Simplicial complexes and Boolean functions*

Jesús María Aransay Azofra, Alejandro del Campo López, Julius Michaelis

April 18, 2024

Abstract

In this work we formalise the isomorphism between simplicial complexes of dimension n and monotone Boolean functions in n variables, mainly following the definitions and results as introduced by N. A. Scoville [3, Ch. 6]. We also take advantage of the AFP representation of ROBDD (Reduced Ordered Binary Decision Diagrams) [2] to compute the ROBDD representation of a given simplicial complex (by means of the isomorphism to Boolean functions). Some examples of simplicial complexes and associated Boolean functions are also presented.

Contents

1 Introduction **2**

2 Boolean functions **2**

3 Threshold function **3**

4 Simplicial Complexes **4**

5 Simplicial complex induced by a monotone Boolean function **6**

6 The simplicial complex induced by the threshold function **10**

7 Bijection between simplicial complexes and monotone Boolean functions **21**

8 Converting boolean functions to BDTs **26**

9 Relation between type *bool vec* ⇒ *bool* and type *'a boolfunc* **28**

10 Definition of *evasive* Boolean function **30**

*This research was partially funded by Ministerio de Ciencia e Innovación (Spain), grant number PID2020-116641GB-I00.

11 Detour: Lexicographic ordering for lists **30**

12 Executably converting Simplicial Complexes to BDDs **32**

13 Binary operations over Boolean functions and simplicial complexes **43**

1 Introduction

theory *Boolean-functions*
imports
Main
Jordan-Normal-Form.Matrix
begin

2 Boolean functions

Definition of monotonicity

We consider (monotone) Boolean functions over vectors of length n, so that we can later prove that those are isomorphic to simplicial complexes of dimension n (in n vertexes).

locale *boolean-functions*
= **fixes** *n*::*nat*
begin

definition *bool-fun-dim-n* :: (*bool vec* => *bool*) *set*
where *bool-fun-dim-n* = {*f*. *f* ∈ *carrier-vec n* → (*UNIV*::*bool set*)}

definition *monotone-bool-fun* :: (*bool vec* => *bool*) => *bool*
where *monotone-bool-fun* ≡ (*mono-on* (*carrier-vec n*))

definition *monotone-bool-fun-set* :: (*bool vec* => *bool*) *set*
where *monotone-bool-fun-set* = (*Collect monotone-bool-fun*)

Some examples of Boolean functions

definition *bool-fun-top* :: *bool vec* => *bool*
where *bool-fun-top f* = *True*

definition *bool-fun-bot* :: *bool vec* => *bool*
where *bool-fun-bot f* = *False*

end

3 Threshold function

definition *count-true* :: *bool vec* => *nat*
  where *count-true v* = *sum* (λ*i*. *if vec-index v i then 1 else 0*::*nat*) {*0*..<*dim-vec v*}

lemma *vec-index* (*vec* (*5*::*nat*) (λ*i*. *False*)) *2* = *False*
  by *simp*

lemma *vec-index* (*vec* (*5*::*nat*) (λ*i*. *True*)) *3* = *True*
  by *simp*

lemma *count-true* (*vec* (*1*::*nat*) (λ*i*. *True*)) = *1*
  unfolding *count-true-def* **by** *simp*

lemma *count-true* (*vec* (*2*::*nat*) (λ*i*. *True*)) = *2*
  unfolding *count-true-def* **by** *simp*

lemma *count-true* (*vec* (*5*::*nat*) (λ*i*. *True*)) = *5*
  unfolding *count-true-def* **by** *simp*

The threshold function is a Boolean function which also satisfies the condition of being *evasive*. We follow the definition by Scoville [3, Problem 6.5].

definition *bool-fun-threshold* :: *nat* => (*bool vec* => *bool*)
  where *bool-fun-threshold i* = (λ*v*. *if i* ≤ *count-true v then True else False*)

context *boolean-functions*
begin

lemma *mono-on UNIV bool-fun-top*
  by (*simp add*: *bool-fun-top-def mono-onI monotone-bool-fun-def*)

lemma *monotone-bool-fun bool-fun-top*
  by (*simp add*: *bool-fun-top-def mono-onI monotone-bool-fun-def*)

lemma *mono-on UNIV bool-fun-bot*
  by (*simp add*: *bool-fun-bot-def mono-onI monotone-bool-fun-def*)

lemma *monotone-bool-fun bool-fun-bot*
  by (*simp add*: *bool-fun-bot-def mono-onI monotone-bool-fun-def*)

lemma
  monotone-count-true:
  assumes *ulev*: (*u*::*bool vec*) ≤ *v*
  shows *count-true u* ≤ *count-true v*
  unfolding *count-true-def*
  using *Groups-Big.ordered-comm-monoid-add-class.sum-mono*
    [*of* {*0*..<*dim-vec u*}

(λi. *if vec-index u i then 1 else 0*)
(λi. *if vec-index v i then 1 else 0*)]
using *ulev*
unfolding *Matrix.less-eq-vec-def*
by *fastforce*

The threshold function is monotone.

lemma
monotone-threshold:
assumes *ulev*: (*u*::*bool vec*) ≤ *v*
shows *bool-fun-threshold n u* ≤ *bool-fun-threshold n v*
unfolding *bool-fun-threshold-def*
using *monotone-count-true* [*OF ulev*] **by** *simp*

lemma
assumes (*u*::*bool vec*) ≤ *v*
and *n* < *dim-vec u*
shows *bool-fun-threshold n u* ≤ *bool-fun-threshold n v*
using *monotone-threshold* [*OF assms*(*1*)] **.**

lemma *mono-on UNIV* (*bool-fun-threshold n*)
by (*meson mono-onI monotone-bool-fun-def monotone-threshold*)

lemma *monotone-bool-fun* (*bool-fun-threshold n*)
unfolding *monotone-bool-fun-def*
by (*meson boolean-functions.monotone-threshold mono-onI*)

end

end

theory *Simplicial-complex*
imports
Boolean-functions
begin

4 Simplicial Complexes

lemma *Pow-singleton*: *Pow* {*a*} = {{},{*a*}} **by** *auto*

lemma *Pow-pair*: *Pow* {*a*,*b*} = {{},{*a*},{*b*},{*a*,*b*}} **by** *auto*

locale *simplicial-complex*
= **fixes** *n*::*nat*
begin

A simplex (in n vertexes) is any set of vertexes, including the empty set.

definition *simplices* :: *nat set set*

where *simplices* = *Pow* {*0*..<*n*}

lemma {} ∈ *simplices*
unfolding *simplices-def* **by** *simp*

lemma {*0*..<*n*} ∈ *simplices*
unfolding *simplices-def* **by** *simp*

lemma *finite-simplex*:
assumes σ ∈ *simplices*
shows *finite* σ
by (*metis Pow-iff assms finite-atLeastLessThan finite-subset simplices-def*)

A simplicial complex (in n vertexes) is a collection of sets of vertexes such that every subset of a set of vertexes also belongs to the simplicial complex.

definition *simplicial-complex* :: *nat set set* => *bool*
where *simplicial-complex K* ≡ (∀σ∈*K*. (σ ∈ *simplices*) ∧ (*Pow* σ) ⊆ *K*)

lemma
finite-simplicial-complex:
assumes *simplicial-complex K*
shows *finite K*
by (*metis assms finite-Pow-iff finite-atLeastLessThan rev-finite-subset simplices-def simplicial-complex-def subsetI*)

lemma *finite-simplices*:
assumes *simplicial-complex K*
and *v* ∈ *K*
shows *finite v*
using *assms finite-simplex simplicial-complex.simplicial-complex-def* **by** *blast*

definition *simplicial-complex-set* :: *nat set set set*
where *simplicial-complex-set* = (*Collect simplicial-complex*)

lemma *simplicial-complex-empty-set*:
fixes *K*::*nat set set*
assumes *k*: *simplicial-complex K*
shows *K* = {} ∨ {} ∈ *K* **using** *k* **unfolding** *simplicial-complex-def Pow-def* **by** *auto*

lemma
simplicial-complex-monotone:
fixes *K*::*nat set set*
assumes *k*: *simplicial-complex K* **and** *s*: *s* ∈ *K* **and** *rs*: *r* ⊆ *s*
shows *r* ∈ *K*
using *k rs s*
unfolding *simplicial-complex-def Pow-def* **by** *auto*

One example of simplicial complex with four simplices.

lemma
assumes *three*: (*3*::*nat*) < *n*
shows *simplicial-complex* {{},{*0*},{*1*},{*2*},{*3*}}
apply (*simp-all add*: *Pow-singleton simplicial-complex-def simplices-def*)
using *Suc-lessD three* **by** *presburger*

lemma ¬ *simplicial-complex* {{*0*,*1*},{*1*}}
by (*simp add*: *Pow-pair simplicial-complex-def*)

Another example of simplicial complex with five simplices.

lemma
assumes *three*: (*3*::*nat*) < *n*
shows *simplicial-complex* {{},{*0*},{*1*},{*2*},{*3*},{*0*,*1*}}
apply (*simp add*: *Pow-pair Pow-singleton simplicial-complex-def simplices-def*)
using *Suc-lessD three* **by** *presburger*

Another example of simplicial complex with ten simplices.

lemma
assumes *three*: (*3*::*nat*) < *n*
shows *simplicial-complex*
{{*2*,*3*},{*1*,*3*},{*1*,*2*},{*0*,*3*},{*0*,*2*},{*3*},{*2*},{*1*},{*0*},{}}
apply (*simp add*: *Pow-pair Pow-singleton simplicial-complex-def simplices-def*)
using *Suc-lessD three* **by** *presburger*

end

5 Simplicial complex induced by a monotone Boolean function

In this section we introduce the definition of the simplicial complex induced by a monotone Boolean function, following the definition in Scoville [3, Def. 6.9].

First we introduce the set of tuples for which a Boolean function is *False*.

definition *ceros-of-boolean-input* :: *bool vec* => *nat set*
where *ceros-of-boolean-input v* = {*x*. *x* < *dim-vec v* ∧ *vec-index v x* = *False*}

lemma
ceros-of-boolean-input-l-dim:
assumes *a*: *a* ∈ *ceros-of-boolean-input v*
shows *a* < *dim-vec v*
using *a* **unfolding** *ceros-of-boolean-input-def* **by** *simp*

lemma *ceros-of-boolean-input v* = {*x*. *x* < *dim-vec v* ∧ ¬ *vec-index v x*}
unfolding *ceros-of-boolean-input-def* **by** *simp*

lemma
ceros-of-boolean-input-complementary:
shows *ceros-of-boolean-input v* = {*x*. *x* < *dim-vec v*} − {*x*. *vec-index v x*}
unfolding *ceros-of-boolean-input-def* **by** *auto*

lemma *monotone-ceros-of-boolean-input*:
fixes *r* **and** *s*::*bool vec*
assumes *r-le-s*: *r* ≤ *s*
shows *ceros-of-boolean-input s* ⊆ *ceros-of-boolean-input r*
proof (*intro subsetI*, *unfold ceros-of-boolean-input-def*, *intro CollectI*, *rule conjI*)
fix *x*
assume *x* ∈ {*x*. *x* < *dim-vec s* ∧ *vec-index s x* = *False*}
hence *xl*: *x* < *dim-vec s* **and** *nr*: *vec-index s x* = *False* **by** *simp-all*
show *vec-index r x* = *False*
using *r-le-s nr xl* **unfolding** *less-eq-vec-def*
by *auto*
show *x* < *dim-vec r*
using *r-le-s xl* **unfolding** *less-eq-vec-def*
by *auto*
qed

We introduce here instantiations of the typ*bool* type for the type classes class*zero* and class*one* that will simplify notation at some points:

instantiation *bool* :: {*zero*,*one*}
begin

definition
zero-bool-def: *0* == *False*

definition
one-bool-def: *1* == *True*

instance proof qed

end

Definition of the simplicial complex induced by a Boolean function f in dimension n.

definition
simplicial-complex-induced-by-monotone-boolean-function
:: *nat* => (*bool vec* => *bool*) => *nat set set*
where *simplicial-complex-induced-by-monotone-boolean-function n f* =
{*y*. ∃ *x*. *dim-vec x* = *n* ∧ *f x* ∧ *ceros-of-boolean-input x* = *y*}

The simplicial complex induced by a Boolean function is a subset of the powerset of the set of vertexes.

lemma

```
  simplicial-complex-induced-by-monotone-boolean-function-subset:
  simplicial-complex-induced-by-monotone-boolean-function n (v::bool vec => bool)
    ⊆ Pow (({0..n}::nat set))
  using ceros-of-boolean-input-def
   simplicial-complex-induced-by-monotone-boolean-function-def
  by force

corollary
  simplicial-complex-induced-by-monotone-boolean-function n (v::bool vec => bool)
    ⊆ Pow ((UNIV::nat set)) by simp
```

The simplicial complex induced by a monotone Boolean function is a simplicial complex. This result is proven in Scoville as part of the proof of Proposition 6.16 [3, Prop. 6.16].

```
context simplicial-complex
begin

lemma
  monotone-bool-fun-induces-simplicial-complex:
  assumes mon: boolean-functions.monotone-bool-fun n f
 shows simplicial-complex (simplicial-complex-induced-by-monotone-boolean-function
n f)
  unfolding simplicial-complex-def
proof (rule, unfold simplicial-complex-induced-by-monotone-boolean-function-def,
safe)
    fix σ :: nat set and x :: bool vec
    assume fx: f x and dim-vec-x: n = dim-vec x
    show ceros-of-boolean-input x ∈ simplicial-complex.simplices (dim-vec x)
      using ceros-of-boolean-input-def dim-vec-x simplices-def by force
  next
    fix σ :: nat set and x :: bool vec and τ :: nat set
   assume fx: f x and dim-vec-x: n = dim-vec x and tau-def: τ ⊆ ceros-of-boolean-input
x
    show ∃ xb. dim-vec xb = dim-vec x ∧ f xb ∧ ceros-of-boolean-input xb = τ
    proof (rule exI [of - vec n (λi. if i ∈ τ then False else True)], intro conjI)
     show dim-vec (vec n (λi. if i ∈ τ then False else True)) = dim-vec x
      unfolding dim-vec using dim-vec-x .
     from mon have mono: mono-on (carrier-vec n) f
      unfolding boolean-functions.monotone-bool-fun-def .
     show f (vec n (λi. if i ∈ τ then False else True))
     proof −
      have f x ≤ f (vec n (λi. if i ∈ τ then False else True))
      proof (rule mono-onD [OF mono])
        show x ∈ carrier-vec n using dim-vec-x by simp
        show vec n (λi. if i ∈ τ then False else True) ∈ carrier-vec n by simp
        show x ≤ vec n (λi. if i ∈ τ then False else True)
          using tau-def dim-vec-x unfolding ceros-of-boolean-input-def
          using less-eq-vec-def by fastforce
      qed
```

```
    thus ?thesis using fx by simp
  qed
  show ceros-of-boolean-input (vec n (λi. if i ∈ τ then False else True)) = τ
    using ‹τ ⊆ ceros-of-boolean-input x› ceros-of-boolean-input-def dim-vec-x by
auto
 qed
qed

end
```

