by auto

  show ?thesis
  proof (cases dir)
    case left
    with child-ix-left[OF ‹d < length b›]
    have (?i-b − 1) * 2^(?l-p − ?l-b) = (?i-c − 1) * 2^(?l-p − ?l-c) and
      ?i-b * 2^(?l-p − ?l-b) = (?i-c + 1) * 2^(?l-p − ?l-c) using level by auto
    hence ?i-p > (?i-b − 1) * 2^(?l-p − ?l-b) and
      ?i-p < ?i-b * 2^(?l-p − ?l-b)
      using range-left and range-right by auto
    with ‹?l-b < ?l-p›
    have l2-φ (?l-p, ?i-p) (?l-b, ?i-b) =
        (1 + real-of-int ?i-p / 2^(?l-p − ?l-b) − real-of-int ?i-b) / 2^(?l-p + 1)
      by (rule l2-when-left-child)
    thus ?thesis using left by (auto simp add: ix-def lv-def)
  next
    case right
    hence ?i-c = 2 * ?i-b + 1 using child-ix-right and ‹d < length b› by auto
    hence ?i-b * 2^(?l-p − ?l-b) = (?i-c − 1) * 2^(?l-p − ?l-c) and
      (?i-b + 1) * 2^(?l-p − ?l-b) = (?i-c + 1) * 2^(?l-p − ?l-c) using level by
auto
    hence ?i-p > ?i-b * 2^(?l-p − ?l-b) and
      ?i-p < (?i-b + 1) * 2^(?l-p − ?l-b)
      using range-left and range-right by auto
    with ‹?l-b < ?l-p›
    have l2-φ (?l-p, ?i-p) (?l-b, ?i-b) =
        (1 − real-of-int ?i-p / 2^(?l-p − ?l-b) + real-of-int ?i-b) / 2^(?l-p + 1)
      by (rule l2-when-right-child)
    thus ?thesis using right by (auto simp add: ix-def lv-def)
  qed
qed

lemma l2-same: l2-φ (p!d) (p!d) = 1/3 / 2^(lv p d)
proof −
  have l2-φ (p!d) (p!d) = l2-φ (lv p d, ix p d) (lv p d, ix p d)
    by (auto simp add: lv-def ix-def)
  thus ?thesis using l2-when-same by auto
qed

lemma l2-disjoint: assumes d < length b and p ∈ grid b {d} and p' ∈ grid b
{d}
  and p' ∉ grid p {d} and lv p' d ≥ lv p d
  shows l2-φ (p' ! d) (p ! d) = 0
proof −
  have range: ix p' d > (ix p d + 1) * 2^(lv p' d − lv p d) ∨ ix p' d < (ix p d −
1) * 2^(lv p' d − lv p d)
```

proof (*rule ccontr*)
assume ¬ *?thesis*
hence *ix p′ d* ≤ (*ix p d* + *1*) ∗ *2^*(*lv p′ d* − *lv p d*) **and** *ix p′ d* ≥ (*ix p d* − *1*) ∗ *2^*(*lv p′ d* − *lv p d*) **by** *auto*
with ‹*p′* ∈ *grid b* {*d*}› **and** ‹*p* ∈ *grid b* {*d*}› **and** ‹*lv p′ d* ≥ *lv p d*› **and** ‹*d* < *length b*›
have *p′* ∈ *grid p* {*d*} **using** *grid-part*[**where** *p*=*p* **and** *b*=*b* **and** *d*=*d* **and** *p′*=*p′*] **by** *auto*
with ‹*p′* ∉ *grid p* {*d*}› **show** *False* **by** *auto*
qed

have *l2-φ* (*p′* ! *d*) (*p* ! *d*) = *l2-φ* (*lv p′ d*, *ix p′ d*) (*lv p d*, *ix p d*) **by** (*auto simp add*: *ix-def lv-def*)
also have ... = *0* **using** *range* **and** ‹*lv p′ d* ≥ *lv p d*› **and** *l2-when-disjoint* **by** *auto*
finally show *?thesis* **.**
qed

lemma *l2-down2*:
fixes *pc pd p*
assumes *d* < *length pd*
assumes *pc-in-grid*: *pc* ∈ *grid* (*child pd dir d*) {*d*}
assumes *pd-is-child*: *pd* = *child p dir d* (**is** *pd* = *?pd*)
shows *l2-φ* (*pc* ! *d*) (*pd* ! *d*) / *2* = *l2-φ* (*pc* ! *d*) (*p* ! *d*)
proof −
have *d* < *length p* **using** *pd-is-child* ‹*d* < *length pd*› **by** *auto*

moreover
have *pc* ∈ *grid ?pd* {*d*} **using** *pd-is-child* **and** *grid-child* **and** *pc-in-grid* **by** *auto*
hence *lv p d* < *lv pc d* **using** *grid-child-level* **and** ‹*d* < *length pd*› **and** *pd-is-child* **by** *auto*

moreover
have *real-of-int* (*sgn dir*) ∗ *real-of-int* (*sgn dir*) = *1* **by** (*cases dir*, *auto*)

ultimately show *?thesis*
unfolding *l2-child*[*OF* ‹*d* < *length pd*› *pc-in-grid*]
l2-child[*OF* ‹*d* < *length p*› ‹*pc* ∈ *grid ?pd* {*d*}›]
using *child-lv* **and** *child-ix* **and** *pd-is-child* **and** *level-shift*
by (*auto simp add*: *algebra-simps diff-divide-distrib add-divide-distrib*)
qed

lemma *l2-zigzag*:
assumes *d* < *length p* **and** *p-child*: *p* = *child p-p dir d*
and *p′-grid*: *p′* ∈ *grid* (*child p* (*inv dir*) *d*) {*d*}
and *ps-intro*: *child p* (*inv dir*) *d* = *child ps dir d* (**is** *?c-p* = *?c-ps*)
shows *l2-φ* (*p′* ! *d*) (*p-p* ! *d*) = *l2-φ* (*p′* ! *d*) (*ps* ! *d*) + *l2-φ* (*p′* ! *d*) (*p* ! *d*) / *2*
proof −
have *length p* = *length ?c-p* **by** *auto*

also have ... = *length ?c-ps* **using** *ps-intro* **by** *auto*
finally have *length p* = *length ps* **using** *ps-intro* **by** *auto*
hence *d* < *length p-p* **using** *p-child* **and** ‹*d* < *length p*› **by** *auto*

moreover
from *ps-intro* **have** *ps* = *p*[*d* := (*lv p d*, *ix p d* − *sgn dir*)] **by** (*rule child-neighbour*)
hence *lv ps d* = *lv p d* **and** *real-of-int* (*ix ps d*) = *real-of-int* (*ix p d*) − *real-of-int* (*sgn dir*)
using *lv-def* **and** *ix-def* **and** ‹*length p* = *length ps*› **and** ‹*d* < *length p*› **by** *auto*

moreover
have *d* < *length ps* **and** ∗: *p'* ∈ *grid* (*child ps dir d*) {*d*}
using *p'-grid ps-intro* ‹*length p* = *length ps*› ‹*d* < *length p*› **by** *auto*

have *p'* ∈ *grid p* {*d*} **using** *p'-grid* **and** *grid-child* **by** *auto*
hence *p-p-grid*: *p'* ∈ *grid* (*child p-p dir d*) {*d*} **using** *p-child* **by** *auto*
hence *lv p' d* > *lv p-p d* **using** *grid-child-level* **and** ‹*d* < *length p-p*› **by** *auto*

moreover
have *real-of-int* (*sgn dir*) ∗ *real-of-int* (*sgn dir*) = *1* **by** (*cases dir*, *auto*)

ultimately show *?thesis*
unfolding *l2-child*[*OF* ‹*d* < *length p*› *p'-grid*] *l2-child*[*OF* ‹*d* < *length ps*› ∗]
l2-child[*OF* ‹*d* < *length p-p*› *p-p-grid*]
using *child-lv* **and** *child-ix* **and** *p-child level-shift*
by (*auto simp add*: *add-divide-distrib algebra-simps diff-divide-distrib*)
qed

end

4 UpDown Scheme

theory *UpDown-Scheme*
imports *Grid*
begin

fun *down'* :: *nat* ⇒ *nat* ⇒ *grid-point* ⇒ *real* ⇒ *real* ⇒ *vector* ⇒ *vector*
where
down' d 0 p f$_l$ *f*$_r$ *α* = *α*
| *down' d* (*Suc l*) *p f*$_l$ *f*$_r$ *α* = (*let*
f$_m$ = (*f*$_l$ + *f*$_r$) / *2* + (*α p*);
α = *α*(*p* := ((*f*$_l$ + *f*$_r$) / *4* + (*1* / *3*) ∗ (*α p*)) / *2* ^ (*lv p d*));
α = *down' d l* (*child p left d*) *f*$_l$ *f*$_m$ *α*;
α = *down' d l* (*child p right d*) *f*$_m$ *f*$_r$ *α*
in α)

definition *down* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *vector* ⇒ *vector* **where**
down = *lift* (*λ d l p*. *down' d l p 0 0*)

fun *up′* :: *nat* ⇒ *nat* ⇒ *grid-point* ⇒ *vector* ⇒ (*real* ∗ *real*) ∗ *vector* **where**
  up′ d 0 p α = ((*0*, *0*), *α*)
| *up′ d* (*Suc l*) *p α* = (*let*
    ((*f*$_l$, *f*$_{ml}$), *α*) = *up′ d l* (*child p left d*) *α*;
    ((*f*$_{mr}$, *f*$_r$), *α*) = *up′ d l* (*child p right d*) *α*;
    result = (*f*$_{ml}$ + *f*$_{mr}$ + (*α p*) / *2* ^ (*lv p d*) / *2*) / *2*
  in ((*f*$_l$ + *result*, *f*$_r$ + *result*), *α*(*p* := *f*$_{ml}$ + *f*$_{mr}$)))

definition *up* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *vector* ⇒ *vector* **where**
  up = *lift* (λ *d lm p α*. *snd* (*up′ d lm p α*))

fun *updown′* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *vector* ⇒ *vector* **where**
  updown′ dm lm 0 α = *α*
| *updown′ dm lm* (*Suc d*) *α* =
  (*sum-vector* (*updown′ dm lm d* (*up dm lm d α*)) (*down dm lm d* (*updown′ dm lm d α*)))

definition *updown* :: *nat* ⇒ *nat* ⇒ *vector* ⇒ *vector* **where**
  updown dm lm α = *updown′ dm lm dm α*

end

5 Up Part

theory *Up*
imports *UpDown-Scheme Triangular-Function*
begin

lemma *up′-inplace*:
  assumes *p′-in*: *p′* ∉ *grid p ds* **and** *d* ∈ *ds*
  shows *snd* (*up′ d l p α*) *p′* = *α p′*
  using *p′-in*
proof (*induct l arbitrary*: *p α*)
  case (*Suc l*)
  let *?ch dir* = *child p dir d*
  let *?up dir α* = *up′ d l* (*?ch dir*) *α*
  let *?upl* = *snd* (*?up left α*)

  from *contrapos-nn*[*OF* ‹*p′* ∉ *grid p ds*› *grid-child*[*OF* ‹*d* ∈ *ds*›]]
  have *left*: *p′* ∉ *grid* (*?ch left*) *ds* **and**
      right: *p′* ∉ *grid* (*?ch right*) *ds* **by** *auto*

  have *p* ≠ *p′* **using** *grid.Start Suc.prems* **by** *auto*
  with *Suc.hyps*[*OF left*, *of α*] *Suc.hyps*[*OF right*, *of ?upl*]
  show *?case*
    by (*cases ?up left α*, *cases ?up right ?upl*, *auto simp add*: *Let-def*)
qed *auto*