Example 6.10 in Scoville, the threshold function for 2 in dimension 4 (with vertexes 0,1,2,3)

```
definition bool-fun-threshold-2-3 :: bool vec => bool
 where bool-fun-threshold-2-3 = (λv. if 2 ≤ count-true v then True else False)

lemma set-list-four: shows {0..<4} = set [0,1,2,3::nat] by auto

lemma comp-fun-commute-lambda:
 comp-fun-commute-on UNIV ((+)
 ∘ (λi. if vec 4 f $ i then 1 else (0::nat)))
 unfolding comp-fun-commute-on-def by auto

lemma bool-fun-threshold-2-3
       (vec 4 (λi. if i = 0 ∨ i = 1 then True else False)) = True
 unfolding bool-fun-threshold-2-3-def
 unfolding count-true-def
 unfolding dim-vec
 unfolding sum.eq-fold
 using index-vec [of - 4]
 apply auto
 unfolding set-list-four
 unfolding comp-fun-commute-on.fold-set-fold-remdups [OF comp-fun-commute-lambda,
simplified]
 by simp

lemma
 0 ∉ ceros-of-boolean-input (vec 4 (λi. if i = 0 ∨ i = 1 then True else False))
 and 1 ∉ ceros-of-boolean-input (vec 4 (λi. if i = 0 ∨ i = 1 then True else False))
 and 2 ∈ ceros-of-boolean-input (vec 4 (λi. if i = 0 ∨ i = 1 then True else False))
 and 3 ∈ ceros-of-boolean-input (vec 4 (λi. if i = 0 ∨ i = 1 then True else False))
 and {2,3} ⊆ ceros-of-boolean-input (vec 4 (λi. if i = 0 ∨ i = 1 then True else
False))
 unfolding ceros-of-boolean-input-def by simp-all

lemma bool-fun-threshold-2-3 (vec 4 (λi. if i = 3 then True else False)) = False
 unfolding bool-fun-threshold-2-3-def
 unfolding count-true-def
 unfolding dim-vec
 unfolding sum.eq-fold
```

using *index-vec* [*of - 4*]
apply *auto*
unfolding *set-list-four*
unfolding *comp-fun-commute-on.fold-set-fold-remdups* [*OF comp-fun-commute-lambda*, *simplified*]
by *simp*

lemma *bool-fun-threshold-2-3* (*vec 4* (λ*i*. *if i = 0 then False else True*))
unfolding *bool-fun-threshold-2-3-def*
unfolding *count-true-def*
unfolding *dim-vec*
unfolding *sum.eq-fold*
using *index-vec* [*of - 4*]
apply *auto*
unfolding *set-list-four*
unfolding *comp-fun-commute-on.fold-set-fold-remdups* [*OF comp-fun-commute-lambda*, *simplified*]
by *simp*

6 The simplicial complex induced by the threshold function

lemma
empty-set-in-simplicial-complex-induced:
{} ∈ *simplicial-complex-induced-by-monotone-boolean-function 4 bool-fun-threshold-2-3*
unfolding *simplicial-complex-induced-by-monotone-boolean-function-def*
unfolding *bool-fun-threshold-2-3-def*
apply *rule*
apply (*rule exI* [*of - vec 4* (λ*x*. *True*)])
unfolding *count-true-def ceros-of-boolean-input-def* **by** *auto*

lemma *singleton-in-simplicial-complex-induced*:
assumes *x*: *x* < *4*
shows {*x*} ∈ *simplicial-complex-induced-by-monotone-boolean-function 4 bool-fun-threshold-2-3*
(**is** *?A* ∈ *simplicial-complex-induced-by-monotone-boolean-function 4 bool-fun-threshold-2-3*)
proof (*unfold simplicial-complex-induced-by-monotone-boolean-function-def*, *rule*,
rule exI [*of - vec 4* (λ*i*. *if i* ∈ *?A then False else True*)],
intro conjI)
show *dim-vec* (*vec 4* (λ*i*. *if i* ∈ {*x*} *then False else True*)) = *4* **by** *simp*
show *bool-fun-threshold-2-3* (*vec 4* (λ*i*. *if i* ∈ *?A then False else True*))
unfolding *bool-fun-threshold-2-3-def*
unfolding *count-true-def*
unfolding *dim-vec*
unfolding *sum.eq-fold*
using *index-vec* [*of - 4*]
apply *auto*
unfolding *set-list-four*
unfolding *comp-fun-commute-on.fold-set-fold-remdups* [*OF comp-fun-commute-lambda*,

simplified]
by *simp*
show *ceros-of-boolean-input* (*vec 4* (λ*i*. *if i* ∈ *?A then False else True*)) = *?A*
unfolding *ceros-of-boolean-input-def* **using** *x* **by** *auto*
qed

lemma *pair-in-simplicial-complex-induced*:
assumes *x*: *x* < *4* **and** *y*: *y* < *4*
shows {*x*,*y*} ∈ *simplicial-complex-induced-by-monotone-boolean-function 4 bool-fun-threshold-2-3*
(**is** *?A* ∈ *simplicial-complex-induced-by-monotone-boolean-function 4 bool-fun-threshold-2-3*)
proof (*unfold simplicial-complex-induced-by-monotone-boolean-function-def*, *rule*,
rule exI [*of* - *vec 4* (λ*i*. *if i* ∈ *?A then False else True*)],
intro conjI)
show *dim-vec* (*vec 4* (λ*i*. *if i* ∈ {*x*, *y*} *then False else True*)) = *4* **by** *simp*
show *bool-fun-threshold-2-3* (*vec 4* (λ*i*. *if i* ∈ *?A then False else True*))
unfolding *bool-fun-threshold-2-3-def*
unfolding *count-true-def*
unfolding *dim-vec*
unfolding *sum.eq-fold*
using *index-vec* [*of* - *4*]
apply *auto*
unfolding *set-list-four*
unfolding *comp-fun-commute-on.fold-set-fold-remdups* [*OF comp-fun-commute-lambda*,
simplified]
by *simp*
show *ceros-of-boolean-input* (*vec 4* (λ*i*. *if i* ∈ *?A then False else True*)) = *?A*
unfolding *ceros-of-boolean-input-def* **using** *x y* **by** *auto*
qed

lemma *finite-False*: *finite* {*x*. *x* < *dim-vec a* ∧ *vec-index* (*a*::*bool vec*) *x* = *False*}
by *auto*

lemma *finite-True*: *finite* {*x*. *x* < *dim-vec a* ∧ *vec-index* (*a*::*bool vec*) *x* = *True*}
by *auto*

lemma *UNIV-disjoint*: {*x*. *x* < *dim-vec a* ∧ *vec-index* (*a*::*bool vec*) *x* = *True*}
∩ {*x*. *x* < *dim-vec a* ∧ *vec-index* (*a*::*bool vec*) *x* = *False*} = {}
by *auto*

lemma *UNIV-union*: {*x*. *x* < *dim-vec a* ∧ *vec-index* (*a*::*bool vec*) *x* = *True*}
∪ {*x*. *x* < *dim-vec a* ∧ *vec-index* (*a*::*bool vec*) *x* = *False*} = {*x*. *x* < *dim-vec a*}
by *auto*

lemma *card-UNIV-union*:
card {*x*. *x* < *dim-vec a* ∧ *vec-index* (*a*::*bool vec*) *x* = *True*}
+ *card* {*x*. *x* < *dim-vec a* ∧ *vec-index* (*a*::*bool vec*) *x* = *False*}
= *card* {*x*. *x* < *dim-vec a*}
(**is** *card ?true* + *card ?false* = -)
proof −

have *card ?true + card ?false = card (?true ∪ ?false) + card (?true ∩ ?false)*
using *card-Un-Int [OF finite-True [of a] finite-False [of a]]* **.**
also have *... = card {x. x < dim-vec a}*
unfolding *UNIV-union UNIV-disjoint* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *card-complementary*:
card (ceros-of-boolean-input v)
+ card {x. x < (dim-vec v) ∧ (vec-index v x = True)} = (dim-vec v)
unfolding *ceros-of-boolean-input-def*
using *card-UNIV-union [of v]* **by** *simp*

corollary
card-ceros-of-boolean-input:
shows *card (ceros-of-boolean-input a) ≤ dim-vec a*
using *card-complementary [of a]* **by** *simp*

lemma
vec-fun:
assumes *v ∈ carrier-vec n*
shows *∃f. v = vec n f* **using** *assms* **unfolding** *carrier-vec-def* **by** *fastforce*

corollary
assumes *dim-vec v = n*
shows *∃f. v = vec n f*
using *carrier-vecI [OF assms]* **unfolding** *carrier-vec-def* **by** *fastforce*

lemma
vec-l-eq:
assumes *i < n*
shows *vec (Suc n) f $ i = vec n f $ i*
by (*simp add*: *assms less-SucI*)

lemma
card-boolean-function:
assumes *d*: *v ∈ carrier-vec n*
shows *card {x. x < n ∧ v $ x = True} = (∑ i = 0..<n. if v $ i then 1 else (0::nat))*
using *d* **proof** (*induction n arbitrary*: *v rule*: *nat-less-induct*)
case (*1 n*)
assume *hyp*: *∀ m<n. ∀ x. x ∈ carrier-vec m ⟶*
card {xa. xa < m ∧ x $ xa = True} = (∑ i = 0..<m. if x $ i then 1 else 0)
and *d*: *v ∈ carrier-vec n*
show *card {x. x < n ∧ v $ x = True} = (∑ i = 0..<n. if v $ i then 1 else 0)*
using *d* **proof** (*cases n*)
case *0*
then show *?thesis* **by** *simp*
next

case (*Suc m*)
assume *v*: *v* ∈ *carrier-vec n*
obtain *f* :: *nat* => *bool* **where** *v-f*: *v* = *vec n f* **using** *vec-fun* [*OF v*] **by** *auto*
have *card* {*x*. *x* < *m* ∧ (*vec m f*) $ *x* = *True*} = (∑ *i* = *0*..<*m*. *if* (*vec m f*) $ *i then 1 else 0*)
using *hyp v Suc* **by** *simp*
show *?thesis* **unfolding** *v-f* **unfolding** *Suc*
proof (*cases vec* (*Suc m*) *f* $ *m* = *True*)
case *True*
have *one*: {*x*. *x* < *Suc m* ∧ *vec* (*Suc m*) *f* $ *x* = *True*} =
({*x*. *x* < *m* ∧ *vec* (*Suc m*) *f* $ *x* = *True*} ∪ {*x*. *x* = *m* ∧ (*vec* (*Suc m*) *f*) $ *x* = *True*})
by *auto*
have *two*: *disjnt* {*x*. *x* < *m* ∧ *vec* (*Suc m*) *f* $ *x* = *True*} {*x*. *x* = *m* ∧ (*vec* (*Suc m*) *f*) $ *x* = *True*}
using *disjnt-iff* **by** *blast*
have *card* {*x*. *x* < *Suc m* ∧ *vec* (*Suc m*) *f* $ *x* = *True*}
= *card* {*x*. *x* < *m* ∧ (*vec* (*Suc m*) *f*) $ *x* = *True*} + *card* {*x*. *x* = *m* ∧ (*vec* (*Suc m*) *f*) $ *x* = *True*}
unfolding *one*
by (*rule card-Un-disjnt* [*OF - - two*], *simp-all*)
also have ... = *card* {*x*. *x* < *m* ∧ (*vec m f*) $ *x* = *True*} + *1*
proof −
have *one*: {*x*. *x* < *m* ∧ *vec* (*Suc m*) *f* $ *x* = *True*} = {*x*. *x* < *m* ∧ *vec m f* $ *x* = *True*}
using *vec-l-eq* [*of - m*] **by** *auto*
have *eq*: {*x*. *x* = *m* ∧ *vec* (*Suc m*) *f* $ *x* = *True*} = {*m*} **using** *True* **by** *auto*
hence *two*: *card* {*x*. *x* = *m* ∧ *vec* (*Suc m*) *f* $ *x* = *True*} = *1* **by** *simp*
show *?thesis* **using** *one two* **by** *simp*
qed
finally have *lhs*: *card* {*x*. *x* < *Suc m* ∧ *vec* (*Suc m*) *f* $ *x* = *True*} = *card* {*x*. *x* < *m* ∧ *vec m f* $ *x* = *True*} + *1* **.**
have (∑ *i* = *0*..<*Suc m*. *if vec* (*Suc m*) *f* $ *i then 1 else 0*) =
(∑ *i* = *0*..<*m*. *if vec* (*Suc m*) *f* $ *i then 1 else 0*) + (*if vec* (*Suc m*) *f* $ *m then 1 else 0*)
by *simp*
also have ... = (∑ *i* = *0*..<*m*. *if vec m f* $ *i then 1 else 0*) + *1*
using *vec-l-eq* [*of - m*] *True* **by** *simp*
finally have *rhs*: (∑ *i* = *0*..<*Suc m*. *if vec* (*Suc m*) *f* $ *i then 1 else 0*) =
(∑ *i* = *0*..<*m*. *if vec m f* $ *i then 1 else 0*) + *1* **.**
show *card* {*x*. *x* < *Suc m* ∧ *vec* (*Suc m*) *f* $ *x* = *True*} =
(∑ *i* = *0*..<*Suc m*. *if vec* (*Suc m*) *f* $ *i then 1 else 0*)
unfolding *lhs rhs* **using** *hyp Suc* **by** *simp*
next
case *False*
have *one*: {*x*. *x* < *Suc m* ∧ *vec* (*Suc m*) *f* $ *x* = *True*} =
({*x*. *x* < *m* ∧ *vec* (*Suc m*) *f* $ *x* = *True*} ∪ {*x*. *x* = *m* ∧ (*vec* (*Suc m*) *f*) $ *x* = *True*})

```
      by auto
    have two: disjnt {x. x < m ∧ vec (Suc m) f $ x = True} {x. x = m ∧ (vec
(Suc m) f) $ x = True}
      using disjnt-iff by blast
    have card {x. x < Suc m ∧ vec (Suc m) f $ x = True}
        = card {x. x < m ∧ (vec (Suc m) f) $ x = True} + card {x. x = m ∧
(vec (Suc m) f) $ x = True}
      unfolding one
      by (rule card-Un-disjnt [OF - - two], simp-all)
    also have ... = card {x. x < m ∧ (vec  m f) $ x = True} + 0
    proof −
      have one: {x. x < m ∧ vec (Suc m) f $ x = True} = {x. x < m ∧ vec m f
$ x = True}
        using vec-l-eq [of - m] by auto
      have eq: {x. x = m ∧ vec (Suc m) f $ x = True} = {} using False by auto
      hence two: card {x. x = m ∧ vec (Suc m) f $ x = True} = 0 by simp
      show ?thesis using one two by simp
    qed
    finally have lhs: card {x. x < Suc m ∧ vec (Suc m) f $ x = True} = card
{x. x < m ∧ vec m f $ x = True} + 0 .
    have (∑ i = 0..<Suc m. if vec (Suc m) f $ i then 1 else 0) =
        (∑ i = 0..<m. if vec (Suc m) f $ i then 1 else 0) + (if vec (Suc m) f $
m then 1 else 0)
      by simp
    also have ... = (∑ i = 0..<m. if vec m f $ i then 1 else 0)
      using vec-l-eq [of - m] False by simp
    finally have rhs: (∑ i = 0..<Suc m. if vec (Suc m) f $ i then 1 else 0) =
      (∑ i = 0..<m. if vec m f $ i then 1 else 0) .
    show card {x. x < Suc m ∧ vec (Suc m) f $ x = True} =
      (∑ i = 0..<Suc m. if vec (Suc m) f $ i then 1 else 0)
      unfolding lhs rhs using hyp Suc by simp
  qed
 qed
qed

lemma card-ceros-count-UNIV:
 shows card (ceros-of-boolean-input a) + count-true ((a::bool vec)) = dim-vec a
 using card-complementary [of a]
 using card-boolean-function
 unfolding ceros-of-boolean-input-def
 unfolding count-true-def by simp
```

We calculate the carrier set of the *ceros-of-boolean-input* function for dimensions 2, 3 and 4.

Vectors of dimension 2.

```
lemma
 dim-vec-2-cases:
 assumes dx: dim-vec x = 2
```

shows $(x\ \$\ 0 = x\ \$\ 1 = True) \vee (x\ \$\ 0 = False \wedge x\ \$\ 1 = True)$
$\vee (x\ \$\ 0 = True \wedge x\ \$\ 1 = False) \vee (x\ \$\ 0 = x\ \$\ 1 = False)$
by *auto*

lemma *tt-2*: **assumes** *dx*: *dim-vec x = 2*
and *be*: $x\ \$\ 0 = True \wedge x\ \$\ 1 = True$
shows *ceros-of-boolean-input x* = {}
using *dx be* **unfolding** *ceros-of-boolean-input-def* **using** *less-2-cases* **by** *auto*

lemma *tf-2*: **assumes** *dx*: *dim-vec x = 2*
and *be*: $x\ \$\ 0 = True \wedge x\ \$\ 1 = False$
shows *ceros-of-boolean-input x* = {*1*}
using *dx be* **unfolding** *ceros-of-boolean-input-def* **using** *less-2-cases* **by** *auto*

lemma *ft-2*: **assumes** *dx*: *dim-vec x = 2*
and *be*: $x\ \$\ 0 = False \wedge x\ \$\ 1 = True$
shows *ceros-of-boolean-input x* = {*0*}
using *dx be* **unfolding** *ceros-of-boolean-input-def* **using** *less-2-cases* **by** *auto*

lemma *ff-2*: **assumes** *dx*: *dim-vec x = 2*
and *be*: $x\ \$\ 0 = False \wedge x\ \$\ 1 = False$
shows *ceros-of-boolean-input x* = {*0,1*}
using *dx be* **unfolding** *ceros-of-boolean-input-def* **using** *less-2-cases* **by** *auto*

lemma
assumes *dx*: *dim-vec x = 2*
shows *ceros-of-boolean-input x* ∈ {{},{*0*},{*1*},{*0,1*}}
using *dim-vec-2-cases* [*OF*]
using *tt-2* [*OF dx*] *tf-2* [*OF dx*] *ft-2* [*OF dx*] *ff-2* [*OF dx*]
by (*metis insertCI*)

Vectors of dimension 3.

lemma *less-3-cases*:
assumes *n*: *n < 3* **shows** *n = 0 ∨ n = 1 ∨ n = (2::nat)*
using *n* **by** *linarith*

lemma
dim-vec-3-cases:
assumes *dx*: *dim-vec x = 3*
shows $(x\ \$\ 0 = x\ \$\ 1 = x\ \$\ 2 = False) \vee (x\ \$\ 0 = x\ \$\ 1 = False \wedge x\ \$\ 2 = True)$
$\vee (x\ \$\ 0 = x\ \$\ 2 = False \wedge x\ \$\ 1 = True) \vee (x\ \$\ 0 = False \wedge x\ \$\ 1 = x\ \$\ 2 = True)$
$\vee (x\ \$\ 0 = True \wedge x\ \$\ 1 = x\ \$\ 2 = False) \vee (x\ \$\ 0 = x\ \$\ 2 = True \wedge x\ \$\ 1 = False)$
$\vee (x\ \$\ 0 = x\ \$\ 1 = True \wedge x\ \$\ 2 = False) \vee (x\ \$\ 0 = x\ \$\ 1 = x\ \$\ 2 = True)$
by *auto*

lemma *fff-3*: **assumes** *dx*: *dim-vec x = 3*

and *be*: *x $ 0 = False ∧ x $ 1 = False ∧ x $ 2 = False*
shows *ceros-of-boolean-input x = {0,1,2}*
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-3-cases* **by** *auto*

lemma *fft-3*: **assumes** *dx*: *dim-vec x = 3*
and *be*: *x $ 0 = False ∧ x $ 1 = False ∧ x $ 2 = True*
shows *ceros-of-boolean-input x = {0,1}*
using *dx be* **unfolding** *ceros-of-boolean-input-def*
using *less-3-cases* **by** *auto*

lemma *ftf-3*: **assumes** *dx*: *dim-vec x = 3*
and *be*: *x $ 0 = False ∧ x $ 1 = True ∧ x $ 2 = False*
shows *ceros-of-boolean-input x = {0,2}*
using *dx be* **unfolding** *ceros-of-boolean-input-def*
using *less-3-cases* **by** *fastforce*

lemma *ftt-3*: **assumes** *dx*: *dim-vec x = 3*
and *be*: *x $ 0 = False ∧ x $ 1 = True ∧ x $ 2 = True*
shows *ceros-of-boolean-input x = {0}*
using *dx be* **unfolding** *ceros-of-boolean-input-def*
using *less-3-cases* **by** *auto*

lemma *tff-3*: **assumes** *dx*: *dim-vec x = 3*
and *be*: *x $ 0 = True ∧ x $ 1 = False ∧ x $ 2 = False*
shows *ceros-of-boolean-input x = {1,2}*
using *dx be* **unfolding** *ceros-of-boolean-input-def*
using *less-3-cases* **by** *auto*

lemma *tft-3*: **assumes** *dx*: *dim-vec x = 3*
and *be*: *x $ 0 = True ∧ x $ 1 = False ∧ x $ 2 = True*
shows *ceros-of-boolean-input x = {1}*
using *dx be* **unfolding** *ceros-of-boolean-input-def*
using *less-3-cases* **by** *auto*

lemma *ttf-3*: **assumes** *dx*: *dim-vec x = 3*
and *be*: *x $ 0 = True ∧ x $ 1 = True ∧ x $ 2 = False*
shows *ceros-of-boolean-input x = {2}*
using *dx be* **unfolding** *ceros-of-boolean-input-def*
using *less-3-cases* **by** *fastforce*

lemma *ttt-3*: **assumes** *dx*: *dim-vec x = 3*
and *be*: *x $ 0 = True ∧ x $ 1 = True ∧ x $ 2 = True*
shows *ceros-of-boolean-input x = {}*
using *dx be* **unfolding** *ceros-of-boolean-input-def*
using *less-3-cases* **by** *auto*

lemma

assumes *dx*: *dim-vec x = 3*
shows *ceros-of-boolean-input x ∈ {{},{0},{1},{2},{0,1},{0,2},{1,2},{0,1,2}}*
using *dim-vec-3-cases* [*OF*]
using *fff-3* [*OF dx*] *fft-3* [*OF dx*] *ftf-3* [*OF dx*] *ftt-3* [*OF dx*]
using *tff-3* [*OF dx*] *tft-3* [*OF dx*] *ttf-3* [*OF dx*] *ttt-3* [*OF dx*]
by (*smt* (*z3*) *insertCI*)

Vectors of dimension 4.

lemma *less-4-cases*:
assumes *n*: *n < 4*
shows *n = 0 ∨ n = 1 ∨ n = 2 ∨ n = (3::nat)*
using *n* **by** *linarith*

lemma
dim-vec-4-cases:
assumes *dx*: *dim-vec x = 4*
shows (*x $ 0 = x $ 1 = x $ 2 = x $ 3 = False*) ∨ (*x $ 0 = x $ 1 = x $ 2 = False ∧ x $ 3 = True*)
∨ (*x $ 0 = x $ 1 = x $ 3 = False ∧ x $ 2 = True*) ∨ (*x $ 0 = x $ 1 = False ∧ x $ 2 = x $ 3 = True*)
∨ (*x $ 0 = x $ 2 = x $ 3 = False ∧ x $ 1 = True*) ∨ (*x $ 0 = x $ 2 = False ∧ x $ 1 = x $ 3 = True*)
∨ (*x $ 0 = x $ 3 = False ∧ x $ 1 = x $ 2 = True*) ∨ (*x $ 0 = False ∧ x $ 1 = x $ 2 = x $ 3 = True*)
∨ (*x $ 0 = True ∧ x $ 1 = x $ 2 = x $ 3 = False*) ∨ (*x $ 0 = x $ 3 = True ∧ x $ 1 = x $ 2 = False*)
∨ (*x $ 0 = x $ 2 = True ∧ x $ 1 = x $ 3 = False*) ∨ (*x $ 0 = x $ 2 = x $ 3 = True ∧ x $ 1 = False*)
∨ (*x $ 0 = x $ 1 = True ∧ x $ 2 = x $ 3 = False*) ∨ (*x $ 0 = x $ 1 = x $ 3 = True ∧ x $ 2 = False*)
∨ (*x $ 0 = x $ 1 = x $ 2 = True ∧ x $ 3 = False*) ∨ (*x $ 0 = x $ 1 = x $ 2 = x $ 3 = True*)
by *blast*

lemma *ffff-4*: **assumes** *dx*: *dim-vec x = 4*
and *be*: *x $ 0 = False ∧ x $ 1 = False ∧ x $ 2 = False ∧ x $ 3 = False*
shows *ceros-of-boolean-input x = {0,1,2,3}*
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *ffft-4*: **assumes** *dx*: *dim-vec x = 4*
and *be*: *x $ 0 = False ∧ x $ 1 = False ∧ x $ 2 = False ∧ x $ 3 = True*
shows *ceros-of-boolean-input x = {0,1,2}*
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *fftf-4*: **assumes** *dx*: *dim-vec x = 4*

and *be*: $x \$ 0 = False \wedge x \$ 1 = False \wedge x \$ 2 = True \wedge x \$ 3 = False$
shows *ceros-of-boolean-input* $x = \{0,1,3\}$
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *fftt-4*: **assumes** *dx*: *dim-vec* $x = 4$
and *be*: $x \$ 0 = False \wedge x \$ 1 = False \wedge x \$ 2 = True \wedge x \$ 3 = True$
shows *ceros-of-boolean-input* $x = \{0,1\}$
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *ftff-4*: **assumes** *dx*: *dim-vec* $x = 4$
and *be*: $x \$ 0 = False \wedge x \$ 1 = True \wedge x \$ 2 = False \wedge x \$ 3 = False$
shows *ceros-of-boolean-input* $x = \{0,2,3\}$
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *ftft-4*: **assumes** *dx*: *dim-vec* $x = 4$
and *be*: $x \$ 0 = False \wedge x \$ 1 = True \wedge x \$ 2 = False \wedge x \$ 3 = True$
shows *ceros-of-boolean-input* $x = \{0,2\}$
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *fttf-4*: **assumes** *dx*: *dim-vec* $x = 4$
and *be*: $x \$ 0 = False \wedge x \$ 1 = True \wedge x \$ 2 = True \wedge x \$ 3 = False$
shows *ceros-of-boolean-input* $x = \{0,3\}$
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *fttt-4*: **assumes** *dx*: *dim-vec* $x = 4$
and *be*: $x \$ 0 = False \wedge x \$ 1 = True \wedge x \$ 2 = True \wedge x \$ 3 = True$
shows *ceros-of-boolean-input* $x = \{0\}$
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *tfff-4*: **assumes** *dx*: *dim-vec* $x = 4$
and *be*: $x \$ 0 = True \wedge x \$ 1 = False \wedge x \$ 2 = False \wedge x \$ 3 = False$
shows *ceros-of-boolean-input* $x = \{1,2,3\}$
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *tfft-4*: **assumes** *dx*: *dim-vec* $x = 4$

and *be*: $x \$ 0 = True \wedge x \$ 1 = False \wedge x \$ 2 = False \wedge x \$ 3 = True$
shows *ceros-of-boolean-input* $x = \{1,2\}$
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *tftf-4*: **assumes** *dx*: *dim-vec* $x = 4$
and *be*: $x \$ 0 = True \wedge x \$ 1 = False \wedge x \$ 2 = True \wedge x \$ 3 = False$
shows *ceros-of-boolean-input* $x = \{1,3\}$
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *tftt-4*: **assumes** *dx*: *dim-vec* $x = 4$
and *be*: $x \$ 0 = True \wedge x \$ 1 = False \wedge x \$ 2 = True \wedge x \$ 3 = True$
shows *ceros-of-boolean-input* $x = \{1\}$
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *ttff-4*: **assumes** *dx*: *dim-vec* $x = 4$
and *be*: $x \$ 0 = True \wedge x \$ 1 = True \wedge x \$ 2 = False \wedge x \$ 3 = False$
shows *ceros-of-boolean-input* $x = \{2,3\}$
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *ttft-4*: **assumes** *dx*: *dim-vec* $x = 4$
and *be*: $x \$ 0 = True \wedge x \$ 1 = True \wedge x \$ 2 = False \wedge x \$ 3 = True$
shows *ceros-of-boolean-input* $x = \{2\}$
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *tttf-4*: **assumes** *dx*: *dim-vec* $x = 4$
and *be*: $x \$ 0 = True \wedge x \$ 1 = True \wedge x \$ 2 = True \wedge x \$ 3 = False$
shows *ceros-of-boolean-input* $x = \{3\}$
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma *tttt-4*: **assumes** *dx*: *dim-vec* $x = 4$
and *be*: $x \$ 0 = True \wedge x \$ 1 = True \wedge x \$ 2 = True \wedge x \$ 3 = True$
shows *ceros-of-boolean-input* $x = \{\}$
using *dx be*
unfolding *ceros-of-boolean-input-def*
using *less-4-cases* **by** *auto*

lemma

ceros-of-boolean-input-set:
assumes *dx*: *dim-vec x = 4*
shows *ceros-of-boolean-input x* ∈ {{},{*0*},{*1*},{*2*},{*3*},{*0,1*},{*0,2*},{*0,3*},{*1,2*},{*1,3*},{*2,3*},
{*0,1,2*},{*0,1,3*},{*0,2,3*},{*1,2,3*},{*0,1,2,3*}}
using *dim-vec-4-cases* [*OF*]
using *ffff-4* [*OF dx*] *ffft-4* [*OF dx*] *fftf-4* [*OF dx*] *fftt-4* [*OF dx*]
using *ftff-4* [*OF dx*] *ftft-4* [*OF dx*] *fttf-4* [*OF dx*] *fttt-4* [*OF dx*]
using *tfff-4* [*OF dx*] *tfft-4* [*OF dx*] *tftf-4* [*OF dx*] *tftt-4* [*OF dx*]
using *ttff-4* [*OF dx*] *ttft-4* [*OF dx*] *tttf-4* [*OF dx*] *tttt-4* [*OF dx*]
by (*smt* (*z3*) *insertCI*)

context *simplicial-complex*
begin

The simplicial complex induced by the monotone Boolean function *bool-fun-threshold-2-3* has the following explicit expression.