```
lemma up'-fl-fr:
    ⟦ d < length p ; p = (child p-r right d) ; p = (child p-l left d) ⟧
     ⟹ fst (up' d lm p α) =
          (∑ p' ∈ lgrid p {d} (lm + level p). (α p') * l2-φ (p' ! d) (p-r ! d),
           ∑ p' ∈ lgrid p {d} (lm + level p). (α p') * l2-φ (p' ! d) (p-l ! d))
proof (induct lm arbitrary: p p-l p-r α)
  case (Suc lm)
  note ‹d < length p›[simp]

  from child-ex-neighbour
  obtain pc-r pc-l
    where pc-r-def: child p right d = child pc-r (inv right) d
    and pc-l-def: child p left d = child pc-l (inv left) d by blast

  define pc where pc dir = (case dir of right ⇒ pc-r | left ⇒ pc-l) for dir
  { fix dir have child p (inv dir) d = child (pc (inv dir)) dir d
     by (cases dir, auto simp add: pc-def pc-r-def pc-l-def) } note pc-child = this
  { fix dir have child p dir d = child (pc dir) (inv dir) d
     by (cases dir, auto simp add: pc-def pc-r-def pc-l-def) } note pc-child-inv =
this
  hence !! dir. length (child p dir d) = length (child (pc dir) (inv dir) d) by auto
  hence !! dir. length p = length (pc dir) by auto
  hence [simp]: !! dir. d < length (pc dir) by auto

  let ?l = λs. lm + level s
  let ?C = λp p'. (α p) * l2-φ (p ! d) (p' ! d)
  let ?sum' = λs p''. ∑ p' ∈ lgrid s {d} (Suc lm + level p). ?C p' p''
  let ?sum = λs dir p. ∑ p' ∈ lgrid (child s dir d) {d} (?l (child s dir d)). ?C p'
p
  let ?ch = λdir. child p dir d
  let ?f = λdir. ?sum p dir (pc dir)
  let ?fm = λdir. ?sum p dir p
  let ?result = (?fm left + ?fm right + (α p) / 2 ^ (lv p d) / 2) / 2
  let ?up = λlm p α. up' d lm p α

  define βl where βl = snd (?up lm (?ch left) α)
  define βr where βr = snd (?up lm (?ch right) βl)

  define p-d where p-d dir = (case dir of right ⇒ p-r | left ⇒ p-l) for dir
  have p-d-child: p = child (p-d dir) dir d for dir
    using Suc.prems p-d-def by (cases dir) auto
  hence ⋀ dir. length p = length (child (p-d dir) dir d) by auto
  hence ⋀ dir. d < length (p-d dir) by auto

  { fix dir

    { fix p' assume p' ∈ lgrid (?ch (inv dir)) {d} (?l (?ch (inv dir)))
      hence ?C p' (pc (inv dir)) + (?C p' p) / 2 = ?C p' (p-d dir)
        using l2-zigzag[OF - p-d-child[of dir] - pc-child[of dir]]
```

```
      by (cases dir) (auto simp add: algebra-simps) }
    hence inv-dir-sum: ?sum p (inv dir) (pc (inv dir)) + (?sum p (inv dir) p) / 2
      = ?sum p (inv dir) (p-d dir)
      by (auto simp add: sum.distrib[symmetric] sum-divide-distrib)

    have ?sum p dir p / 2 = ?sum p dir (p-d dir)
      using l2-down2[OF - - ‹p = child (p-d dir) dir d›]
      by (force intro!: sum.cong simp add: sum-divide-distrib)
    moreover
    have ?C p (p-d dir) = (α p) / 2 ^ (lv p d) / 4
      using l2-child[OF ‹d < length (p-d dir)›, of p dir {d}] p-d-child[of dir]
      ‹d < length (p-d dir)› child-lv child-ix grid.Start[of p {d}]
      by (cases dir) (auto simp add: add-divide-distrib field-simps)
    ultimately
    have ?sum' p (p-d dir) =
      ?sum p (inv dir) (pc (inv dir)) +
      (?sum p (inv dir) p) / 2 + ?sum p dir p / 2 + (α p) / 2 ^ (lv p d) / 4
      using lgrid-sum[where b=p] and child-level and inv-dir-sum
      by (cases dir) auto
    hence ?sum p (inv dir) (pc (inv dir)) + ?result = ?sum' p (p-d dir)
      by (cases dir) auto }
  note this[of left] this[of right]
  moreover
  note eq = up'-inplace[OF grid-not-child[OF ‹d < length p›], of d {d} lm]
  { fix p' assume p' ∈ lgrid (?ch right) {d} (lm + level (?ch right))
    with grid-disjunct[of d p] up'-inplace[of p' ?ch left {d} d lm α] βl-def
    have βl p' = α p' by auto }
  hence fst (?up (Suc lm) p α) = (?f left + ?result, ?f right + ?result)
    using βl-def pc-child-inv[of left] pc-child-inv[of right]
      Suc.hyps[of ?ch left pc left p α] eq[of left α]
      Suc.hyps[of ?ch right p pc right βl] eq[of right βl]
    by (cases ?up lm (?ch left) α, cases ?up lm (?ch right) βl) (simp add: Let-def)
  ultimately show ?case by (auto simp add: p-d-def)
next
  case 0
  show ?case by simp
qed

lemma up'-β:
  ⟦ d < length b ; l + level b = lm ; b ∈ sparsegrid' dm ; p ∈ sparsegrid' dm ⟧
  ⟹
  (snd (up' d l b α)) p =
    (if p ∈ lgrid b {d} lm
     then ∑ p' ∈ (lgrid p {d} lm) − {p}. α p' * l2-φ (p' ! d) (p ! d)
     else α p)
  (is ⟦ - ; - ; - ; - ⟧ ⟹ (?goal l b p α))
proof (induct l arbitrary: b p α)
  case (Suc l)
```

let *?l* = *child b left d* **and** *?r* = *child b right d*
obtain *p-l* **where** *p-l-def*: *?r* = *child p-l left d* **using** *child-ex-neighbour*[**where** *dir=right*] **by** *auto*
obtain *p-r* **where** *p-r-def*: *?l* = *child p-r right d* **using** *child-ex-neighbour*[**where** *dir=left*] **by** *auto*

let *?ul* = *up′ d l ?l* α
let *?ur* = *up′ d l ?r* (*snd ?ul*)

let *?C p′* = α *p′* ∗ *l2-φ* (*p′* ! *d*) (*p* ! *d*)
let *?s s* = $\sum$ *p′* ∈ (*lgrid s* {*d*} *lm*). *?C p′*

from ‹*b* ∈ *sparsegrid′ dm*› **have** *length b* = *dm* **unfolding** *sparsegrid′-def start-def*
by *auto*
hence *d* < *dm* **using** ‹*d* < *length b*› **by** *auto*

{ fix *p′* **assume** *p′* ∈ *grid ?r* {*d*}
hence *p′* ∉ *grid ?l* {*d*}
using *grid-disjunct*[*OF* ‹*d* < *length b*›] **by** *auto*
hence *snd ?ul p′* = α *p′* **using** *up′-inplace* **by** *auto*
} note *eq* = *this*

show *?goal* (*Suc l*) *b p* α
proof (*cases p* = *b*)
case *True*

let *?C p′* = α *p′* ∗ *l2-φ* (*p′* ! *d*) (*b* ! *d*)
let *?s s* = $\sum$ *p′* ∈ (*lgrid s* {*d*} *lm*). *?C p′*

have *d* < *length ?l* **using** ‹*d* < *length b*› **by** *auto*
from *up′-fl-fr*[*OF this p-r-def*]
have *fml*: *snd* (*fst ?ul*) = ($\sum$ *p′* ∈ *lgrid ?l* {*d*} (*l* + *level ?l*). *?C p′*) **by** *simp*

have *d* < *length ?r* **using** ‹*d* < *length b*› **by** *auto*
from *up′-fl-fr*[*OF this - p-l-def*, **where** α=*snd ?ul*]
have *fmr*: *fst* (*fst ?ur*) = ($\sum$ *p′* ∈ *lgrid ?r* {*d*} (*l* + *level ?r*).
((*snd ?ul*) *p′*) ∗ *l2-φ* (*p′* ! *d*) (*b* ! *d*)) **by** *simp*

have *level b* < *lm* **using** ‹*Suc l* + *level b* = *lm*› **by** *auto*
hence { *b* } ⊆ *lgrid b* {*d*} *lm* **unfolding** *lgrid-def* **by** *auto*
from *sum-diff*[*OF lgrid-finite this*]
have ($\sum$ *p′* ∈ (*lgrid b* {*d*} *lm*) − {*b*}. *?C p′*) = *?s b* − *?C b* **by** *simp*
also have ... = *?s ?l* + *?s ?r*
using *lgrid-sum* **and** ‹*level b* < *lm*› **and** ‹*d* < *length b*› **by** *auto*
also have ... = *snd* (*fst ?ul*) + *fst* (*fst ?ur*) **using** *fml* **and** *fmr*
and ‹*Suc l* + *level b* = *lm*› **and** *child-level*[*OF* ‹*d* < *length b*›]
using *eq* **unfolding** *True lgrid-def* **by** *auto*