lemma
simplicial-complex-induced-by-monotone-boolean-function-4-bool-fun-threshold-2-3:
shows {{},{*0*},{*1*},{*2*},{*3*},{*0,1*},{*0,2*},{*0,3*},{*1,2*},{*1,3*},{*2,3*}}
= *simplicial-complex-induced-by-monotone-boolean-function 4 bool-fun-threshold-2-3*
(**is** {{},*?a*,*?b*,*?c*,*?d*,*?e*,*?f*,*?g*,*?h*,*?i*,*?j*} = -)
proof (*rule*)
show {{},*?a*,*?b*,*?c*,*?d*,*?e*,*?f*,*?g*,*?h*,*?i*,*?j*}
⊆ *simplicial-complex-induced-by-monotone-boolean-function 4 bool-fun-threshold-2-3*
by (*simp add*:
empty-set-in-simplicial-complex-induced
singleton-in-simplicial-complex-induced pair-in-simplicial-complex-induced)+
show *simplicial-complex-induced-by-monotone-boolean-function 4 bool-fun-threshold-2-3*
⊆ {{},*?a*,*?b*,*?c*,*?d*,*?e*,*?f*,*?g*,*?h*,*?i*,*?j*}
unfolding *simplicial-complex-induced-by-monotone-boolean-function-def*
unfolding *bool-fun-threshold-2-3-def*
proof
fix *y*::*nat set*
assume *y*: *y* ∈ {*y*. ∃ *x*. *dim-vec x = 4* ∧ (*if 2* ≤ *count-true x then True else False*) ∧ *ceros-of-boolean-input x = y*}
then obtain *x*::*bool vec*
where *ct-ge-2*: (*if 2* ≤ *count-true x then True else False*)
and *cx*: *ceros-of-boolean-input x = y* **and** *dx*: *dim-vec x = 4* **by** *auto*
have *count-true x + card* (*ceros-of-boolean-input x*) = *dim-vec x*
using *card-ceros-count-UNIV* [*of x*] **by** *simp*
hence *card* (*ceros-of-boolean-input x*) ≤ *2*
using *ct-ge-2*
using *card-boolean-function*
using *dx* **by** *presburger*
hence *card-le*: *card y* ≤ *2* **using** *cx* **by** *simp*
have *y* ∈ {{},*?a*,*?b*,*?c*,*?d*,*?e*,*?f*,*?g*,*?h*,*?i*,*?j*}
proof (*rule ccontr*)
assume *y* ∉ {{},*?a*,*?b*,*?c*,*?d*,*?e*,*?f*,*?g*,*?h*,*?i*,*?j*}
then have *y-nin*: *y* ∉ *set* [{},*?a*,*?b*,*?c*,*?d*,*?e*,*?f*,*?g*,*?h*,*?i*,*?j*] **by** *simp*

```
      have y ∈ set [{0,1,2},{0,1,3},{0,2,3},{1,2,3},{0,1,2,3}]
        using ceros-of-boolean-input-set [OF dx] y-nin
        unfolding cx by simp
      hence card y ≥ 3 by auto
      thus False using card-le by simp
    qed
    then show y ∈ {{},?a,?b,?c,?d,?e,?f,?g,?h,?i,?j}
    by simp
  qed
qed

end

end

theory Bij-betw-simplicial-complex-bool-func
  imports
    Simplicial-complex
begin
```

7 Bijection between simplicial complexes and monotone Boolean functions

```
context simplicial-complex
begin

lemma ceros-of-boolean-input-in-set:
  assumes s: σ ∈ simplices
  shows ceros-of-boolean-input (vec n (λi. i ∉ σ)) = σ
  unfolding ceros-of-boolean-input-def using s unfolding simplices-def by auto

lemma
  assumes sigma: σ ∈ simplices
  and nempty: σ ≠ {}
  shows Max σ < n
proof −
   have Max σ ∈ σ using linorder-class.Max-in [OF finite-simplex [OF sigma]
nempty] .
  thus ?thesis using sigma unfolding simplices-def by auto
qed

definition bool-vec-from-simplice :: nat set => (bool vec)
  where bool-vec-from-simplice σ = vec n (λi. i ∉ σ)

lemma [simp]:
  assumes σ ∈ simplices
  shows ceros-of-boolean-input (bool-vec-from-simplice σ) = σ
  unfolding bool-vec-from-simplice-def
```

unfolding *ceros-of-boolean-input-def*
unfolding *dim-vec*
using *assms* **unfolding** *simplices-def* **by** *auto*

lemma [*simp*]:
assumes *n*: *dim-vec f* = *n*
shows *bool-vec-from-simplice* (*ceros-of-boolean-input f*) = *f*
unfolding *bool-vec-from-simplice-def*
unfolding *ceros-of-boolean-input-def*
using *n* **by** *auto*

lemma *bool-vec-from-simplice* {*0*} = *vec n* (λ*i*. *i* ∉ {*0*})
unfolding *bool-vec-from-simplice-def* **by** *auto*

lemma *bool-vec-from-simplice* {*1,2*} =
vec n (λ*i*. *i* ∉ {*1,2*})
unfolding *bool-vec-from-simplice-def* **by** *auto*

lemma *simplicial-complex-implies-true*:
assumes σ ∈ *simplicial-complex-induced-by-monotone-boolean-function n f*
shows *f* (*bool-vec-from-simplice* σ)
unfolding *bool-vec-from-simplice-def*
using *assms*
unfolding *simplicial-complex-induced-by-monotone-boolean-function-def*
unfolding *ceros-of-boolean-input-def*
apply *auto*
by (*smt* (*verit*, *best*) *dim-vec eq-vecI index-vec*)

definition *bool-vec-set-from-simplice-set* :: *nat set set* => (*bool vec*) *set*
where *bool-vec-set-from-simplice-set K* =
{σ. (*dim-vec* σ = *n*) ∧ (∃ *k*∈*K*. σ = *bool-vec-from-simplice k*)}

definition *boolean-function-from-simplicial-complex* :: *nat set set* => (*bool vec* => *bool*)
where *boolean-function-from-simplicial-complex K* =
(λ*x*. *x* ∈ (*bool-vec-set-from-simplice-set K*))

lemma *Collect* (*boolean-function-from-simplicial-complex K*) = (*bool-vec-set-from-simplice-set K*)
unfolding *boolean-function-from-simplicial-complex-def* **by** *simp*

The Boolean function induced by a simplicial complex is monotone. This result is proven in Scoville as part of the proof of Proposition 6.16. The opposite direction has been proven as *boolean-functions.monotone-bool-fun n ?f* ⟹ *simplicial-complex* (*simplicial-complex-induced-by-monotone-boolean-function n ?f*).

lemma
simplicial-complex-induces-monotone-bool-fun:
assumes *mon*: *simplicial-complex* (*K*::*nat set set*)

shows *boolean-functions.monotone-bool-fun n* (*boolean-function-from-simplicial-complex K*)
proof (*unfold boolean-functions.monotone-bool-fun-def*)
show *mono-on* (*carrier-vec n*) (*boolean-function-from-simplicial-complex K*)
proof (*intro mono-onI*)
fix *r* **and** *s*::*bool vec*
assume *r-le-s*: $r \leq s$
show *boolean-function-from-simplicial-complex K r*
$\leq$ *boolean-function-from-simplicial-complex K s*
proof (*cases boolean-function-from-simplicial-complex K r*)
case *False* **then show** *?thesis* **by** *simp*
next
case *True*
have *ce*: *ceros-of-boolean-input s* $\subseteq$ *ceros-of-boolean-input r*
using *monotone-ceros-of-boolean-input* [*OF r-le-s*] **.**
from *True* **obtain** *k* **where** *r-def*: $r = vec\ n\ (\lambda i.\ i \notin k)$
and *k*: $k \in K$
unfolding *boolean-function-from-simplicial-complex-def*
unfolding *bool-vec-set-from-simplice-set-def*
unfolding *bool-vec-from-simplice-def* **by** *auto*
have *r-in-K*: *ceros-of-boolean-input r* $\in K$
using *k mon*
unfolding *r-def*
unfolding *ceros-of-boolean-input-def*
unfolding *dim-vec*
using *simplicial-complex-def* [*of K*] **by** *fastforce*
have *boolean-function-from-simplicial-complex K s*
proof (*unfold boolean-function-from-simplicial-complex-def*
bool-vec-set-from-simplice-set-def, *rule*, *rule conjI*)
show *dim-vec s* = *n*
by (*metis less-eq-vec-def dim-vec r-def r-le-s*)
show $\exists k \in K.\ s = bool\text{-}vec\text{-}from\text{-}simplice\ k$
proof (*rule bexI* [*of - ceros-of-boolean-input s*], *unfold bool-vec-from-simplice-def*)
show *ceros-of-boolean-input s* $\in K$
using *simplicial-complex-monotone* [*OF mon r-in-K ce*] **.**
show *s* = *vec n* ($\lambda i.\ i \notin$ *ceros-of-boolean-input s*)
using *ce* **unfolding** *ceros-of-boolean-input-def*
using *r-le-s*
unfolding *less-eq-vec-def*
unfolding *r-def*
unfolding *dim-vec* **by** *auto*
qed
qed
thus *?thesis* **by** *simp*
qed
qed
qed

lemma shows (*simplicial-complex-induced-by-monotone-boolean-function n*) $\in$

boolean-functions.monotone-bool-fun-set n
→ (simplicial-complex-set::nat set set set)
proof
fix *x::bool vec ⇒ bool*
assume *x: x ∈ boolean-functions.monotone-bool-fun-set n*
show *simplicial-complex-induced-by-monotone-boolean-function n x ∈ simplicial-complex-set*
using *monotone-bool-fun-induces-simplicial-complex [of x] x*
unfolding *boolean-functions.monotone-bool-fun-set-def*
unfolding *boolean-functions.monotone-bool-fun-def simplicial-complex-set-def*
by *auto*
qed

lemma shows *boolean-function-from-simplicial-complex*
∈ (simplicial-complex-set::nat set set set)
→ boolean-functions.monotone-bool-fun-set n
proof
fix *x::nat set set* **assume** *x: x ∈ simplicial-complex-set*
show *boolean-function-from-simplicial-complex x ∈ boolean-functions.monotone-bool-fun-set n*
using *simplicial-complex-induces-monotone-bool-fun [of x]*
unfolding *boolean-functions.monotone-bool-fun-set-def*
unfolding *boolean-functions.monotone-bool-fun-def*
using *x* **unfolding** *simplicial-complex-set-def*
unfolding *mem-Collect-eq* **by** *fast*
qed

Given a Boolean function *f*, if we build its associated simplicial complex and then the associated Boolean function, we obtain *f*.