```
    finally show ?thesis unfolding up'.simps Let-def and fun-upd-def lgrid-def
      using ‹p = b› and ‹level b < lm›
      by (cases ?ul, cases ?ur, auto)
  next
    case False

    have ?r ∈ sparsegrid' dm and ?l ∈ sparsegrid' dm
     using ‹b ∈ sparsegrid' dm› and ‹d < dm› unfolding sparsegrid'-def by auto
    from Suc.hyps[OF - - this(1)] Suc.hyps[OF - - this(2)]
    have ?goal l ?l p α and ?goal l ?r p (snd ?ul)
      using ‹d < length b› and ‹Suc l + level b = lm› and ‹p ∈ sparsegrid' dm›
by auto

    show ?thesis
    proof (cases p ∈ lgrid b {d} lm)
      case True
      hence level p < lm and p ∈ grid b {d} unfolding lgrid-def by auto
      hence p ∈ grid ?l {d} ∨ p ∈ grid ?r {d}
        unfolding grid-partition[of b] using ‹p ≠ b› by auto
      thus ?thesis
      proof (rule disjE)
        assume p ∈ grid (child b left d) {d}
        hence p ∉ grid (child b right d) {d}
          using grid-disjunct[OF ‹d < length b›] by auto
        thus ?thesis
          using ‹?goal l ?l p α› and ‹?goal l ?r p (snd ?ul)›
          using ‹p ≠ b› ‹p ∈ lgrid b {d} lm›
          unfolding lgrid-def grid-partition[of b]
          by (cases ?ul, cases ?ur, auto simp add: Let-def)
      next
        assume ∗: p ∈ grid (child b right d) {d}
        hence p ∉ grid (child b left d) {d}
          using grid-disjunct[OF ‹d < length b›] by auto
        moreover
        { fix p' assume p' ∈ grid p {d}
          from grid-transitive[OF this ∗] eq[of p']
          have snd ?ul p' = α p' by simp
        }
        ultimately show ?thesis
          using ‹?goal l ?l p α› and ‹?goal l ?r p (snd ?ul)›
          using ‹p ≠ b› ‹p ∈ lgrid b {d} lm› ∗
          unfolding lgrid-def
          by (cases ?ul, cases ?ur, auto simp add: Let-def)
  qed
next
      case False
      then have p ∉ lgrid ?l {d} lm and p ∉ lgrid ?r {d} lm
        unfolding lgrid-def and grid-partition[where p=b] by auto
      with False show ?thesis using ‹?goal l ?l p α› and ‹?goal l ?r p (snd ?ul)›
```

```
      using ‹p ≠ b› ‹p ∉ lgrid b {d} lm›
      unfolding lgrid-def
      by (cases ?ul, cases ?ur, auto simp add: Let-def)
    qed
  qed
next
  case 0
  then have lgrid b {d} lm = {}
    using lgrid-empty'[where p=b and lm=lm and ds={d}] by auto
  with 0 show ?case unfolding up'.simps by auto
qed

lemma up:
  assumes d < dm and p ∈ sparsegrid dm lm
  shows (up dm lm d α) p = (∑ p' ∈ (lgrid p {d} lm) - {p}. α p' * l2-φ (p' ! d) (p ! d))
proof -
  let ?S = λ x p p'. if p' ∈ grid p {d} - {p} then x * l2-φ (p'!d) (p!d) else 0
  let ?F = λ d lm p α. snd (up' d lm p α)

  { fix p b assume p ∈ grid b {d}
    from grid-transitive[OF - this subset-refl subset-refl]
    have lgrid b {d} lm ∩ (grid p {d} - {p}) = lgrid p {d} lm - {p}
      unfolding lgrid-def by auto
  } note lgrid-eq = this

  { fix l b p α
    assume b: b ∈ lgrid (start dm) ({0..<dm} - {d}) lm
    hence b ∈ sparsegrid' dm and d < length b using sparsegrid'-start ‹d < dm›
by auto
    assume l: l + level b = lm and p: p ∈ sparsegrid dm lm
    note sparsegridE[OF p]

    note up' = up'-β[OF ‹d < length b› l ‹b ∈ sparsegrid' dm› ‹p ∈ sparsegrid'
dm›]

    have ?F d l b α p =
        (if b = base {d} p then (∑p'∈lgrid b {d} lm. ?S (α p') p p') else α p)
    proof (cases b = base {d} p)
      case True with baseE(2)[OF ‹p ∈ sparsegrid' dm›] ‹level p < lm›
      have p ∈ lgrid b {d} lm and p ∈ grid b {d} by auto
      show ?thesis
        using lgrid-eq[OF ‹p ∈ grid b {d}›]
        unfolding up' if-P[OF True] if-P[OF ‹p ∈ lgrid b {d} lm›]
        by (intro sum.mono-neutral-cong-left lgrid-finite) auto
    next
      case False
      moreover have p ∉ lgrid b {d} lm
      proof (rule ccontr)
```

assume ¬ *?thesis*
hence *base* {*d*} *p* = *b* **using** *b* **by** (*auto intro*!: *baseI*)
thus *False* **using** *False* **by** *auto*
qed
ultimately show *?thesis* **unfolding** *up′* **by** *auto*
qed }
with *lift*[**where** *F* = *?F*, *OF* ‹*d* < *dm*› ‹*p* ∈ *sparsegrid dm lm*›]
have *lift-eq*: *lift ?F dm lm d α p* =
(∑ *p′*∈*lgrid* (*base* {*d*} *p*) {*d*} *lm*. *?S* (*α p′*) *p p′*) **by** *auto*
from *lgrid-eq*[*OF baseE*(*2*)[*OF sparsegrid-subset*[*OF* ‹*p* ∈ *sparsegrid dm lm*›]]]
show *?thesis*
unfolding *up-def lift-eq* **by** (*intro sum.mono-neutral-cong-right lgrid-finite*) *auto*
qed

end

6 Down part

theory *Down*
imports *Triangular-Function UpDown-Scheme*
begin

lemma *sparsegrid′-parents*:
assumes *b*: *b* ∈ *sparsegrid′ dm* **and** *p′*: *p′* ∈ *parents d b p*
shows *p′* ∈ *sparsegrid′ dm*
using *assms parents-def sparsegrid′I* **by** *auto*

lemma *down′-β*: ⟦ *d* < *length b* ; *l* + *level b* = *lm* ; *b* ∈ *sparsegrid′ dm* ; *p* ∈ *sparsegrid′ dm* ⟧ ⟹
down′ d l b fl fr α p = (*if p* ∈ *lgrid b* {*d*} *lm*
then
(*fl* + (*fr* − *fl*) / *2* ∗ (*real-of-int* (*ix p d*) / *2*^(*lv p d* − *lv b d*) − *real-of-int* (*ix b d*) + *1*)) / *2* ^ (*lv p d* + *1*) +
(∑ *p′* ∈ *parents d b p*. (*α p′*) ∗ *l2-φ* (*p* ! *d*) (*p′* ! *d*))
else α p)
proof (*induct l arbitrary*: *b α fl fr p*)
case (*Suc l*)

let *?l* = *child b left d* **and** *?r* = *child b right d*
let *?result* = ((*fl* + *fr*) / *4* + (*1* / *3*) ∗ (*α b*)) / *2* ^ (*lv b d*)
let *?fm* = (*fl* + *fr*) / *2* + (*α b*)
let *?down-l* = *down′ d l* (*child b left d*) *fl ?fm* (*α*(*b* := *?result*))

have *length b* = *dm* **using** ‹*b* ∈ *sparsegrid′ dm*›
unfolding *sparsegrid′-def start-def* **by** *auto*
hence *d* < *dm* **using** ‹*d* < *length b*› **by** *auto*

have !!*dir*. *d* < *length* (*child b dir d*) **using** ‹*d* < *length b*› **by** *auto*

```
  have !!dir. l + level (child b dir d) = lm
    using ‹d < length b› and ‹Suc l + level b = lm› and child-level by auto
  have !!dir. (child b dir d) ∈ sparsegrid' dm
    using ‹b ∈ sparsegrid' dm› and ‹d < dm› and sparsegrid'-def by auto
  note hyps = Suc.hyps[OF ‹!! dir. d < length (child b dir d)›
    ‹!!dir. l + level (child b dir d) = lm›
    ‹!!dir. (child b dir d) ∈ sparsegrid' dm›]

  show ?case
  proof (cases p ∈ lgrid b {d} lm)
    case False
    moreover hence p ≠ b and p ∉ lgrid ?l {d} lm
      and p ∉ lgrid ?r {d} lm unfolding lgrid-def
      unfolding grid-partition[where p=b] using ‹Suc l + level b = lm› by auto
    ultimately show ?thesis
      unfolding down'.simps Let-def fun-upd-def hyps[OF ‹p ∈ sparsegrid' dm›]
      by auto
  next
    case True hence level p < lm and p ∈ grid b {d} unfolding lgrid-def by
auto
    let ?lb = lv b d   and ?ib = real-of-int (ix b d)
    let ?lp   = lv p d  and ?ip   = real-of-int (ix p d)
    show ?thesis
    proof (cases ∃ dir. p ∈ grid (child b dir d){d})
      case True
      obtain dir where p-grid: p ∈ grid (child b dir d) {d} using True by auto
       hence p ∈ lgrid (child b dir d) {d} lm using ‹level p < lm› unfolding
lgrid-def by auto
     have lv b d < lv p d using child-lv[OF ‹d < length b›] and grid-single-level[OF
p-grid ‹d < length (child b dir d)›] by auto

      let ?ch = child b dir d
      let ?ich = child b (inv dir) d

      show ?thesis
      proof (cases dir)
       case right
       hence p ∈ lgrid ?r {d} lm and p ∈ grid ?r {d}
        using ‹p ∈ grid ?ch {d}› and ‹level p < lm› unfolding lgrid-def by auto

       { fix p' fix fl fr x assume p': p' ∈ parents d (child b right d) p
        hence p' ∈ grid (child b right d) {d} unfolding parents-def by simp
        hence p' ∉ lgrid (child b left d) {d} lm and p' ≠ b
          unfolding lgrid-def
          using grid-disjunct[OF ‹d < length b›] grid-not-child by auto

        from hyps[OF sparsegrid'-parents[OF ‹child b right d ∈ sparsegrid' dm›
                  p']] this
        have down' d l (child b left d) fl fr (α(b := x)) p' = α p' by auto }
```