The result holds for every Boolean function *f* (the premise on *f* being monotone can be omitted).

lemma
boolean-function-from-simplicial-complex-simplicial-complex-induced-by-monotone-boolean-function:
fixes *f :: bool vec ⇒ bool*
assumes *dim: v ∈ carrier-vec n*
shows *boolean-function-from-simplicial-complex*
(simplicial-complex-induced-by-monotone-boolean-function n f) v = f v
proof (*intro iffI*)
assume *xb: f v*
show *bf: boolean-function-from-simplicial-complex*
(simplicial-complex-induced-by-monotone-boolean-function n f) v
proof −
have *f v ∧ v = bool-vec-from-simplice (ceros-of-boolean-input v)*
unfolding *ceros-of-boolean-input-def*
unfolding *bool-vec-from-simplice-def*
using *xb dim* **unfolding** *carrier-vec-def* **by** *auto*
then show *?thesis*
unfolding *simplicial-complex-induced-by-monotone-boolean-function-def*
unfolding *boolean-function-from-simplicial-complex-def*

```
    unfolding bool-vec-set-from-simplice-set-def
    unfolding mem-Collect-eq
    using dim unfolding carrier-vec-def by blast
   qed
next
  assume boolean-function-from-simplicial-complex
      (simplicial-complex-induced-by-monotone-boolean-function n f) v
  then show f v
    unfolding simplicial-complex-induced-by-monotone-boolean-function-def
    unfolding boolean-function-from-simplicial-complex-def
    unfolding bool-vec-set-from-simplice-set-def
    unfolding mem-Collect-eq
    using ‹boolean-function-from-simplicial-complex
      (simplicial-complex-induced-by-monotone-boolean-function n f) v›
      boolean-function-from-simplicial-complex-def
      simplicial-complex.bool-vec-set-from-simplice-set-def
      simplicial-complex-implies-true by fastforce
qed
```

Given a simplicial complex K, if we build its associated Boolean function, and then the associated simplicial complex, we obtain K.

```
lemma
 simplicial-complex-induced-by-monotone-boolean-function-boolean-function-from-simplicial-complex:
  fixes K :: nat set set
  assumes K: simplicial-complex K
  shows simplicial-complex-induced-by-monotone-boolean-function n
    (boolean-function-from-simplicial-complex K) = K
proof (intro equalityI)
  show simplicial-complex-induced-by-monotone-boolean-function n
    (boolean-function-from-simplicial-complex K) ⊆ K
  proof
    fix x :: nat set
    assume x: x ∈ simplicial-complex-induced-by-monotone-boolean-function
            n (boolean-function-from-simplicial-complex K)
    show x ∈ K
      using x
      unfolding boolean-function-from-simplicial-complex-def
      unfolding simplicial-complex-induced-by-monotone-boolean-function-def
      unfolding bool-vec-from-simplice-def bool-vec-set-from-simplice-set-def
      using K
      unfolding simplicial-complex-def simplices-def
      by auto (metis assms bool-vec-from-simplice-def
          ceros-of-boolean-input-in-set simplicial-complex-def)
  qed
next
  show K ⊆ simplicial-complex-induced-by-monotone-boolean-function n
          (boolean-function-from-simplicial-complex K)
 proof
   fix x :: nat set
```

```
  assume x ∈ K
  hence x: x ∈ simplices using K unfolding simplicial-complex-def by simp
  have bvs: ceros-of-boolean-input (bool-vec-from-simplice x) = x
    unfolding one-bool-def
    unfolding bool-vec-from-simplice-def
    using ceros-of-boolean-input-in-set [OF x] .
  show x ∈ simplicial-complex-induced-by-monotone-boolean-function n
        (boolean-function-from-simplicial-complex K)
    unfolding boolean-function-from-simplicial-complex-def
    unfolding simplicial-complex-induced-by-monotone-boolean-function-def
    unfolding bool-vec-from-simplice-def bool-vec-set-from-simplice-set-def
    using x bool-vec-from-simplice-def bvs
    by (metis (mono-tags, lifting) ‹x ∈ K› dim-vec mem-Collect-eq)
 qed
qed

end

end
```

Author: Julius Michaelis

```
theory MkIfex
  imports ROBDD.BDT
begin
```

8 Converting boolean functions to BDTs

The following function builds an *'a ifex* (a binary decision tree) from a boolean function and its list of variables (note that in this development we will be using boolean functions over sets of the natural numbers of the form $\{..<n\}$).

```
fun mk-ifex :: ('a :: linorder) boolfunc ⇒ 'a list ⇒ 'a ifex where
mk-ifex f [] = (if f (const False) then Trueif else Falseif) |
mk-ifex f (v#vs) = ifex-ite
                    (IF v Trueif Falseif)
                    (mk-ifex (bf-restrict v True f) vs)
                    (mk-ifex (bf-restrict v False f) vs)
```

The result of *mk-ifex* is *ifex-ordered* and *ifex-minimal*.

```
lemma mk-ifex-ro: ro-ifex (mk-ifex f vs)
  by(induction vs arbitrary: f; fastforce
      intro: order-ifex-ite-invar minimal-ifex-ite-invar
      simp del: ifex-ite.simps)
```

To prove that *mk-ifex* has correctly captured a boolean function *f*, we need know that all variables that *f* depends on were considered by *mk-ifex*. In that regard, one troublesome aspect of *boolfunc* from *ROBDD.Bool-Func* is that it is too general: Boolean functions' assignments assign a Boolean value

to any natural number, and functions are not limited to "reading" only from a finite set of variables. This for example allows for the boolean function that asks "Is the assignment true in a finite number of variables: $\lambda as.$ *finite* $\{x.\ as\ x\}$." This function does not depend on any single variable, but on the set of all of them. A definition that proved to work despite such subtleties is that a function f only depends on the variables in set x iff for any pair of assignments that agree in x (but are arbitrary otherwise), the values of f agree:

definition *reads-inside-set f x* ≡
  (∀ *assmt1 assmt2*. (∀ *p*. *p* ∈ *x* ⟶ *assmt1 p* = *assmt2 p*) ⟶ *f assmt1* = *f assmt2*)

lemma *reads-inside-set-subset*: *reads-inside-set f a* ⟹ *a* ⊆ *b* ⟹ *reads-inside-set f b*
  unfolding *reads-inside-set-def* **by** *blast*

lemma *reads-inside-set-restrict*: *reads-inside-set f s* ⟹ *reads-inside-set* (*bf-restrict i v f*) (*Set.remove i s*)
  unfolding *reads-inside-set-def bf-restrict-def* **by** *force*

lemma *collect-upd-true*: *Collect* (*x*(*y*:= *True*)) = *insert y* (*Collect x*) **by** *auto*
lemma *collect-upd-false*: *Collect* (*x*(*y*:= *False*)) = *Set.remove y* (*Collect x*) **by** *auto metis*

lemma *reads-none*: *reads-inside-set f* {} ⟹ *f* = *bf-True* ∨ *f* = *bf-False*
  unfolding *reads-inside-set-def* **by** *fast*

lemma *val-ifex-ite-subst*: ⟦*ro-ifex i*; *ro-ifex t*; *ro-ifex e*⟧ ⟹ *val-ifex* (*ifex-ite i t e*) = *bf-ite* (*val-ifex i*) (*val-ifex t*) (*val-ifex e*)
  using *val-ifex-ite* **by** *blast*

theorem
  val-ifex-mk-ifex-equal:
  reads-inside-set f (*set vs*) ⟹ *val-ifex* (*mk-ifex f vs*) *assmt* = *f assmt*
proof(*induction vs arbitrary*: *f assmt*)
  case *Nil*
  then show *?case* **using** *reads-none* **by** *auto*
next
  case (*Cons v vs*)
  have *reads-inside-set* (*bf-restrict v x f*) (*set vs*) **for** *x*
    using *reads-inside-set-restrict*[*OF Cons.prems*] *reads-inside-set-subset* **by** *fast-force*
  from *Cons.IH*[*OF this*] **show** *?case*
    unfolding *mk-ifex.simps val-ifex.simps bf-restrict-def*
    by(*subst val-ifex-ite-subst*; *simp add*: *bf-ite-def fun-upd-idem mk-ifex-ro*)
qed

end

theory *Evasive*
imports
Bij-betw-simplicial-complex-bool-func
MkIfex
begin

9 Relation between type *bool vec* ⇒ *bool* and type *'a boolfunc*

definition *vec-to-boolfunc* :: *nat* ⇒ (*bool vec* => *bool*) => (*nat boolfunc*)
where *vec-to-boolfunc n f* = (λ*i*. *f* (*vec n i*))

lemma
ris: *reads-inside-set* (λ*i*. *bool-fun-threshold-2-3* (*vec 4 i*)) (*set* [*0*,*1*,*2*,*3*])
unfolding *reads-inside-set-def*
unfolding *bool-fun-threshold-2-3-def*
unfolding *count-true-def*
unfolding *dim-vec*
unfolding *set-list-four* [*symmetric*] **by** *simp*

lemma
shows *val-ifex* (*mk-ifex* (*vec-to-boolfunc 4 bool-fun-threshold-2-3*) [*0*,*1*,*2*,*3*])
= *vec-to-boolfunc 4 bool-fun-threshold-2-3*
apply (*rule ext*)
apply (*rule val-ifex-mk-ifex-equal*)
unfolding *vec-to-boolfunc-def*
using *ris* **.**

For any Boolean function in dimension n, its ifex representation is *ifex-ordered* and *ifex-minimal*.

lemma *mk-ifex-boolean-function*:
fixes *f* :: *bool vec* => *bool*
shows *ro-ifex* (*mk-ifex* (*vec-to-boolfunc n f*) [*0*..*n*])
using *mk-ifex-ro* **by** *fast*

Any Boolean function in dimension n can be seen as an expression over the underlying set of variables.

lemma
reads-inside-set-boolean-function:
fixes *f* :: *bool vec* => *bool*
shows *reads-inside-set* (*vec-to-boolfunc n f*) {..<*n*}
unfolding *vec-to-boolfunc-def*
unfolding *reads-inside-set-def*
by (*smt* (*verit*, *best*) *dim-vec eq-vecI index-vec lessThan-iff*)

Any Boolean function of a finite dimension is equal to its ifex representation by means of *mk-ifex*.

lemma
mk-ifex-equivalence:
fixes *f* :: *bool vec* => *bool*
shows *val-ifex* (*mk-ifex* (*vec-to-boolfunc n f*) [*0..n*])
= *vec-to-boolfunc n f*
apply (*rule ext*)
apply (*rule val-ifex-mk-ifex-equal*)
using *reads-inside-set-boolean-function* [*of n f*]
unfolding *reads-inside-set-def* **by** *auto*

definition *bcount-true* :: *nat* => (*nat*=> *bool*) => *nat*
where *bcount-true n f* = (∑ *i* = *0..<n. if f i then 1 else* (*0*::*nat*))

definition *boolfunc-threshold-2-3* :: (*nat* => *bool*) => *bool*
where *boolfunc-threshold-2-3* = (λ*v. 2* ≤ *bcount-true 4 v*)

definition *proj-2* :: (*nat* => *bool*) => *bool*
where *proj-2* = (λ*v. v 2*)

definition *proj-2-n3* :: (*nat* => *bool*) => *bool*
where *proj-2-n3* = (λ*v. v 2* ∧ ¬ *v 3*)

The following definition computes the height of a *'a ifex* expression.

fun *height* :: *'a ifex* => *nat*
where *height Trueif* = *0*
| *height Falseif* = *0*
| *height* (*IF v va vb*) = *1* + *max* (*height va*) (*height vb*)

Both *mk-ifex* and *height* can be used in computations.

lemma *height* (*mk-ifex* (*boolfunc-threshold-2-3*) [*0,1,2,3*]) = *4*
by *eval*

lemma *height* (*mk-ifex* (*proj-2*) [*0,1,2,3*]) = *1*
by *eval*

lemma *mk-ifex* (*proj-2*) [*0*] = *Falseif* **by** *eval*

lemma *height* (*mk-ifex* (*proj-2*) [*0*]) = *0* **by** *eval*

lemma *mk-ifex* (*proj-2*) [*3,2,1,0*] = *IF 2 Trueif Falseif*
by *eval*

lemma *mk-ifex* (*proj-2*) [*0,1,2,3*] = *IF 2 Trueif Falseif*
by *eval*

lemma *height* (*mk-ifex* (*proj-2*) [*0,1,2,3*]) = *1* **by** *eval*

lemma *mk-ifex* (*proj-2-n3*) [*0*,*1*,*2*,*3*] = *IF 2* (*IF 3 Falseif Trueif*) *Falseif* **by** *eval*

lemma *mk-ifex* (*bf-False*::*nat boolfunc*) [*0*,*1*,*2*,*3*] = *Falseif* **by** *eval*

lemma *height* (*mk-ifex* (*bf-False*::*nat boolfunc*) [*0*,*1*,*2*,*3*]) = *0* **by** *eval*

lemma *mk-ifex* (*bf-True*::*nat boolfunc*) [*0*,*1*,*2*,*3*] = *Trueif* **by** *eval*

lemma *height* (*mk-ifex* (*bf-True*::*nat boolfunc*) [*0*,*1*,*2*,*3*]) = *0* **by** *eval*

10 Definition of *evasive* Boolean function

Now we introduce the definition of evasive Boolean function. It is based on the height of the ifex expression of the given function. The definition is inspired by the one by Scoville [3, Ex. 6.19].

definition *evasive* :: *nat* => ((*nat* => *bool*) => *bool*) => *bool*
  where *evasive n f* ≡ (*height* (*mk-ifex f* [*0*..*n*])) = *n*

corollary *evasive 4 boolfunc-threshold-2-3* **by** *eval*

lemma ¬ *evasive 4 proj-2* **by** *eval*

lemma ¬ *evasive 4 proj-2-n3* **by** *eval*

lemma ¬ *evasive 4 bf-True* **by** *eval*

lemma ¬ *evasive 4 bf-False* **by** *eval*

end

theory *ListLexorder*
imports *Main*
begin

11 Detour: Lexicographic ordering for lists

Simplicial complexes are defined as sets of sets. To conveniently run computations on them, we convert those sets to lists via *sorted-list-of-set*. This requires providing an arbitrary linear order for lists. We pick a lexicographic order.

datatype *'a* :: *linorder linorder-list* = *LinorderList 'a list*

definition *linorder-list-unwrap L* ≡ *case L of LinorderList L* ⇒ *L*

fun *less-eq-linorder-list-pre* **where**
  less-eq-linorder-list-pre (*LinorderList* []) (*LinorderList* []) = *True* |

```
  less-eq-linorder-list-pre (LinorderList []) - = True |
  less-eq-linorder-list-pre - (LinorderList []) = False |
  less-eq-linorder-list-pre (LinorderList (a # as)) (LinorderList (b # bs))
   = (if a = b then less-eq-linorder-list-pre (LinorderList as) (LinorderList bs) else
a < b)

instantiation linorder-list :: (linorder) linorder
begin
definition less-linorder-list x y ≡
            (less-eq-linorder-list-pre x y ∧ ¬ less-eq-linorder-list-pre y x)
definition less-eq-linorder-list x y ≡ less-eq-linorder-list-pre x y
instance
proof (standard; unfold less-eq-linorder-list-def less-linorder-list-def)
  fix x y z
  show less-eq-linorder-list-pre x x
  proof(induction x)
    case (LinorderList xa)
    then show ?case by(induction xa; simp)
  qed
  show less-eq-linorder-list-pre x y ⟹ less-eq-linorder-list-pre y x ⟹ x = y
    by(induction x y rule: less-eq-linorder-list-pre.induct; simp split: if-splits)
  show less-eq-linorder-list-pre x y ∨ less-eq-linorder-list-pre y x
    by(induction x y rule: less-eq-linorder-list-pre.induct; auto)
 show less-eq-linorder-list-pre x y ⟹ less-eq-linorder-list-pre y z ⟹ less-eq-linorder-list-pre
x z
  proof(induction x z arbitrary: y rule: less-eq-linorder-list-pre.induct)
    case (3 va vb)
    then show ?case
      using less-eq-linorder-list-pre.elims(2) by blast
  next
    case (4 a1 as b1 bs)
    obtain y1 ys where y: y = LinorderList (y1 # ys)
      using 4.prems(1) less-eq-linorder-list-pre.elims(2) by blast
    then show ?case proof(cases a1 = b1)
      case True
        have prems: less-eq-linorder-list-pre (LinorderList as) (LinorderList ys)
less-eq-linorder-list-pre (LinorderList ys) (LinorderList bs)
     by (metis 4.prems True y less-eq-linorder-list-pre.simps(4) not-less-iff-gr-or-eq)+
      note IH = 4.IH[OF - this]
      then show ?thesis
        using True by simp

    next
      case False
        then show ?thesis using 4.prems less-trans y by (simp  split: if-splits)
      qed
  qed simp-all
qed simp
```

end

The main product of this theory file:

definition *sorted-list-of-list-set L* ≡
map linorder-list-unwrap (*sorted-list-of-set* (*LinorderList* ‘ *L*))

lemma *set-sorted-list-of-list-set*[*simp*]:
finite L ⟹ *set* (*sorted-list-of-list-set L*) = *L*
by(*force simp add*: *sorted-list-of-list-set-def linorder-list-unwrap-def*)

end

theory *BDD*
imports
Evasive
ROBDD.Level-Collapse
ListLexorder
begin

12 Executably converting Simplicial Complexes to BDDs

We already know how to convert a simplicial complex to a boolean function, and that to a BDT. We could trivially convert convert boolean functions to BDDs the same way they are converted to BDTs, but the conversion to BDTs necessarily takes an amount of steps exponential in the number of variables (vertices). The following method avoids this exponential method.