```
      thus ?thesis
        unfolding down'.simps Let-def hyps[OF ‹p ∈ sparsegrid' dm›]
          parent-sum[OF ‹p ∈ grid ?r {d}› ‹d < length b›]
          l2-child[OF ‹d < length b› ‹p ∈ grid ?r {d}›]
        using child-ix child-lv ‹d < length b› level-shift[OF ‹lv b d < lv p d›]
            sgn.simps ‹p ∈ lgrid b {d} lm› ‹p ∈ lgrid ?r {d} lm›
        by (auto simp add: algebra-simps diff-divide-distrib add-divide-distrib)
    next
      case left
      hence p ∈ lgrid ?l {d} lm and p ∈ grid ?l {d}
        using ‹p ∈ grid ?ch {d}› and ‹level p < lm› unfolding lgrid-def by auto
      hence ¬ p ∈ lgrid ?r {d} lm
        using grid-disjunct[OF ‹d < length b›] unfolding lgrid-def by auto
      { fix p' assume p': p' ∈ parents d (child b left d) p
        hence p' ∈ grid (child b left d) {d} unfolding parents-def by simp
        hence p' ≠ b using grid-not-child[OF ‹d < length b›] by auto }
      thus ?thesis
        unfolding down'.simps Let-def hyps[OF ‹p ∈ sparsegrid' dm›]
            parent-sum[OF ‹p ∈ grid ?l {d}› ‹d < length b›]
            l2-child[OF ‹d < length b› ‹p ∈ grid ?l {d}›] sgn.simps
            if-P[OF ‹p ∈ lgrid b {d} lm›] if-P[OF ‹p ∈ lgrid ?l {d} lm›]
            if-not-P[OF ‹p ∉ lgrid ?r {d} lm›]
        using child-ix child-lv ‹d < length b› level-shift[OF ‹lv b d < lv p d›]
        by (auto simp add: algebra-simps diff-divide-distrib add-divide-distrib)
    qed
  next
    case False hence not-child: !! dir. ¬ p ∈ grid (child b dir d) {d} by auto
    hence p = b using grid-onedim-split[where ds={} and d=d and b=b] ‹p
∈ grid b {d}› unfolding grid-empty-ds[where b=b] by auto
    from not-child have lnot-child: !! dir. ¬ p ∈ lgrid (child b dir d) {d} lm
unfolding lgrid-def by auto
    have result: ((fl + fr) / 4 + 1 / 3 * α b) / 2 ^ lv b d = (fl + (fr − fl) /
2) / 2 ^ (lv b d + 1) + α b * l2-φ (b ! d) (b ! d)
    by (auto simp: l2-same diff-divide-distrib add-divide-distrib times-divide-eq-left[symmetric]
algebra-simps)
    show ?thesis
      unfolding down'.simps Let-def fun-upd-def hyps[OF ‹p ∈ sparsegrid' dm›]
if-P[OF ‹p ∈ lgrid b {d} lm›] if-not-P[OF lnot-child] if-P[OF ‹p = b›]
      unfolding ‹p = b› parents-single unfolding result by auto
  qed
 qed
next
 case 0
 have p ∉ lgrid b {d} lm
 proof (rule ccontr)
  assume ¬ p ∉ lgrid b {d} lm
  hence p ∈ grid b {d} and level p < lm unfolding lgrid-def by auto
  moreover from grid-level[OF ‹p ∈ grid b {d}›] and ‹0 + level b = lm› have
lm ≤ level p by auto
```

```
    ultimately show False by auto
  qed
  thus ?case unfolding down'.simps by auto
qed

lemma down: assumes d < dm and p: p ∈ sparsegrid dm lm
  shows (down dm lm d α) p = (∑ p' ∈ parents d (base {d} p) p. (α p') * l2-φ
(p ! d) (p' ! d))
proof −
  let ?F d l p = down' d l p 0 0
  let ?S x p p' = if p' ∈ parents d (base {d} p) p then x * l2-φ (p ! d) (p' ! d)
else 0

  { fix p α assume p ∈ sparsegrid dm lm
    from le-less-trans[OF grid-level sparsegridE(2)[OF this]]
    have parents d (base {d} p) p ⊆ lgrid (base {d} p) {d} lm
      unfolding lgrid-def parents-def by auto
    hence (∑ p'∈lgrid (base {d} p) {d} lm. ?S (α p') p p') =
      (∑ p'∈parents d (base {d} p) p. α p' * l2-φ (p ! d) (p' ! d))
      using lgrid-finite by (intro sum.mono-neutral-cong-right) auto
  } note sum-eq = this

  { fix l p b α
    assume b: b ∈ lgrid (start dm) ({0..<dm} − {d}) lm and l + level b = lm
      and p ∈ sparsegrid dm lm
    hence b-spg: b ∈ sparsegrid' dm and p-spg: p ∈ sparsegrid' dm and
      d < length b and level p < lm
      using sparsegrid'-start sparsegrid-subset ‹d < dm› by auto
    have ?F d l b α p = (if b = base {d} p then ∑ p'∈lgrid b {d} lm. ?S (α p')
p p' else α p)
    proof (cases b = base {d} p)
      case True
      have p ∈ lgrid (base {d} p) {d} lm
        using baseE(2)[OF p-spg] and ‹level p < lm›
        unfolding lgrid-def by auto
      thus ?thesis unfolding if-P[OF True]
        unfolding True sum-eq[OF ‹p ∈ sparsegrid dm lm›]
        unfolding down'-β[OF ‹d < length b› ‹l + level b = lm› b-spg p-spg,
          unfolded True] by auto
    next
      case False
      have p ∉ lgrid b {d} lm
      proof (rule ccontr)
        assume ¬ ?thesis hence p ∈ grid b {d} by auto
        from b this have b = base {d} p using baseI by auto
        thus False using False by simp
      qed
      thus ?thesis
        unfolding if-not-P[OF False]
```

unfolding *down′-β*[*OF* ‹*d* < *length b*› ‹*l* + *level b* = *lm*› *b-spg p-spg*]
by *auto*
qed }
from *lift*[*OF* ‹*d* < *dm*› ‹*p* ∈ *sparsegrid dm lm*›, **where** *F* = *?F* **and** *S* = *?S*, *OF this*]
show *?thesis*
unfolding *down-def*
unfolding *sum-eq*[*OF p*] **by** *simp*
qed

end

7 UpDown

theory *Up-Down*
imports *Up Down*
begin

lemma *updown′*: ⟦ *d* ≤ *dm*; *p* ∈ *sparsegrid dm lm* ⟧
⟹ (*updown′ dm lm d* α) *p* = (∑ *p′* ∈ *lgrid* (*base* {*0* ..< *d*} *p*) {*0* ..< *d*} *lm*. α *p′* ∗ (∏ *d′* ∈ {*0* ..< *d*}. *l2-φ* (*p′* ! *d′*) (*p* ! *d′*)))
(**is** ⟦ - ; - ⟧ ⟹ - = (∑ *p′* ∈ *?subgrid d p*. α *p′* ∗ *?prod d p′ p*))
proof (*induct d arbitrary*: α *p*)
case *0* **hence** *?subgrid 0 p* = {*p*} **using** *base-empty* **unfolding** *lgrid-def* **and** *sparsegrid-def sparsegrid′-def* **by** *auto*
thus *?case* **unfolding** *updown′.simps* **by** *auto*
next
case (*Suc d*)
let *?leafs p* = (*lgrid p* {*d*} *lm*) − {*p*}
let *?parents* = *parents d* (*base* {*d*} *p*) *p*
let *?b* = *base* {*0..<d*} *p*
have *d* < *dm* **using** ‹*Suc d* ≤ *dm*› **by** *auto*

have *p-spg*: *p* ∈ *grid* (*start dm*) {*0..<dm*} **and** *p-spg′*: *p* ∈ *sparsegrid′ dm* **and** *level p* < *lm* **using** ‹*p* ∈ *sparsegrid dm lm*›
unfolding *sparsegrid-def* **and** *sparsegrid′-def* **and** *lgrid-def* **by** *auto*
have *p′-in-spg*: !! *p′*. *p′* ∈ *?subgrid d p* ⟹ *p′* ∈ *sparsegrid dm lm*
using *base-grid*[*OF p-spg′*] **unfolding** *sparsegrid′-def sparsegrid-def lgrid-def* **by** *auto*

from *baseE*[*OF p-spg′*, **where** *ds*={*0..<d*}]
have *?b* ∈ *grid* (*start dm*) {*d..<dm*} **and** *p-bgrid*: *p* ∈ *grid ?b* {*0..<d*} **by** *auto*
hence *d* < *length ?b* **using** ‹*Suc d* ≤ *dm*› **by** *auto*
have *p* ! *d* = *?b* ! *d* **using** *base-out*[*OF* - - *p-spg′*] ‹*Suc d* ≤ *dm*› **by** *auto*

have *length p* = *dm* **using** ‹*p* ∈ *sparsegrid dm lm*› **unfolding** *sparsegrid-def lgrid-def* **by** *auto*
hence *d* < *length p* **using** ‹*d* < *dm*› **by** *auto*

```
  have updown' dm lm d (up dm lm d α) p =
    (∑ p' ∈ ?subgrid d p. (up dm lm d α) p' * (?prod d p' p))
    using Suc by auto
  also have ... = (∑ p' ∈ ?subgrid d p. (∑ p'' ∈ ?leafs p'. α p'' * ?prod (Suc
d) p'' p))
  proof (intro sum.cong refl)
    fix p' assume p' ∈ ?subgrid d p
    hence d < length p' unfolding lgrid-def using base-length[OF p-spg'] ‹Suc d
≤ dm› by auto

    have up dm lm d α p' * ?prod d p' p =
      (∑ p'' ∈ ?leafs p'. α p'' * l2-φ (p'' ! d) (p' ! d)) * ?prod d p' p
      using ‹p' ∈ ?subgrid d p› up ‹Suc d ≤ dm› p'-in-spg by auto
    also have ... = (∑ p'' ∈ ?leafs p'. α p'' * l2-φ (p'' ! d) (p' ! d) * ?prod d p'
p)
      using sum-distrib-right by auto
    also have ... = (∑ p'' ∈ ?leafs p'. α p'' * ?prod (Suc d) p'' p)
    proof (intro sum.cong refl)
      fix p'' assume p'' ∈ ?leafs p'
      have ?prod d p' p = ?prod d p'' p
      proof (intro prod.cong refl)
        fix d' assume d' ∈ {0..<d}
        hence d-lt-p: d' < length p' and d'-not-d: d' ∉ {d} using ‹d < length p'›
by auto
          hence p' ! d' = p'' ! d' using ‹p'' ∈ ?leafs p'› grid-invariant[OF d-lt-p
d'-not-d] unfolding lgrid-def by auto
        thus l2-φ (p'!d') (p!d') = l2-φ (p''!d') (p!d') by auto
      qed
      moreover have p' ! d = p ! d
        using ‹p' ∈ ?subgrid d p› and grid-invariant[OF ‹d < length ?b›, where
p=p' and ds={0..<d}] unfolding lgrid-def ‹p ! d = ?b ! d› by auto
      ultimately have l2-φ (p'' ! d) (p' ! d) * ?prod d p' p =
        l2-φ (p'' ! d) (p ! d) * ?prod d p'' p by auto
      also have ... = ?prod (Suc d) p'' p
      proof −
        have insert d {0..<d} = {0..<Suc d} by auto
        moreover from prod.insert
        have prod (λ d'. l2-φ (p'' ! d') (p ! d')) (insert d {0..<d}) =
        (λ d'. l2-φ (p'' ! d') (p ! d')) d * prod (λ d'. l2-φ (p'' ! d') (p ! d')) {0..<d}
          by auto
        ultimately show ?thesis by auto
      qed
      finally show α p'' * l2-φ (p'' ! d) (p' ! d) * ?prod d p' p = α p'' * ?prod
(Suc d) p'' p by auto
    qed
    finally show (up dm lm d α) p' * (?prod d p' p) = (∑ p'' ∈ ?leafs p'. α p''
* ?prod (Suc d) p'' p) by auto
  qed
  also have ... = (∑ (p', p'') ∈ Sigma (?subgrid d p) (λp'. (?leafs p')). (α p'') *
```

(*?prod* (*Suc d*) *p''* *p*))
by (*rule sum.Sigma*, *auto simp add*: *lgrid-finite*)
also have ... = (∑ *p'''* ∈ (⋃ *p'* ∈ *?subgrid d p*. (⋃ *p''* ∈ *?leafs p'*. { (*p'*, *p''*) })).
(((λ *p''*. *α p''* ∗ *?prod* (*Suc d*) *p''* *p*) *o snd*) *p'''*)) **unfolding** *Sigma-def* **by** (*rule sum.cong*[*OF refl*], *auto*)
also have ... = (∑ *p''* ∈ *snd* ‘ (⋃ *p'* ∈ *?subgrid d p*. (⋃ *p''* ∈ *?leafs p'*. { (*p'*, *p''*) })).
α p'' ∗ (*?prod* (*Suc d*) *p''* *p*)) **unfolding** *lgrid-def*
by (*rule sum.reindex*[*symmetric*],
rule subset-inj-on[*OF grid-grid-inj-on*[*OF ivl-disj-int*(*15*)[**where** *l=0* **and** *m=d* **and** *u=d*], **where** *b=?b*]])
auto
also have ... = (∑ *p''* ∈ (⋃ *p'* ∈ *?subgrid d p*. (⋃ *p''* ∈ *?leafs p'*. *snd* ‘ { (*p'*, *p''*) })).
α p'' ∗ (*?prod* (*Suc d*) *p''* *p*)) **by** (*auto simp only*: *image-UN*)
also have ... = (∑ *p''* ∈ (⋃ *p'* ∈ *?subgrid d p*. *?leafs p'*). *α p''* ∗ (*?prod* (*Suc d*) *p''* *p*)) **by** *auto*
finally have *up-part*: *updown' dm lm d* (*up dm lm d α*) *p* = (∑ *p''* ∈ (⋃ *p'* ∈ *?subgrid d p*. *?leafs p'*). *α p''* ∗ (*?prod* (*Suc d*) *p''* *p*)) **.**

have *down dm lm d* (*updown' dm lm d α*) *p* = (∑ *p'* ∈ *?parents*. (*updown' dm lm d α p'*) ∗ *l2-φ* (*p* ! *d*) (*p'* ! *d*))
using ‹*Suc d* ≤ *dm*› **and** *down* **and** ‹*p* ∈ *sparsegrid dm lm*› **by** *auto*
also have ... = (∑ *p'* ∈ *?parents*. ∑ *p''* ∈ *?subgrid d p'*. *α p''* ∗ *?prod* (*Suc d*) *p''* *p*)
proof (*rule sum.cong*[*OF refl*])
fix *p'* **let** *?b'* = *base* {*d*} *p*
assume *p'* ∈ *?parents*
hence *p-lgrid*: *p'* ∈ *lgrid ?b'* {*d*} (*level p* + *1*) **using** *parents-subset-lgrid* **by** *auto*
hence *p'* ∈ *sparsegrid dm lm* **and** *p'-spg'*: *p'* ∈ *sparsegrid' dm* **using** ‹*level p* < *lm*› *base-grid*[*OF p-spg'*] **unfolding** *sparsegrid-def lgrid-def sparsegrid'-def* **by** *auto*
hence *length p'* = *dm* **unfolding** *sparsegrid-def lgrid-def* **by** *auto*
hence *d* < *length p'* **using** ‹*Suc d* ≤ *dm*› **by** *auto*

from *p-lgrid* **have** *p'-grid*: *p'* ∈ *grid ?b'* {*d*} **unfolding** *lgrid-def* **by** *auto*

have (*updown' dm lm d α p'*) ∗ *l2-φ* (*p* ! *d*) (*p'* ! *d*) = (∑ *p''* ∈ *?subgrid d p'*. *α p''* ∗ *?prod d p'' p'*) ∗ *l2-φ* (*p* ! *d*) (*p'* ! *d*)
using ‹*p'* ∈ *sparsegrid dm lm*› *Suc* **by** *auto*
also have ... = (∑ *p''* ∈ *?subgrid d p'*. *α p''* ∗ *?prod d p'' p'* ∗ *l2-φ* (*p* ! *d*) (*p'* ! *d*))
using *sum-distrib-right* **by** *auto*
also have ... = (∑ *p''* ∈ *?subgrid d p'*. *α p''* ∗ *?prod* (*Suc d*) *p''* *p*)
proof (*rule sum.cong*[*OF refl*])
fix *p''* **assume** *p''* ∈ *?subgrid d p'*

have *?prod d p'' p' = ?prod d p'' p*
proof (*rule prod.cong*, *rule refl*)
fix *d'* **assume** *d' ∈ {0..<d}*
hence *d' < dm* **and** *d' ∉ {d}* **using** ‹*Suc d ≤ dm*› **by** *auto*
from *grid-base-out*[*OF this p-spg' p'-grid*]
show *l2-φ (p''!d') (p'!d') = l2-φ (p''!d') (p!d')* **by** *auto*
qed
moreover
have *l2-φ (p ! d) (p' ! d) = l2-φ (p'' ! d) (p ! d)*
proof −
have *d < dm* **and** *d ∉ {0..<d}* **using** ‹*Suc d ≤ dm*› *base-length p'-spg'* **by** *auto*
from *grid-base-out*[*OF this p'-spg'*] ‹*p'' ∈ ?subgrid d p'*›[*unfolded lgrid-def*]
show *?thesis* **using** *l2-commutative* **by** *auto*
qed
moreover have *?prod d p'' p * l2-φ (p'' ! d) (p ! d) = ?prod (Suc d) p'' p*
proof −
have *insert d {0..<d} = {0..<Suc d}* **by** *auto*
moreover from *prod.insert*
have *(λ d'. l2-φ (p'' ! d') (p ! d')) d * prod (λ d'. l2-φ (p'' ! d') (p ! d')) {0..<d} =*
prod (λ d'. l2-φ (p'' ! d') (p ! d')) (insert d {0..<d})
by *auto*
hence *(prod (λ d'. l2-φ (p'' ! d') (p ! d')) {0..<d}) * (λ d'. l2-φ (p'' ! d') (p ! d')) d =*
prod (λ d'. l2-φ (p'' ! d') (p ! d')) (insert d {0..<d})
by *auto*
ultimately show *?thesis* **by** *auto*
qed
ultimately show *α p'' * ?prod d p'' p' * l2-φ (p ! d) (p' ! d) = α p'' * ?prod (Suc d) p'' p* **by** *auto*
qed
finally show *(updown' dm lm d α p') * l2-φ (p ! d) (p' ! d) = (∑ p'' ∈ ?subgrid d p'. α p'' * ?prod (Suc d) p'' p)* **by** *auto*
qed
also have *... = (∑ (p', p'') ∈ (Sigma ?parents (?subgrid d)). α p'' * ?prod (Suc d) p'' p)*
by (*rule sum.Sigma*, *auto simp add*: *parents-finite lgrid-finite*)
also have *... = (∑ p''' ∈ (⋃ p' ∈ ?parents. (⋃ p'' ∈ ?subgrid d p'. { (p', p'') })).*
*(((λ p''. α p'' * ?prod (Suc d) p'' p) o snd) p'''))* **unfolding** *Sigma-def* **by** (*rule sum.cong*[*OF refl*], *auto*)
also have *... = (∑ p'' ∈ snd ‘ (⋃ p' ∈ ?parents. (⋃ p'' ∈ ?subgrid d p'. { (p', p'') })). α p'' * (?prod (Suc d) p'' p))*
proof (*rule sum.reindex*[*symmetric*], *rule inj-onI*)
fix *x y*
assume *x ∈ (⋃p'∈parents d (base {d} p) p. ⋃p''∈lgrid (base {0..<d} p') {0..<d} lm. {(p', p'')})*
hence *x-snd*: *snd x ∈ grid (base {0..<d} (fst x)) {0..<d}* **and** *fst x ∈ grid*

(*base* {*d*} *p*) {*d*} **and** *p* ∈ *grid* (*fst x*) {*d*}
unfolding *parents-def lgrid-def* **by** *auto*
hence *x-spg*: *fst x* ∈ *sparsegrid' dm* **using** *base-grid*[*OF p-spg'*] **by** *auto*

assume *y* ∈ (⋃*p'*∈*parents d* (*base* {*d*} *p*) *p*. ⋃*p''*∈*lgrid* (*base* {*0..<d*} *p'*) {*0..<d*} *lm*. {(*p'*, *p''*)})
hence *y-snd*: *snd y* ∈ *grid* (*base* {*0..<d*} (*fst y*)) {*0..<d*} **and** *fst y* ∈ *grid* (*base* {*d*} *p*) {*d*} **and** *p* ∈ *grid* (*fst y*) {*d*}
unfolding *parents-def lgrid-def* **by** *auto*
hence *y-spg*: *fst y* ∈ *sparsegrid' dm* **using** *base-grid*[*OF p-spg'*] **by** *auto*
hence *length* (*fst y*) = *dm* **unfolding** *sparsegrid'-def* **by** *auto*

assume *snd x* = *snd y*
have *fst x* = *fst y*
proof (*rule nth-equalityI*)
show *l-eq*: *length* (*fst x*) = *length* (*fst y*) **using** *grid-length*[*OF* ‹*p* ∈ *grid* (*fst y*) {*d*}›] *grid-length*[*OF* ‹*p* ∈ *grid* (*fst x*) {*d*}›]
by *auto*
show *fst x* ! *i* = *fst y* ! *i* **if** *i* < *length* (*fst x*) **for** *i*
proof −
have *i* < *length* (*fst y*) **and** *i* < *dm* **using** *that l-eq* **and** ‹*length* (*fst y*) = *dm*› **by** *auto*
show *fst x* ! *i* = *fst y* ! *i*
proof (*cases i* = *d*)
case *False* **hence** *i* ∉ {*d*} **by** *auto*
with *grid-invariant*[*OF* ‹*i* < *length* (*fst x*)› *this* ‹*p* ∈ *grid* (*fst x*) {*d*}›] *grid-invariant*[*OF* ‹*i* < *length* (*fst y*)› *this* ‹*p* ∈ *grid* (*fst y*) {*d*}›]
show *?thesis* **by** *auto*
next
case *True* **with** *grid-base-out*[*OF* ‹*i* < *dm*› *- y-spg y-snd*] *grid-base-out*[*OF* ‹*i* < *dm*› *- x-spg x-snd*]
show *?thesis* **using** ‹*snd x* = *snd y*› **by** *auto*
qed
qed
qed
show *x* = *y* **using** *prod-eqI*[*OF* ‹*fst x* = *fst y*› ‹*snd x* = *snd y*›] **.**
qed
also have ... = (∑ *p''* ∈ (⋃ *p'* ∈ *?parents*. (⋃ *p''* ∈ *?subgrid d p'*. *snd* ‘ { (*p'*, *p''*) })).
α p'' ∗ (*?prod* (*Suc d*) *p''* *p*)) **by** (*auto simp only*: *image-UN*)
also have ... = (∑ *p''* ∈ (⋃ *p'* ∈ *?parents*. *?subgrid d p'*). *α p''* ∗ *?prod* (*Suc d*) *p''* *p*) **by** *auto*
finally have *down-part*: *down dm lm d* (*updown' dm lm d α*) *p* =
(∑ *p''* ∈ (⋃ *p'* ∈ *?parents*. *?subgrid d p'*). *α p''* ∗ *?prod* (*Suc d*) *p''* *p*) **.**

have *updown' dm lm* (*Suc d*) *α p* =
(∑ *p''* ∈ (⋃ *p'* ∈ *?subgrid d p*. *?leafs p'*). *α p''* ∗ *?prod* (*Suc d*) *p''* *p*) +
(∑ *p''* ∈ (⋃ *p'* ∈ *?parents*. *?subgrid d p'*). *α p''* ∗ *?prod* (*Suc d*) *p''* *p*)
unfolding *sum-vector-def updown'.simps down-part* **and** *up-part* **..**

also have ... = ($\sum$ p'' ∈ ($\bigcup$ p' ∈ *?subgrid d p. ?leafs p'*) ∪ ($\bigcup$ p' ∈ *?parents. ?subgrid d p'*). α p'' * *?prod* (*Suc d*) p'' *p*)
proof (*rule sum.union-disjoint*[*symmetric*], *simp add*: *lgrid-finite*, *simp add*: *lgrid-finite parents-finite*,
rule iffD2[*OF disjoint-iff-not-equal*], *rule ballI*, *rule ballI*)
fix *x y*
assume *x* ∈ ($\bigcup$ p' ∈ *?subgrid d p. ?leafs p'*)
then obtain *px* **where** *px* ∈ *grid* (*base* {*0..<d*} *p*) {*0..<d*} **and** *x* ∈ *grid px* {*d*} **and** *x* ≠ *px* **unfolding** *lgrid-def* **by** *auto*
with *grid-base-out*[*OF - - p-spg' this*(*1*)] ‹*Suc d* ≤ *dm*› *base-length*[*OF p-spg'*] *grid-level-d*
have *lv px d* < *lv x d* **and** *px* ! *d* = *p* ! *d* **by** *auto*
hence *lv p d* < *lv x d* **unfolding** *lv-def* **by** *auto*
moreover
assume *y* ∈ ($\bigcup$ p' ∈ *?parents. ?subgrid d p'*)
then obtain *py* **where** *y-grid*: *y* ∈ *grid* (*base* {*0..<d*} *py*) {*0..<d*} **and** *py* ∈ *?parents* **unfolding** *lgrid-def* **by** *auto*
hence *py* ∈ *grid* (*base* {*d*} *p*) {*d*} **and** *p* ∈ *grid py* {*d*} **unfolding** *parents-def* **by** *auto*
hence *py-spg*: *py* ∈ *sparsegrid' dm* **using** *base-grid*[*OF p-spg'*] **by** *auto*
have *y* ! *d* = *py* ! *d* **using** *grid-base-out*[*OF - - py-spg y-grid*] ‹*Suc d* ≤ *dm*› **by** *auto*
hence *lv y d* ≤ *lv p d* **using** *grid-single-level*[*OF* ‹*p* ∈ *grid py* {*d*}›] ‹*Suc d* ≤ *dm*› **and** *sparsegrid'-length*[*OF py-spg*] **unfolding** *lv-def* **by** *auto*
ultimately
show *x* ≠ *y* **by** *auto*
qed
also have ... = ($\sum$ p' ∈ *?subgrid* (*Suc d*) *p*. α p' * *?prod* (*Suc d*) p' *p*) (**is** ($\sum$ *x* ∈ *?in*. *?F x*) = ($\sum$ *x* ∈ *?out*. *?F x*))
proof (*rule sum.mono-neutral-left*, *simp add*: *lgrid-finite*)
show *?in* ⊆ *?out* (**is** *?children* ∪ *?siblings* ⊆ -)
proof (*rule subsetI*)
fix *x* **assume** *x* ∈ *?in*
show *x* ∈ *?out*
proof (*cases x* ∈ *?children*)
case *False* **hence** *x* ∈ *?siblings* **using** ‹*x* ∈ *?in*› **by** *auto*
then obtain *px* **where** *px* ∈ *parents d* (*base* {*d*} *p*) *p* **and** *x* ∈ *lgrid* (*base* {*0..<d*} *px*) {*0..<d*} *lm* **by** *auto*
hence *level x* < *lm* **and** *px* ∈ *grid* (*base* {*d*} *p*) {*d*} **and** *x* ∈ *grid* (*base* {*0..<d*} *px*) {*0..<d*} **and** {*d*} ∪ {*0..<d*} = {*0..<Suc d*} **unfolding** *lgrid-def parents-def* **by** *auto*
with *grid-base-union*[*OF p-spg' this*(*2*) *this*(*3*)] **show** *?thesis* **unfolding** *lgrid-def* **by** *auto*
next
have *d-eq*: {*0..<Suc d*} ∪ {*d*} = {*0..<Suc d*} **by** *auto*
case *True*
then obtain *px* **where** *px* ∈ *?subgrid d p* **and** *x* ∈ *lgrid px* {*d*} *lm* **and** *x* ≠ *px* **by** *auto*
hence *px* ∈ *grid* (*base* {*0..<d*} *p*) {*0..<d*} **and** *x* ∈ *grid px* {*d*} **and** *level*