This theory does not include a proof on the run-time complexity of the conversion.

The basic idea is that each vertex in a simplicial complex corresponds to one *True* line in the truth table of the inducing Boolean function. This is captured by the following definition, which is part of the correctness assumptions of the final theorem.

definition *bf-from-sc* :: *nat set set* => (*bool vec* ⇒ *bool*)
where *bf-from-sc K* ≡ (λ*v*. {*i*. *i* < *dim-vec v* ∧ ¬ (*vec-index v i*)} ∈ *K*)

lemma *bf-from-sc*:
assumes *sc*: *simplicial-complex.simplicial-complex n K*
shows *simplicial-complex-induced-by-monotone-boolean-function n* (*bf-from-sc K*)
= *K*
unfolding *bf-from-sc-def simplicial-complex-induced-by-monotone-boolean-function-def*
using *sc*
unfolding *simplicial-complex.simplicial-complex-def*
unfolding *simplicial-complex.simplices-def*
unfolding *ceros-of-boolean-input-def*

by *auto* (*metis ceros-of-boolean-input-def dim-vec sc*
simplicial-complex.ceros-of-boolean-input-in-set simplicial-complex.simplicial-complex-def)

definition *boolfunc-from-sc* :: *nat* => *nat set set* ⇒ *nat boolfunc*
where *boolfunc-from-sc n K* ≡ λ*p*. {*i*. *i* < *n* ∧ ¬ *p i*} ∈ *K*

The conversion proven correct in two major steps:

- Prove that we can convert the list form of simplicial complexes to boolean functions instead of the set form (*boolfunc-from-sc-list*)
- Prove that we can convert the list form of simplicial complexes to BDDs (*boolfunc-bdd-from-sc-list*)

definition *sc-threshold-2-3* ≡ {{},{*0*::*nat*},{*1*},{*2*},{*3*},{*0*,*1*},{*0*,*2*},{*0*,*3*},{*1*,*2*},{*1*,*3*},{*2*,*3*}}

Example: The truth table (as separate lemmas) for *sc-threshold-2-3*:

lemma *hlp1*: {*i*. *i* < *4* ∧ ¬ (*f*(*0* := *a0*, *1* := *a1*, *2* := *a2*, *3* := *a3*)) *i*} =
(*if a0 then* {} *else* {*0*::*nat*})
∪ (*if a1 then* {} *else* {*1*})
∪ (*if a2 then* {} *else* {*2*})
∪ (*if a3 then* {} *else* {*3*})
by *auto*

lemma *sc-threshold-2-3-ffff*:
boolfunc-from-sc 4 sc-threshold-2-3 (*a* (*0*:=*False*,*1*:=*False*,*2*:=*False*,*3*:=*False*)) =
False
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def*
by *simp* (*smt* (*z3*) *Suc-eq-numeral insert-absorb insert-commute insert-ident*
insert-not-empty numeral-2-eq-2 singleton-inject zero-neq-numeral)

lemma *sc-threshold-2-3-fffft*:
boolfunc-from-sc 4 sc-threshold-2-3 (*a*(*0*:=*False*,*1*:=*False*,*2*:=*False*,*3*:=*True*)) =
False
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def*
by *simp* (*smt* (*z3*) *Suc-eq-numeral insertI1 insert-absorb insert-commute*
insert-ident insert-not-empty numeral-2-eq-2 singleton-inject zero-neq-numeral)

lemma *sc-threshold-2-3-fftf*:
boolfunc-from-sc 4 sc-threshold-2-3 (*a*(*0*:=*False*,*1*:=*False*,*2*:=*True*,*3*:=*False*)) =
False
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def*
by *simp* (*smt* (*z3*) *insertI1 insert-iff numeral-1-eq-Suc-0*
numeral-2-eq-2 numeral-eq-iff semiring-norm(*86*) *singleton-iff zero-neq-numeral*)

lemma *sc-threshold-2-3-ftff*:
boolfunc-from-sc 4 sc-threshold-2-3 (*a*(*0*:=*False*,*1*:=*True*,*2*:=*False*,*3*:=*False*)) =
False
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def*

by *simp* (*smt* (*verit*, *ccfv-SIG*) *insert-absorb insert-iff insert-not-empty numeral-eq-iff semiring-norm*(*89*) *zero-neq-numeral*)

lemma *sc-threshold-2-3-tfff*:
boolfunc-from-sc 4 sc-threshold-2-3 (*a*(*0*:=*True*,*1*:=*False*,*2*:=*False*,*3*:=*False*)) = *False*
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def*
by *simp* (*smt* (*z3*) *eval-nat-numeral*(*3*) *insertI1 insert-commute insert-iff n-not-Suc-n numeral-1-eq-Suc-0 numeral-2-eq-2 numeral-eq-iff singletonD verit-eq-simplify*(*12*))

lemma *sc-threshold-2-3-fftt*:
boolfunc-from-sc 4 sc-threshold-2-3 (*a*(*0*:=*False*,*1*:=*False*,*2*:=*True*,*3*:=*True*)) = *True*
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def* **by** *auto*

lemma *sc-threshold-2-3-ftft*:
boolfunc-from-sc 4 sc-threshold-2-3 (*a*(*0*:=*False*,*1*:=*True*,*2*:=*False*,*3*:=*True*)) = *True*
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def* **by** *auto*

lemma *sc-threshold-2-3-fttf*:
boolfunc-from-sc 4 sc-threshold-2-3 (*a*(*0*:=*False*,*1*:=*True*,*2*:=*True*,*3*:=*False*)) = *True*
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def* **by** *auto*

lemma *sc-threshold-2-3-fttt*:
boolfunc-from-sc 4 sc-threshold-2-3 (*a*(*0*:=*False*,*1*:=*True*,*2*:=*True*,*3*:=*True*)) = *True*
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def* **by** *auto*

lemma *sc-threshold-2-3-tfft*:
boolfunc-from-sc 4 sc-threshold-2-3 (*a*(*0*:=*True*,*1*:=*False*,*2*:=*False*,*3*:=*True*)) = *True*
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def* **by** *auto*

lemma *sc-threshold-2-3-tftf*:
boolfunc-from-sc 4 sc-threshold-2-3 (*a*(*0*:=*True*,*1*:=*False*,*2*:=*True*,*3*:=*False*)) = *True*
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def* **by** *auto*

lemma *sc-threshold-2-3-tftt*:
boolfunc-from-sc 4 sc-threshold-2-3 (*a*(*0*:=*True*,*1*:=*False*,*2*:=*True*,*3*:=*True*)) = *True*
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def* **by** *auto*

lemma *sc-threshold-2-3-ttff*:
boolfunc-from-sc 4 sc-threshold-2-3 (*a*(*0*:=*True*,*1*:=*True*,*2*:=*False*,*3*:=*False*)) = *True*

unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def* **by** *auto*

lemma *sc-threshold-2-3-ttft*:
boolfunc-from-sc 4 sc-threshold-2-3 (a(0:=True,1:=True,2:=False,3:=True)) = True
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def* **by** *auto*

lemma *sc-threshold-2-3-tttf*:
boolfunc-from-sc 4 sc-threshold-2-3 (a(0:=True,1:=True,2:=True,3:=False)) = True
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def* **by** *auto*

lemma *sc-threshold-2-3-tttt*:
boolfunc-from-sc 4 sc-threshold-2-3 (a(0:=True,1:=True,2:=True,3:=True)) = True
unfolding *hlp1 boolfunc-from-sc-def sc-threshold-2-3-def* **by** *auto*

lemma *boolfunc-from-sc n UNIV = bf-True*
unfolding *boolfunc-from-sc-def* **by** *simp*

lemma *boolfunc-from-sc n {} = bf-False*
unfolding *boolfunc-from-sc-def* **by** *simp*

This may seem like an extra step, but effectively, it means: require that all the atoms outside the vertex are true, but don't care about what's in the vertex.

lemma *boolfunc-from-sc-lazy*:
simplicial-complex.simplicial-complex n K
⟹ boolfunc-from-sc n K = (λp. Pow {i. i < n ∧ ¬ p i} ⊆ K)
unfolding *simplicial-complex.simplicial-complex-def boolfunc-from-sc-def*
by *auto*

primrec *boolfunc-from-vertex-list* :: *nat list ⇒ nat list ⇒ (nat ⇒ bool) ⇒ bool*
where
boolfunc-from-vertex-list n [] = bf-True |
boolfunc-from-vertex-list n (f#fs) =
bf-and (boolfunc-from-vertex-list n fs) (if f ∈ set n then bf-True else bf-lit f)

lemma *boolfunc-from-vertex-list-Cons*:
boolfunc-from-vertex-list (a # as) lUNIV =
(λv. (boolfunc-from-vertex-list as lUNIV) (v(a:=True)))
by (*induction lUNIV*; *simp add*: *bf-lit-def*)

lemma *boolfunc-from-vertex-list-Empty*:
boolfunc-from-vertex-list [] lUNIV = Ball (set lUNIV)
by(*induction lUNIV*) (*auto simp add*: *bf-lit-def*)