```
x < lm and {d} ∪ {0..<d} = {0..<Suc d} unfolding lgrid-def by auto
      from grid-base-dim-add[OF - p-spg' this(1)]
      have px ∈ grid (base {0..<Suc d} p) {0..<Suc d} by auto
      from grid-transitive[OF ‹x ∈ grid px {d}› this]
      show ?thesis unfolding lgrid-def using ‹level x < lm› d-eq by auto
     qed
    qed

    show ∀ x ∈ ?out − ?in. ?F x = 0
    proof
     fix x assume x ∈ ?out − ?in

     hence x ∈ ?out and up-ps': !! p'. p' ∈ ?subgrid d p ⟹ x ∉ lgrid p' {d} lm
− {p'}
      and down-ps': !! p'. p' ∈ ?parents ⟹ x ∉ ?subgrid d p' by auto
     hence x-eq: x ∈ grid (base {0..<Suc d} p) {0..<Suc d} and level x < lm
unfolding lgrid-def by auto
     hence up-ps: !! p'. p' ∈ ?subgrid d p ⟹ x ∉ grid p' {d} − {p'} and
       down-ps: !! p'. p' ∈ ?parents ⟹ x ∉ grid (base {0..<d} p') {0..<d}
       using up-ps' down-ps' unfolding lgrid-def by auto

     have ds-eq: {0..<Suc d} = {0..<d} ∪ {d} by auto
    have x ∉ grid (base {0..<d} p) {0..<Suc d} − grid (base {0..<d} p) {0..<d}
     proof
      assume x ∈ grid (base {0..<d} p) {0..<Suc d} − grid (base {0..<d} p)
{0..<d}
      hence x ∈ grid (base {0..<d} p) ({d} ∪ {0..<d}) and x-ngrid: x ∉ grid
(base {0..<d} p) {0..<d} using ds-eq by auto
       from grid-split[OF this(1)] obtain px where px-grid: px ∈ grid (base
{0..<d} p) {0..<d} and x ∈ grid px {d} by auto
      from grid-level[OF this(2)] ‹level x < lm› have level px < lm by auto
      hence px ∈ ?subgrid d p using px-grid unfolding lgrid-def by auto
      hence x ∉ grid px {d} − {px} using up-ps by auto
     moreover have x ≠ px proof (rule ccontr) assume ¬ x ≠ px with px-grid
and x-ngrid show False by auto qed
      ultimately show False using ‹x ∈ grid px {d}› by auto
     qed
     moreover have p ∈ ?parents unfolding parents-def using baseE(2)[OF
p-spg'] by auto
     hence x ∉ grid (base {0..<d} p) {0..<d} by (rule down-ps)
     ultimately have x-ngrid: x ∉ grid (base {0..<d} p) {0..<Suc d} by auto

     have x-spg: x ∈ sparsegrid' dm using base-grid[OF p-spg'] x-eq by auto
     hence length x = dm using grid-length by auto

     let ?bx = base {0..<d} x and ?bp = base {0..<d} p and ?bx1 = base {d}
x and ?bp1 = base {d} p and ?px = p[d := x ! d]