lemma *boolfunc-from-vertex-list*:
set lUNIV = {..<n} ⟹ boolfunc-from-vertex-list a lUNIV = (λp. {i. i < n ∧ ¬

```
p i} ⊆ set a)
  by (induction a; fastforce
        simp add: boolfunc-from-vertex-list-Empty boolfunc-from-vertex-list-Cons)

primrec boolfunc-from-sc-list :: nat list ⇒ nat list list ⇒ (nat ⇒ bool) ⇒ bool
 where
   boolfunc-from-sc-list lUNIV [] = bf-False |
   boolfunc-from-sc-list lUNIV (l#L) =
      bf-or (boolfunc-from-sc-list lUNIV L) (boolfunc-from-vertex-list l lUNIV)

lemma boolfunc-from-sc-un:
  boolfunc-from-sc n (a∪b) = bf-or (boolfunc-from-sc n a) (boolfunc-from-sc n b)
  unfolding boolfunc-from-sc-def unfolding bf-or-def bf-ite-def by force

lemma bf-ite-const[simp]: bf-ite bf-True a b = a bf-ite bf-False a b = b
  by (simp-all)

lemma Pow-subset-Pow: Pow a ⊆ Pow b = (a ⊆ b)
  by blast

lemma boolfunc-from-sc-list-concat:
  boolfunc-from-sc-list lUNIV (a @ b) =
     bf-or (boolfunc-from-sc-list lUNIV a) (boolfunc-from-sc-list lUNIV b)
  by (induction a; auto)

lemma boolfunc-from-sc-list-existing-useless:
  a ∈ set as ⟹ boolfunc-from-sc-list l (a # as) = boolfunc-from-sc-list l as
proof(induction as)
  case (Cons a1s as) then show ?case by (cases a1s = a; simp) metis
qed simp

primrec remove :: 'a ⇒ 'a list ⇒ 'a list
  where
  remove a [] = [] |
  remove a (a1 # as) = (if a = a1 then [] else [a1]) @ remove a as

lemma set-remove[simp]: set (remove a as) = set as − {a}
  by(induction as; auto)

lemma remove-concat[simp]: remove a (a1 @ a2) = remove a a1 @ remove a a2
  by(induction a1; simp)

lemma boolfunc-from-sc-list-dedup1:
  boolfunc-from-sc-list l (a # as) = boolfunc-from-sc-list l (a # remove a as)
proof(induction as)
  case (Cons a1s as) then show ?case by(cases a1s = a; simp) metis
qed simp

lemma boolfunc-from-sc-list-reorder:
```

```
  set a = set b ⟹ boolfunc-from-sc-list l a = boolfunc-from-sc-list l b
proof(induction a arbitrary: b)
next
  case (Cons a1 a2)
  then obtain b1 b2 where  b: b = b1 @ a1 # b2 by (metis list.set-intros(1)
split-list)
  have cons-concat: ⋀a as. a # as = [a] @ as by simp
  have bb: boolfunc-from-sc-list l b =
      bf-or (boolfunc-from-vertex-list a1 l)
            (bf-or (boolfunc-from-sc-list l b1) (boolfunc-from-sc-list l b2))
    apply(subst b)
    apply(subst boolfunc-from-sc-list-concat)
    apply(subst cons-concat)
    apply(subst boolfunc-from-sc-list-concat)
    apply(auto)
    done
  have bbb: boolfunc-from-sc-list l b = boolfunc-from-sc-list l (a1 # (remove a1 (b1
@ b2)))
    unfolding bb boolfunc-from-sc-list-dedup1[symmetric]
    by (auto simp add: boolfunc-from-sc-list-concat)
  show ?case proof(cases a1 ∈ set a2)
    case True
    then show ?thesis using Cons by (metis insert-absorb list.set(2))
  next
    case False
    then have a2: set a2 = set (remove a1 (b1 @ b2))
      using Cons.prems b by fastforce
    show ?thesis using  Cons.IH[OF a2] bbb by simp
  qed
qed simp

lemma boolfunc-from-sc-list:
  set lUNIV = {..<n::nat}
    ⟹ simplicial-complex.simplicial-complex n (set ‘ set L)
      ⟹ boolfunc-from-sc-list lUNIV L = boolfunc-from-sc n (set ‘ set L)
proof −
  assume lUNIV: set lUNIV = {..<n::nat}
  assume sc: simplicial-complex.simplicial-complex n (set ‘ set L)
  define sorted where sorted ≡ sorted-wrt (λa b :: nat list. card (set b) ≤ card
(set a))

  have i: sorted L
          ⟹ simplicial-complex.simplicial-complex n (set ‘ set L)
          ⟹ boolfunc-from-sc-list lUNIV L = boolfunc-from-sc n (set ‘ set L) for
L
  proof(induction L)
    case Nil
    show ?case by (simp add: boolfunc-from-sc-def)
  next
```

```
    case (Cons a L)
    from Cons.prems(2) have p: Pow (set a) ⊆ (set ‘ set (a # L))
      unfolding simplicial-complex.simplicial-complex-def by simp
    hence pun: insert (set a) (set ‘ set L) = Pow (set a) ∪ (set ‘ set  L) by auto
    from p and Cons.prems(2)
    have sp: simplicial-complex.simplicial-complex n (Pow (set a))
       by (meson PowD Pow-subset-Pow simplicial-complex.simplicial-complex-def
subsetD)
     have bfSing: boolfunc-from-sc-list lUNIV [a] = boolfunc-from-sc n (Pow (set
a))
      unfolding boolfunc-from-sc-lazy [OF sp]
      by (simp add: Pow-subset-Pow boolfunc-from-vertex-list [OF lUNIV])
    have bflCons: boolfunc-from-sc-list lUNIV (a # L) =
      bf-or (boolfunc-from-sc-list lUNIV [a]) (boolfunc-from-sc-list lUNIV L)
      unfolding boolfunc-from-sc-list.simps(2) [of - a L] by auto
    from Cons.prems have simplicial-complex.simplicial-complex n (set ‘ set L)
      unfolding simplicial-complex.simplicial-complex-def sorted-def
      by simp (metis List.finite-set PowD card-seteq insert-image subset-insert)
    from Cons.IH[OF - this] Cons.prems(1)
    have boolfunc-from-sc-list lUNIV L = boolfunc-from-sc n (set ‘ set L)
      unfolding sorted-def by simp
    thus ?case
      apply(subst bflCons)
      apply(simp del: boolfunc-from-sc-list.simps)
      apply(subst pun)
      apply(subst boolfunc-from-sc-un)
      apply(subst bfSing)
      apply(simp)
      done
  qed
  define sort where sort ≡ rev (sort-key (λl. card (set l)) L)
  have sc: simplicial-complex.simplicial-complex n (set ‘ set sort)
    unfolding sort-def using sc by simp
  have sorted: sorted sort
   by(simp add: sorted-def sort-def sorted-wrt-rev) (metis sorted-map sorted-sort-key)
  have set: set sort = set L unfolding sort-def by simp
  from boolfunc-from-sc-list-reorder[OF set] i[OF sorted sc] set
  show ?thesis by presburger
qed

lemma boolfunc-from-sc-alt: boolfunc-from-sc n K = vec-to-boolfunc n (bf-from-sc
K)
  unfolding boolfunc-from-sc-def vec-to-boolfunc-def bf-from-sc-def
  unfolding dim-vec by(fastforce intro!: eqelem-imp-iff)

primrec bdd-from-vertex-list :: nat list ⇒ nat list ⇒ bddi ⇒ (nat × bddi) Heap
  where
    bdd-from-vertex-list n [] s = tci s |
    bdd-from-vertex-list n (f#fs) s = do {
```

```
    (f, s) ← if f ∈ set n then tci s else litci f s;
    (fs, s) ← bdd-from-vertex-list n fs s;
   andci fs f s
}
```

```
primrec bdd-from-sc-list :: nat list ⇒ nat list list ⇒ bddi ⇒ (nat × bddi) Heap
  where
   bdd-from-sc-list lUNIV [] s = fci s |
   bdd-from-sc-list lUNIV (l#L) s = do {
    (l, s) ← bdd-from-vertex-list l lUNIV s;
    (L, s) ← bdd-from-sc-list lUNIV L s;
    orci L l s
}
```

```
definition nat-list-from-vertex v ≡ sorted-list-of-set v
```

```
definition nat-list-from-sc K ≡ sorted-list-of-list-set (nat-list-from-vertex ‘ K)
```

```
definition ex-2-3 ≡ do {
  s ← emptyci;
  (ex, s) ← bdd-from-sc-list [0, 1, 2, 3] (nat-list-from-sc sc-threshold-2-3) s;
  graphifyci ''threshold-two-three'' ex s
}
```

```
lemma nat-list-from-vertex:
  assumes finite l
  shows set (nat-list-from-vertex l) = {i . i ∈ l}
  unfolding nat-list-from-vertex-def sorted-list-of-set-def
  by auto (metis assms set-sorted-list-of-set sorted-list-of-set-def)+
```

```
lemma
  finite-sorted-list-of-set:
  assumes finite L
  shows finite (sorted-list-of-set ‘ L)
    using finite-imageI [OF assms, of sorted-list-of-set] .
```

```
lemma nat-list-from-sc:
  assumes L: finite L
  and l: ∀ l∈L. finite l
  shows set ‘ set (nat-list-from-sc (L :: nat set set)) = {{i . i ∈ l} | l. l ∈  L}
  unfolding nat-list-from-sc-def
  unfolding nat-list-from-vertex-def
  unfolding set-sorted-list-of-list-set [OF finite-sorted-list-of-set [OF L]]
proof (safe)
  fix x :: nat set
  assume xl: x ∈ L
  hence fx: finite x using l by simp
```

show *exl*: $\exists\, l.\ set\ (sorted\text{-}list\text{-}of\text{-}set\ x) = \{i.\ i \in l\} \wedge l \in L$
by (*rule exI* [*of* - *x*], *auto simp add*: *xl fx*)
show $\{i.\ i \in x\} \in set\ `\ sorted\text{-}list\text{-}of\text{-}set\ `\ L$
by (*metis Collect-mem-eq exl fx image-iff set-sorted-list-of-set*)
qed

definition *ex-false* ≡ *do* {
s ← *emptyci*;
(*ex*, *s*) ← *bdd-from-sc-list* [*0*, *1*, *2*, *3*] (*nat-list-from-sc* {}) *s*;
graphifyci ''*false*'' *ex s*
}

definition *ex-true* ≡ *do* {
s ← *emptyci*;
(*ex*, *s*) ← *bdd-from-sc-list* [*0*, *1*, *2*, *3*]
(*nat-list-from-sc*
{{},{*0*},{*1*},{*2*},{*3*},
{*0,1*},{*0,2*},{*0,3*},{*1,2*},{*1,3*},{*2,3*},
{*0,1,2*},{*0,1,3*},{*0,2,3*},{*1,2,3*},{*0,1,2,3*}}) *s*;
graphifyci ''*true*'' *ex s*
}

definition *another-ex* ≡ *do* {
s ← *emptyci*;
(*ex*, *s*) ← *bdd-from-sc-list* [*0*, *1*, *2*, *3*]
(*nat-list-from-sc*
{{},{*0*},{*1*},{*2*},{*3*},
{*0,1*},{*0,2*},{*0,3*},{*1,2*},{*1,3*},{*2,3*},
{*0,1,2*},{*0,1,3*},{*0,2,3*},{*1,2,3*}}) *s*;
graphifyci ''*another-ex*'' *ex s*
}

definition *one-another-ex* ≡ *do* {
s ← *emptyci*;
(*ex*, *s*) ← *bdd-from-sc-list* [*0*, *1*, *2*, *3*]
(*nat-list-from-sc*
{{},{*0*},{*1*},{*2*},{*3*},
{*0,1*},{*0,2*},{*0,3*},{*1,2*},{*1,3*},{*2,3*},
{*0,1,2*},{*0,1,3*},{*1,2,3*}}) *s*;
graphifyci ''*one-another-ex*'' *ex s*
}

lemma *bf-ite-direct*[*simp*]: *bf-ite i bf-True bf-False* = *i* **by** *simp*

lemma *andciI*: *node-relator* (*tb*, *tc*) *rp* ⟹ *node-relator* (*eb*, *ec*) *rp* ⟹ *rq* ⊆ *rp*
⟹

```
    <bdd-relator rp s> andci tc ec s <λ(r,s'). bdd-relator (insert (bf-and tb eb,r)
rq) s'>
  by sep-auto

lemma bdd-from-vertex-list[sep-heap-rules]:
  shows <bdd-relator rp s>
    bdd-from-vertex-list n l s
  <λ(r,s'). bdd-relator (insert (boolfunc-from-vertex-list n l, r) rp) s'>
proof(induction l arbitrary: rp s)
  case Nil then show ?case by (sep-auto)
next
  case (Cons a l)
  show ?case proof(cases a ∈ set n)
    case True
    show ?thesis
      apply(simp only: bdd-from-vertex-list.simps list.map
          boolfunc-from-vertex-list.simps True if-True)
      apply(sep-auto simp only:)
       apply(rule Cons.IH)
      apply(clarsimp simp del: bf-ite-def)
      apply(sep-auto)
      done
  next
    case False
    show ?thesis
      apply(simp only: bdd-from-vertex-list.simps list.map
          boolfunc-from-vertex-list.simps False if-False)
      apply(sep-auto simp only:)
       apply(rule Cons.IH)
      apply(sep-auto simp del: bf-ite-def bf-and-def)
    done
  qed
qed

lemma boolfunc-bdd-from-sc-list:
  shows <bdd-relator rp s>
    bdd-from-sc-list lUNIV K s
  <λ(r,s'). bdd-relator (insert (boolfunc-from-sc-list lUNIV K, r) rp) s'>
proof(induction K arbitrary: rp s)
  case Nil
  then show ?case by sep-auto
next
  case (Cons a K)
  show ?case by(sep-auto heap add: Cons.IH simp del: bf-ite-def bf-or-def)
qed

lemma map-map-idI: (⋀x. x ∈ ⋃(set ' set l) ⟹ f x = x) ⟹ map (map f) l = l
  by(induct l; simp; meson map-idI)
```

definition
bdd-from-sc K n ≡ *bdd-from-sc-list* (*nat-list-from-vertex* {..<*n*}) (*nat-list-from-sc K*)

theorem *bdd-from-sc*:
assumes *simplicial-complex.simplicial-complex n* (*K* :: *nat set set*)
shows <*bdd-relator rp s*>
bdd-from-sc K n s
<λ(*r*,*s'*). *bdd-relator* (*insert* (*vec-to-boolfunc n* (*bf-from-sc K*), *r*) *rp*) *s'*>
proof −
have *fK*: *finite K*
using *simplicial-complex.finite-simplicial-complex* [*OF assms*] **.**
have *fv*: ∀ *v*∈*K*. *finite v*
using *simplicial-complex.finite-simplices* [*OF assms*] **..**
define *lUNIV* **where** *lUNIV-def*: *lUNIV* = *nat-list-from-vertex* {..<*n*}
hence *set-lUNIV*: *set lUNIV* = {..<*n*}
unfolding *nat-list-from-vertex-def*
using *sorted-list-of-set*(*1*) [*OF finite-lessThan* [*of n*]] **by** *simp*
define *Klist* **where** *Klist* ≡ (*nat-list-from-sc K*)
have *Klist-set*: *set* ‘ *set Klist* = *K*
using *nat-list-from-sc* [*OF fK fv*]
unfolding *Klist-def nat-list-from-sc-def nat-list-from-vertex-def* **by** *simp*
have *Klist-map*: *Klist* = *nat-list-from-sc K*
unfolding *Klist-def* **..**
have *sc-Klist*: *simplicial-complex.simplicial-complex n* (*set* ‘ *set Klist*)
unfolding *Klist-set* **using** *assms* **.**
show *?thesis*
apply (*insert boolfunc-bdd-from-sc-list*[*of rp s lUNIV Klist*])
unfolding *bdd-from-sc-def* **unfolding** *Klist-def* [*symmetric*]
unfolding *lUNIV-def* [*symmetric*]
unfolding *boolfunc-from-sc-list* [*OF set-lUNIV sc-Klist*]
unfolding *Klist-set*
unfolding *boolfunc-from-sc-alt* **by** *simp*
qed

code-identifier
code-module *Product-Type* ⇀ (*SML*) *IBDD*
and (*OCaml*) *IBDD* **and** (*Haskell*) *IBDD*
| **code-module** *Typerep* ⇀ (*SML*) *IBDD*
and (*OCaml*) *IBDD* **and** (*Haskell*) *IBDD*
| **code-module** *String* ⇀ (*SML*) *IBDD*
and (*OCaml*) *IBDD* **and** (*Haskell*) *IBDD*

export-code open *bdd-from-sc ex-2-3 ex-false ex-true another-ex one-another-ex*
in *Haskell* **module-name** *IBDD* **file** *BDD*

export-code *bdd-from-sc ex-2-3*
in *SML* **module-name** *IBDD* **file** *SMLBDD*

end

theory *Binary-operations*
imports *Bij-betw-simplicial-complex-bool-func*
begin

13 Binary operations over Boolean functions and simplicial complexes

In this theory some results on binary operations over Boolean functions and their relationship to operations over the induced simplicial complexes are presented. We follow the presentation by Chastain and Scoville [1, Sect. 1.1].