     have x-nochild-p: ?bx ∉ grid ?bp {d}
```

proof (*rule ccontr*)
assume ¬ *base* {*0*..<*d*} *x* ∉ *grid* (*base* {*0*..<*d*} *p*) {*d*}
hence *base* {*0*..<*d*} *x* ∈ *grid* (*base* {*0*..<*d*} *p*) {*d*} **by** *auto*
from *grid-transitive*[*OF baseE*(*2*)[*OF x-spg*] *this*]
have *x* ∈ *grid* (*base* {*0*..<*d*} *p*) {*0*..<*Suc d*} **using** *ds-eq* **by** *auto*
thus *False* **using** *x-ngrid* **by** *auto*
qed

have *d* < *length ?bx* **and** *d* < *length ?bp* **and** *d* < *length ?bx1* **and** *d* < *length ?bp1* **using** *base-length*[*OF x-spg*] *base-length*[*OF p-spg'*] **and** ‹*d* < *dm*› **by** *auto*
have *p-nochild-x*: *?bp* ∉ *grid ?bx* {*d*} (**is** *?assm*)
proof (*rule ccontr*)
have *ds*: {*0*..<*d*} ∪ {*0*..<*Suc d*} = {*d*} ∪ {*0*..<*d*} **by** *auto*
have *d-sub*: {*d*} ⊆ {*0*..<*Suc d*} **by** *auto*
assume ¬ *?assm* **hence** *b-in-bx*: *base* {*0*..<*d*} *p* ∈ *grid ?bx* {*d*} **by** *auto*

have *d* ∉ {*0*..<*d*} **and** *d* ∈ {*d*} **by** *auto*
from *grid-replace-dim*[*OF* ‹*d* < *length ?bx*› ‹*d* < *length p*› *grid.Start*[**where** *b*=*p* **and** *ds*={*d*}] *b-in-bx*]
have *p* ∈ *grid ?px* {*d*} **unfolding** *base-out*[*OF* ‹*d* < *dm*› ‹*d* ∉ {*0*..<*d*}› *x-spg*] *base-out*[*OF* ‹*d* < *dm*› ‹*d* ∉ {*0*..<*d*}› *p-spg'*] *list-update-id* **.**
moreover
from *grid-replace-dim*[*OF* ‹*d* < *length ?bx1*› ‹*d* < *length ?bp1*› *baseE*(*2*)[*OF p-spg'*, **where** *ds*={*d*}] *baseE*(*2*)[*OF x-spg*, **where** *ds*={*d*}]]
have *?px* ∈ *grid ?bp1* {*d*} **unfolding** *base-in*[*OF* ‹*d* < *dm*› ‹*d* ∈ {*d*}› *x-spg*] **unfolding** *base-in*[*OF* ‹*d* < *dm*› ‹*d* ∈ {*d*}› *p-spg'*, *symmetric*] *list-update-id* **.**
ultimately
have *x* ∉ *grid* (*base* {*0*..<*d*} *?px*) {*0*..<*d*} **using** *down-ps*[*unfolded parents-def*, **where** *p'*=*?px*] **by** (*auto simp only*:)
moreover
have *base* {*0*..<*d*} *?px* = *?bx*
proof (*rule nth-equalityI*)
from ‹*?px* ∈ *grid ?bp1* {*d*}› **have** *px-spg*: *?px* ∈ *sparsegrid' dm* **using** *base-grid*[*OF p-spg'*] **by** *auto*
from *base-length*[*OF this*] *base-length*[*OF x-spg*] **show** *l-eq*: *length* (*base* {*0*..<*d*} *?px*) = *length ?bx* **by** *auto*
show *base* {*0*..<*d*} *?px* ! *i* = *?bx* ! *i* **if** *i* < *length* (*base* {*0*..<*d*} *?px*) **for** *i*
proof −
have *i* < *length ?bx* **and** *i* < *dm* **using** *that l-eq* **and** *base-length*[*OF px-spg*] **by** *auto*
show *base* {*0*..<*d*} *?px* ! *i* = *?bx* ! *i*
proof (*cases i* < *d*)
case *True* **hence** *i* ∈ {*0*..<*d*} **by** *auto*
from *base-in*[*OF* ‹*i* < *dm*› *this*] **show** *?thesis* **using** *px-spg x-spg* **by** *auto*
next
case *False* **hence** *i* ∉ {*0*..<*d*} **by** *auto*

```
          have ?px ! i = x ! i
          proof (cases i > d)
            have i-le: i < length (base {0..<Suc d} p) using base-length[OF
p-spg'] and ‹i < dm› by auto
            case True hence i ∉ {0..<Suc d} by auto
              from grid-invariant[OF i-le this x-eq] base-out[OF ‹i < dm› this
p-spg']
            show ?thesis using list-update-id and True by auto
          next
            case False hence d = i using ‹¬ i < d› by auto
              thus ?thesis unfolding ‹d = i› using ‹i < dm› ‹length p = dm›
nth-list-update-eq by auto
          qed
            thus ?thesis using base-out[OF ‹i < dm› ‹i ∉ {0..<d}› px-spg]
base-out[OF ‹i < dm› ‹i ∉ {0..<d}› x-spg] by auto
        qed
      qed
    qed
    ultimately have x ∉ grid ?bx {0..<d} by auto
    thus False using baseE(2)[OF x-spg] by auto
  qed

  have x-grid: ?bx ∈ grid (base {0..<Suc d} p) {d} using grid-shift-base[OF
- p-spg' x-eq[unfolded ds-eq]] unfolding ds-eq by auto

  have p-grid: ?bp ∈ grid (base {0..<Suc d} p) {d} using grid-shift-base[OF -
p-spg' baseE(2)[OF p-spg', where ds={0..<d} ∪ {d}]] unfolding ds-eq by auto

  have l2-φ (?bp ! d) (?bx ! d) = 0
  proof (cases lv ?bx d ≤ lv ?bp d)
    case True from l2-disjoint[OF - x-grid p-grid p-nochild-x this] ‹d < dm›
and base-length[OF p-spg']
    show ?thesis by auto
  next
    case False hence lv ?bx d ≥ lv ?bp d by auto
  from l2-disjoint[OF - p-grid x-grid x-nochild-p this] ‹d < dm› and base-length[OF
p-spg']
    show ?thesis by (auto simp: l2-commutative)
  qed
  hence l2-φ (p ! d) (x ! d) = 0 using base-out[OF ‹d < dm›] p-spg' x-spg by
auto
  hence ∃ d ∈ {0..<Suc d}. l2-φ (p ! d) (x ! d) = 0 by auto
  from prod-zero[OF - this]
  show ?F x = 0 by (auto simp: l2-commutative)
 qed
qed
finally show ?case .
qed
```

theorem *updown*:
assumes *p-spg*: $p \in$ *sparsegrid dm lm*
shows *updown dm lm* α *p* = $(\sum\ p' \in$ *sparsegrid dm lm*. $\alpha\ p' *$ *l2* $p'\ p)$
proof −
have $p \in$ *sparsegrid'* *dm* **using** *p-spg* **unfolding** *sparsegrid-def sparsegrid'-def lgrid-def* **by** *auto*
have $!!p'.\ p' \in$ *lgrid* (*base* {*0..<dm*} *p*) {*0..<dm*} *lm* $\Longrightarrow$ *length* $p' =$ *dm*
proof −
fix p' **assume** $p' \in$ *lgrid* (*base* {*0..<dm*} *p*) {*0..<dm*} *lm*
with *base-grid*[*OF* ‹$p \in$ *sparsegrid'* *dm*›] **have** $p' \in$ *sparsegrid'* *dm* **unfolding** *lgrid-def* **by** *auto*
thus *length* $p' =$ *dm* **by** *auto*
qed
thus *?thesis*
unfolding *updown-def sparsegrid-def base-start-eq*[*OF p-spg*]
using *updown'*[*OF - p-spg*, **where** *d*=*dm*] *p-spg*[*unfolded sparsegrid-def lgrid-def*]
by (*auto simp*: *atLeast0LessThan p-spg*[*THEN sparsegrid-length*] *l2-eq*)
qed

corollary
fixes α
assumes *p*: $p \in$ *sparsegrid dm lm*
defines $f_\alpha \equiv \lambda x.\ (\sum p \in$ *sparsegrid dm lm*. $\alpha\ p * \Phi\ p\ x)$
shows *updown dm lm* α *p* = $(\int x.\ f_\alpha\ x * \Phi\ p\ x\ \partial(\Pi_M\ d \in \{..<dm\}.$ *lborel*))
unfolding *updown*[*OF p*] *l2-def* f_α*-def sum-distrib-right*
apply (*intro has-bochner-integral-integral-eq*[*symmetric*] *has-bochner-integral-sum*)
apply (*subst mult.assoc*)
apply (*intro has-bochner-integral-mult-right*)
apply (*simp add*: *sparsegrid-length*)
apply (*rule has-bochner-integral-integrable*)
using *p*
apply (*simp add*: *sparsegrid-length* Φ*-def prod.distrib*[*symmetric*])
proof (*rule product-sigma-finite.product-integrable-prod*)
show *product-sigma-finite* ($\lambda d.$ *lborel*) **..**
qed (*auto intro*: *integrable-*φ*2*)

end

8 Imperative Version

theory *Imperative*
imports *UpDown-Scheme Separation-Logic-Imperative-HOL.Sep-Main*
begin

type-synonym *pointmap* = *grid-point* ⇒ *nat*
type-synonym *impgrid* = *rat array*

instance *rat* :: *heap* **..**

```
primrec rat-pair where rat-pair (a, b) = (of-rat a, of-rat b)

declare rat-pair.simps [simp del]

definition
  zipWithA :: ('a::heap ⇒ 'b::heap ⇒ 'a::heap) ⇒ 'a array ⇒ 'b array ⇒ 'a array
Heap
where
 zipWithA f a b = do {
    n ← Array.len a;
    Heap-Monad.fold-map (λn. do {
      x ← Array.nth a n ;
      y ← Array.nth b n ;
      Array.upd n (f x y) a
    }) [0..<n];
    return a
  }

theorem zipWithA [sep-heap-rules]:
 fixes xs ys :: 'a::heap list
 assumes length xs = length ys
 shows < a ↦a xs * b ↦a ys > zipWithA f a b < λr. (a ↦a map (case-prod f)
(zip xs ys)) * b ↦a ys * ↑(a = r) >
proof −
 { fix n and xs :: 'a list
   let ?part-res = λn xs. (map (case-prod f) (zip (take n xs) (take n ys)) @ drop
n xs)
   assume n ≤ length xs length xs = length ys
   then have < a ↦a xs * b ↦a ys > Heap-Monad.fold-map (λn. do {
       x ← Array.nth a n ;
       y ← Array.nth b n ;
       Array.upd n (f x y) a
     }) [0..<n] < λr. a ↦a ?part-res n xs * b ↦a ys >
   proof (induct n arbitrary: xs)
     case 0 then show ?case by sep-auto
   next
     case (Suc n)
     note Suc.hyps [sep-heap-rules]
     have *: (?part-res n xs)[n := f (?part-res n xs ! n) (ys ! n)] =  ?part-res (Suc
n) xs
     using Suc.prems by (simp add: nth-append take-Suc-conv-app-nth upd-conv-take-nth-drop)
     from Suc.prems show ?case
       by (sep-auto simp add: fold-map-append *)
   qed }
 note this[sep-heap-rules]
 show ?thesis
   unfolding zipWithA-def
   by (sep-auto simp add: assms)
qed
```

definition *copy-array* :: *'a::heap array* ⇒ (*'a::heap array*) *Heap* **where**
copy-array a = *Array.freeze a* ⤜ *Array.of-list*

theorem *copy-array* [*sep-heap-rules*]:
< *a* ↦$_a$ *xs* > *copy-array a* < λ*r*. *a* ↦$_a$ *xs* ∗ *r* ↦$_a$ *xs* >
unfolding *copy-array-def*
by *sep-auto*

definition *sum-array* :: *rat array* ⇒ *rat array* ⇒ *unit Heap* **where**
sum-array a b = *zipWithA* (+) *a b* ⤜ *return* ()

theorem *sum-array* [*sep-heap-rules*]:
fixes *xs ys* :: *rat list*
shows *length xs* = *length ys* ⟹ < *a* ↦$_a$ *xs* ∗ *b* ↦$_a$ *ys* > *sum-array a b* < λ*r*.
(*a* ↦$_a$ *map* (λ(*a*, *b*). *a* + *b*) (*zip xs ys*)) ∗ *b* ↦$_a$ *ys* >
unfolding *sum-array-def* **by** *sep-auto*

locale *linearization* =
fixes *dm lm* :: *nat*
fixes *pm* :: *pointmap*
assumes *pm*: *bij-betw pm* (*sparsegrid dm lm*) {..< *card* (*sparsegrid dm lm*)}
begin

lemma *linearizationD*:
p ∈ *sparsegrid dm lm* ⟹ *pm p* < *card* (*sparsegrid dm lm*)
using *pm* **by** (*auto simp*: *bij-betw-def*)

definition *gridI* :: *impgrid* ⇒ (*grid-point* ⇒ *real*) ⇒ *assn* **where**
gridI a v =
(∃$_A$ *xs*. *a* ↦$_a$ *xs* ∗ ↑((∀ *p*∈*sparsegrid dm lm*. *v p* = *of-rat* (*xs* ! *pm p*)) ∧ *length*
xs = *card* (*sparsegrid dm lm*)))

lemma *gridI-nth-rule* [*sep-heap-rules*]:
g ∈ *sparsegrid dm lm* ⟹ < *gridI a v* > *Array.nth a* (*pm g*) <λ*r*. *gridI a v* ∗ ↑
(*of-rat r* = *v g*)>
using *pm* **by** (*sep-auto simp*: *bij-betw-def gridI-def*)

lemma *gridI-upd-rule* [*sep-heap-rules*]:
g ∈ *sparsegrid dm lm* ⟹
< *gridI a v* > *Array.upd* (*pm g*) *x a* <λ*a'*. *gridI a* (*fun-upd v g* (*of-rat x*)) ∗
↑(*a'* = *a*)>
unfolding *gridI-def* **using** *pm*
by (*sep-auto simp*: *bij-betw-def inj-onD intro*!: *nth-list-update-eq*[*symmetric*] *nth-list-update-neq*[*symmetric*])

primrec *upI'* :: *nat* ⇒ *nat* ⇒ *grid-point* ⇒ *impgrid* ⇒ (*rat* ∗ *rat*) *Heap* **where**
upI' d 0 *p a* = *return* (0, 0) |
upI' d (*Suc l*) *p a* = *do* {
(*fl*, *fml*) ← *upI' d l* (*child p left d*) *a* ;

```
    (fmr, fr) ← upI' d l (child p right d) a ;
    val ← Array.nth a (pm p) ;
    Array.upd (pm p) (fml + fmr) a ;
    let result = ((fml + fmr + val / 2 ^ (lv p d) / 2) / 2) ;
    return (fl + result, fr + result)
  }

lemma upI' [sep-heap-rules]:
  assumes lin[simp]: d < dm
    and l: level p + l = lm l = 0 ∨ p ∈ sparsegrid dm lm
  shows < gridI a v > upI' d l p a <λr. let (r', v') = up' d l p v in gridI a v' *
↑(rat-pair r = r') >
  using l
proof (induct l arbitrary: p v)
  note rat-pair.simps [simp]
  case 0 then show ?case by sep-auto
next
  case (Suc l)
  from Suc.prems ‹d < dm›
  have [simp]: level (child p left d) + l = lm level (child p right d) + l = lm p ∈
sparsegrid dm lm
    by (auto simp: sparsegrid-length)

  have [simp]: child p left d ∉ sparsegrid dm lm ⟹ l = 0 child p right d ∉
sparsegrid dm lm ⟹ l = 0
    using Suc.prems by (auto simp: sparsegrid-def lgrid-def)

  note Suc(1)[sep-heap-rules]
  show ?case
   by (sep-auto split: prod.split simp: of-rat-add of-rat-divide of-rat-power of-rat-mult
rat-pair-def Let-def)
qed

primrec downI' :: nat ⇒ nat ⇒ grid-point ⇒ impgrid ⇒ rat ⇒ rat ⇒ unit Heap
where
  downI' d       0 p a fl fr = return () |
  downI' d (Suc l) p a fl fr = do {
     val ← Array.nth a (pm p) ;
     let fm = ((fl + fr) / 2 + val) ;
     Array.upd (pm p) (((fl + fr) / 4 + (1 / 3) * val) / 2 ^ (lv p d)) a ;
     downI' d l (child p left d) a fl fm ;
     downI' d l (child p right d) a fm fr
   }

lemma downI' [sep-heap-rules]:
  assumes lin[simp]: d < dm
    and l: level p + l = lm l = 0 ∨ p ∈ sparsegrid dm lm
  shows < gridI a v > downI' d l p a fl fr <λr. gridI a (down' d l p (of-rat fl)
(of-rat fr) v) >
```

```
  using l
proof (induct l arbitrary: p v fl fr)
  note rat-pair.simps [simp]
  case 0 then show ?case by sep-auto
next
  case (Suc l)
  from Suc.prems ‹d < dm›
  have [simp]: level (child p left d) + l = lm level (child p right d) + l = lm p ∈
sparsegrid dm lm
    by (auto simp: sparsegrid-length)

   have [simp]: child p left d ∉ sparsegrid dm lm ⟹ l = 0 child p right d ∉
sparsegrid dm lm ⟹ l = 0
    using Suc.prems by (auto simp: sparsegrid-def lgrid-def)

  note Suc(1)[sep-heap-rules]
  show ?case
   by (sep-auto split: prod.split simp: of-rat-add of-rat-divide of-rat-power of-rat-mult
rat-pair-def Let-def fun-upd-def)
qed

definition liftI :: (nat ⇒ nat ⇒ grid-point ⇒ impgrid ⇒ unit Heap) ⇒ nat ⇒
impgrid ⇒ unit Heap where
  liftI f d a =
    foldr (λ p n. n ≫ f d (lm − level p) p a) (gridgen (start dm) ({ 0 ..< dm }
− { d }) lm) (return ())

theorem liftI [sep-heap-rules]:
  assumes d < dm
  and f[sep-heap-rules]: ⋀v p. p ∈ lgrid (start dm) ({0..<dm} − {d}) lm ⟹
    < gridI a v > f d (lm − level p) p a <λr. gridI a (f' d (lm − level p) p v) >
  shows < gridI a v > liftI f d a <λr. gridI a (Grid.lift f' dm lm d v) >
proof −
  let ?ds = {0..<dm} − {d} and ?g = gridI a
  { fix ps assume set ps ⊆ set (gridgen (start dm) ?ds lm) and distinct ps
    then have < ?g v >
        foldr (λp n. (n :: unit Heap) ≫ f d (lm − level p) p a) ps (return ())
      <λr. ?g (foldr (λp α. f' d (lm − level p) p α) ps v) >
      by (induct ps arbitrary: v) (sep-auto simp: gridgen-lgrid-eq)+ }
  from this[OF subset-refl gridgen-distinct]
  show ?thesis
    by (simp add: liftI-def Grid.lift-def)
qed

definition upI where upI = liftI (λd l p a. upI' d l p a ≫ return ())

theorem upI [sep-heap-rules]:
  assumes [simp]: d < dm
  shows < gridI a v > upI d a <λr. gridI a (up dm lm d v) >
```

unfolding *up-def upI-def*
by (*sep-auto simp*: *lgrid-def sparsegrid-def lgrid-def split*: *prod.split*
intro: *grid-union-dims*[*of* {*0*..<*dm*} − {*d*} {*0*..<*dm*}])

definition *downI* **where** *downI* = *liftI* (λ*d l p a*. *downI'* *d l p a 0 0*)

theorem *downI* [*sep-heap-rules*]:
assumes [*simp*]: *d* < *dm*
shows < *gridI a v* > *downI d a* <λ*r*. *gridI a* (*down dm lm d v*) >
unfolding *down-def downI-def*
by (*sep-auto simp*: *lgrid-def sparsegrid-def lgrid-def split*: *prod.split*
intro: *grid-union-dims*[*of* {*0*..<*dm*} − {*d*} {*0*..<*dm*}])

theorem *copy-array-gridI* [*sep-heap-rules*]:
< *gridI a v* > *copy-array a* < λ*r*. *gridI a v* ∗ *gridI r v* >
unfolding *gridI-def*
by *sep-auto*

theorem *sum-array-gridI* [*sep-heap-rules*]:
< *gridI a v* ∗ *gridI b w* > *sum-array a b* < λ*r*. *gridI a* (*sum-vector v w*) ∗ *gridI b w* >
unfolding *gridI-def*
by (*sep-auto simp*: *sum-vector-def nth-map linearizationD of-rat-add*)

primrec *updownI'* :: *nat* ⇒ *impgrid* ⇒ *unit Heap* **where**
updownI' 0 a = *return* () |
updownI' (*Suc d*) *a* = *do* {
b ← *copy-array a* ;
upI d a ;
updownI' d a ;
updownI' d b ;
downI d b ;
sum-array a b
}

theorem *updownI'* [*sep-heap-rules*]:
d ≤ *dm* ⟹ < *gridI a v* > *updownI' d a* <λ*r*. *gridI a* (*updown' dm lm d v*) >$_t$
proof (*induct d arbitrary*: *a v*)
case (*Suc d*)
note *Suc.hyps* [*sep-heap-rules*]
from *Suc.prems* **show** *?case*
by *sep-auto*
qed *sep-auto*

definition *updownI* **where** *updownI a* = *updownI' dm a*

theorem *updownI* [*sep-heap-rules*]:
< *gridI a v* > *updownI a* <λ*r*. *gridI a* (*updown dm lm v*) >$_t$
unfolding *updown-def updownI-def* **by** *sep-auto*

end

end

Literatur

[1] J. Hölzl. Automatischer Korrektheitsbeweis von Algorithmen zur hierarchischen Basistransformation. IDP, Institut für Informatik, Technische Universität München, 2009.

[2] D. Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. Verlag Dr. Hut, München, Aug. 2010.