definition *bool-fun-or* :: *nat* ⇒ (*bool vec* ⇒ *bool*) ⇒ (*bool vec* ⇒ *bool*) ⇒ (*bool vec* ⇒ *bool*)
where (*bool-fun-or n f g*) ≡ (λ*x*. *f x* ∨ *g x*)

definition *bool-fun-and* :: *nat* ⇒ (*bool vec* ⇒ *bool*) ⇒ (*bool vec* ⇒ *bool*) ⇒ (*bool vec* ⇒ *bool*)
where (*bool-fun-and n f g*) ≡ (λ*x*. *f x* ∧ *g x*)

lemma *eq-union-or*:
simplicial-complex-induced-by-monotone-boolean-function n (*bool-fun-or n f g*)
= *simplicial-complex-induced-by-monotone-boolean-function n f*
∪ *simplicial-complex-induced-by-monotone-boolean-function n g*
(**is** *?sc n* (*?bf-or n f g*) = *?sc n f* ∪ *?sc n g*)
proof
show *?sc n f* ∪ *?sc n g* ⊆ *?sc n* (*?bf-or n f g*)
proof
fix σ :: *nat set*
assume σ ∈ (*?sc n f* ∪ *?sc n g*)
hence *sigma*: σ ∈ *?sc n f* ∨ σ ∈ *?sc n g* **by** *auto*
have *f* (*simplicial-complex.bool-vec-from-simplice n* σ)
∨ *g* (*simplicial-complex.bool-vec-from-simplice n* σ)
proof (*cases* σ ∈ *?sc n f*)
case *True*
from *simplicial-complex.simplicial-complex-implies-true* [*OF True*]
show *f* (*simplicial-complex.bool-vec-from-simplice n* σ)
∨ *g* (*simplicial-complex.bool-vec-from-simplice n* σ) **by** *fast*
next
case *False*
hence *sigmain*: σ ∈ *?sc n g* **using** *sigma* **by** *fast*
from *simplicial-complex.simplicial-complex-implies-true* [*OF sigmain*]
show *f* (*simplicial-complex.bool-vec-from-simplice n* σ)
∨ *g* (*simplicial-complex.bool-vec-from-simplice n* σ) **by** *fast*
qed
thus σ ∈ *?sc n* (*?bf-or n f g*)

```
    using simplicial-complex-induced-by-monotone-boolean-function-def
    using bool-fun-or-def sigma by auto
  qed
next
  show ?sc n (?bf-or n f g) ⊆ ?sc n f ∪ ?sc n g
  proof
    fix σ::nat set
    assume sigma: σ ∈ ?sc n (?bf-or n f g)
    hence bool-fun-or n f g (simplicial-complex.bool-vec-from-simplice n σ)
      unfolding simplicial-complex.bool-vec-from-simplice-def
      unfolding simplicial-complex-induced-by-monotone-boolean-function-def
      unfolding ceros-of-boolean-input-def
      by auto (smt (verit) dim-vec eq-vecI index-vec)+
    hence (f (simplicial-complex.bool-vec-from-simplice n σ))
          ∨ (g (simplicial-complex.bool-vec-from-simplice n σ))
      unfolding bool-fun-or-def
      by auto
    hence σ ∈ ?sc n f ∨ σ ∈ ?sc n g
      by (smt (z3) sigma bool-fun-or-def mem-Collect-eq
          simplicial-complex-induced-by-monotone-boolean-function-def)
    thus σ ∈ simplicial-complex-induced-by-monotone-boolean-function n f
        ∪ simplicial-complex-induced-by-monotone-boolean-function n g
      by auto
  qed
qed

lemma eq-inter-and:
  simplicial-complex-induced-by-monotone-boolean-function n (bool-fun-and n f g)
  = simplicial-complex-induced-by-monotone-boolean-function n f
    ∩ simplicial-complex-induced-by-monotone-boolean-function n g
  (is ?sc n (?bf-and n f g) = ?sc n f ∩ ?sc n g)
proof
  show ?sc n f ∩ ?sc n g ⊆ ?sc n (?bf-and n f g)
  proof
    fix σ :: nat set
    assume σ ∈ (?sc n f ∩ ?sc n g)
    hence sigma: σ ∈ ?sc n f ∧ σ ∈ ?sc n g by auto
    have f (simplicial-complex.bool-vec-from-simplice n σ)
         ∧ g (simplicial-complex.bool-vec-from-simplice n σ)
    proof −
      from sigma have sigmaf: σ ∈ ?sc n f and sigmag: σ ∈ ?sc n g
        by auto
      have f (simplicial-complex.bool-vec-from-simplice n σ)
        using simplicial-complex.simplicial-complex-implies-true [OF sigmaf] .
      moreover have g (simplicial-complex.bool-vec-from-simplice n σ)
        using simplicial-complex.simplicial-complex-implies-true [OF sigmag] .
      ultimately show ?thesis by fast
    qed
    thus σ ∈ ?sc n (?bf-and n f g)
```

```
    unfolding simplicial-complex-induced-by-monotone-boolean-function-def
    unfolding bool-fun-and-def
    using sigma apply auto
  by (smt (z3) Collect-cong ceros-of-boolean-input-def dim-vec index-vec mem-Collect-eq
      simplicial-complex.bool-vec-from-simplice-def
      simplicial-complex-induced-by-monotone-boolean-function-def)
 qed
next
 show ?sc n (?bf-and n f g) ⊆ ?sc n f ∩ ?sc n g
 proof
  fix σ :: nat set
  assume sigma: σ ∈ ?sc n (?bf-and n f g)
  hence bool-fun-and n f g (simplicial-complex.bool-vec-from-simplice n σ)
    unfolding simplicial-complex.bool-vec-from-simplice-def
    unfolding simplicial-complex-induced-by-monotone-boolean-function-def
    unfolding ceros-of-boolean-input-def
    by auto (smt (verit) dim-vec eq-vecI index-vec)+
  hence (f (simplicial-complex.bool-vec-from-simplice n σ))
      ∧ (g (simplicial-complex.bool-vec-from-simplice n σ))
    unfolding bool-fun-and-def
    by auto
  hence σ ∈ ?sc n f ∧ σ ∈ ?sc n g
  using bool-fun-and-def sigma simplicial-complex-induced-by-monotone-boolean-function-def
by auto
  thus σ ∈ simplicial-complex-induced-by-monotone-boolean-function n f
      ∩ simplicial-complex-induced-by-monotone-boolean-function n g
    by auto
 qed
qed

definition bool-fun-ast :: (nat × nat) ⇒ (bool vec ⇒ bool) × (bool vec ⇒ bool)
  ⇒ (bool vec × bool vec ⇒ bool)
 where (bool-fun-ast n f) ≡ (λ (x,y). (fst f x) ∧ (snd f y))

definition
 simplicial-complex-induced-by-monotone-boolean-function-ast
  :: (nat × nat) ⇒ ((bool vec × bool vec ⇒ bool)) ⇒ (nat set * nat set) set
 where simplicial-complex-induced-by-monotone-boolean-function-ast n f =
     {z. ∃ x y. dim-vec x = fst n ∧ dim-vec y = snd n ∧ f (x, y)
       ∧ ((ceros-of-boolean-input x), (ceros-of-boolean-input y)) = z}

lemma fst-es-simplice:
 a ∈ simplicial-complex-induced-by-monotone-boolean-function-ast n f
  ⟹ (∃ x y. f (x, y) ∧ (ceros-of-boolean-input x) = fst(a))
 by (smt (verit) fst-conv mem-Collect-eq
     simplicial-complex-induced-by-monotone-boolean-function-ast-def)

lemma snd-es-simplice:
 a ∈ simplicial-complex-induced-by-monotone-boolean-function-ast n f
```

$\Longrightarrow (\exists\, x\ y.\ f\ (x,\ y) \wedge (\mathit{ceros\text{-}of\text{-}boolean\text{-}input}\ y) = \mathit{snd}(a))$
by (*smt* (*verit*) *snd-conv mem-Collect-eq*
simplicial-complex-induced-by-monotone-boolean-function-ast-def)

definition *set-ast* :: (*nat set*) *set* ⇒ (*nat set*) *set* ⇒ ((*nat set*∗*nat set*) *set*)
where *set-ast A B* ≡ {*c*. ∃ *a*∈*A*. ∃ *b*∈*B*. *c* = (*a*,*b*)}

definition *set-fst* :: (*nat*∗*nat*) *set* ⇒ *nat set*
where *set-fst AB* = {*a*. ∃ *ab*∈*AB*. *a* = *fst ab*}

lemma *set-fst-simp* [*simp*]:
assumes *y* ≠ {}
shows *set-fst* (*x* × *y*) = *x*
proof
show *set-fst* (*x* × *y*) ⊆ *x*
by (*smt* (*verit*) *SigmaE mem-Collect-eq prod.sel*(*1*) *set-fst-def subsetI*)
show *x* ⊆ *set-fst* (*x* × *y*)
proof
fix *a*::*nat*
assume *a* ∈ *x*
then obtain *b* **where** *b* ∈ *y* **and** (*a*,*b*) ∈ (*x*×*y*)
using *assms* **by** *blast*
then show *a* ∈ *set-fst* (*x* × *y*)
using *set-fst-def* **by** *fastforce*
qed
qed

definition *set-snd* :: (*nat*∗*nat*) *set* ⇒ *nat set*
where *set-snd AB* = {*b*. ∃ *ab*∈*AB*. *b* = *snd*(*ab*)}

lemma
simplicial-complex-ast-implies-fst-true:
assumes γ ∈ *simplicial-complex-induced-by-monotone-boolean-function-ast nn*
(*bool-fun-ast nn f*)
shows *fst f* (*simplicial-complex.bool-vec-from-simplice* (*fst nn*) (*fst* γ))
using *assms*
unfolding *simplicial-complex.bool-vec-from-simplice-def*
unfolding *simplicial-complex-induced-by-monotone-boolean-function-ast-def*
unfolding *bool-fun-ast-def*
unfolding *ceros-of-boolean-input-def*
apply *auto*
by (*smt* (*verit*, *ccfv-threshold*) *bool-fun-ast-def case-prod-conv dim-vec index-vec vec-eq-iff*)

lemma
simplicial-complex-ast-implies-snd-true:
assumes γ ∈ *simplicial-complex-induced-by-monotone-boolean-function-ast nn*
(*bool-fun-ast nn f*)
shows *snd f* (*simplicial-complex.bool-vec-from-simplice* (*snd nn*) (*snd* γ))

using *assms*
unfolding *simplicial-complex.bool-vec-from-simplice-def*
unfolding *simplicial-complex-induced-by-monotone-boolean-function-ast-def*
unfolding *bool-fun-ast-def*
unfolding *ceros-of-boolean-input-def*
by *auto* (*smt* (*verit*, *ccfv-threshold*) *bool-fun-ast-def*
case-prod-conv dim-vec index-vec vec-eq-iff)

lemma *eq-ast*:
simplicial-complex-induced-by-monotone-boolean-function-ast (*n*, *m*) (*bool-fun-ast* (*n*, *m*) *f*)
= *set-ast* (*simplicial-complex-induced-by-monotone-boolean-function n* (*fst f*))
(*simplicial-complex-induced-by-monotone-boolean-function m* (*snd f*))
proof
show *set-ast* (*simplicial-complex-induced-by-monotone-boolean-function n* (*fst f*))
(*simplicial-complex-induced-by-monotone-boolean-function m* (*snd f*))
⊆ *simplicial-complex-induced-by-monotone-boolean-function-ast* (*n*, *m*)
(*bool-fun-ast* (*n*, *m*) *f*)
proof
fix *γ*::*nat set*∗*nat set*
assume *pert*: *γ* ∈ *set-ast* (*simplicial-complex-induced-by-monotone-boolean-function n* (*fst f*))
(*simplicial-complex-induced-by-monotone-boolean-function m* (*snd f*))
hence *f*: (*fst γ*) ∈ *simplicial-complex-induced-by-monotone-boolean-function n* (*fst f*)
unfolding *set-ast-def*
by *auto*
have *sigma*: *fst f* (*simplicial-complex.bool-vec-from-simplice n* (*fst γ*))
using *simplicial-complex.simplicial-complex-implies-true* [*OF f*] **.**
from *pert* **have** *g*: (*snd γ*) ∈ *simplicial-complex-induced-by-monotone-boolean-function m* (*snd f*)
unfolding *set-ast-def* **by** *auto*
have *tau*: (*snd f*) (*simplicial-complex.bool-vec-from-simplice m* (*snd γ*))
using *simplicial-complex.simplicial-complex-implies-true* [*OF g*] **.**
from *sigma* **and** *tau* **have** *sigtau*: *bool-fun-ast* (*n*, *m*) *f*
((*simplicial-complex.bool-vec-from-simplice n* (*fst γ*)),
(*simplicial-complex.bool-vec-from-simplice m* (*snd γ*)))
unfolding *bool-fun-ast-def*
by *auto*
from *sigtau*
show *γ* ∈ *simplicial-complex-induced-by-monotone-boolean-function-ast* (*n*, *m*)
(*bool-fun-ast* (*n*, *m*) *f*)
unfolding *simplicial-complex-induced-by-monotone-boolean-function-ast-def*
unfolding *bool-fun-ast-def*
using *sigma* **apply** *auto*
using *f g simplicial-complex-induced-by-monotone-boolean-function-def* **by** *fastforce*
qed
next

```
  show simplicial-complex-induced-by-monotone-boolean-function-ast (n, m)
     (bool-fun-ast (n, m) f)
    ⊆ set-ast (simplicial-complex-induced-by-monotone-boolean-function n (fst f))
        (simplicial-complex-induced-by-monotone-boolean-function m (snd f))
    proof
    fix γ :: nat set∗nat set
    assume pert: γ ∈ simplicial-complex-induced-by-monotone-boolean-function-ast
(n, m)
     (bool-fun-ast (n, m) f)
    have sigma: (fst γ) ∈ simplicial-complex-induced-by-monotone-boolean-function
n (fst f)
      unfolding bool-fun-ast-def
      unfolding simplicial-complex-induced-by-monotone-boolean-function-def
      unfolding simplicial-complex-induced-by-monotone-boolean-function-ast-def
      apply auto
     apply (rule exI [of - simplicial-complex.bool-vec-from-simplice n (fst γ)], safe)
      using simplicial-complex.bool-vec-from-simplice-def apply auto[1]
       apply (metis fst-conv pert simplicial-complex-ast-implies-fst-true)
      using ceros-of-boolean-input-def simplicial-complex.bool-vec-from-simplice-def
       apply fastforce
      using ceros-of-boolean-input-def pert
         simplicial-complex.bool-vec-from-simplice-def
         simplicial-complex-induced-by-monotone-boolean-function-ast-def by force
   have tau: (snd γ) ∈ simplicial-complex-induced-by-monotone-boolean-function m
(snd f)
      unfolding bool-fun-ast-def
      unfolding simplicial-complex-induced-by-monotone-boolean-function-def
      unfolding simplicial-complex-induced-by-monotone-boolean-function-ast-def
      apply auto
       apply (rule exI [of - simplicial-complex.bool-vec-from-simplice m (snd γ)],
safe)
      using simplicial-complex.bool-vec-from-simplice-def apply auto[1]
       apply (metis snd-conv pert simplicial-complex-ast-implies-snd-true)
      using ceros-of-boolean-input-def simplicial-complex.bool-vec-from-simplice-def
      apply fastforce
      using ceros-of-boolean-input-def pert
       simplicial-complex.bool-vec-from-simplice-def
       simplicial-complex-induced-by-monotone-boolean-function-ast-def by force
    from sigma and tau
    show γ ∈ set-ast
        (simplicial-complex-induced-by-monotone-boolean-function n (fst f))
        (simplicial-complex-induced-by-monotone-boolean-function m (snd f))
      using set-ast-def
      by force
  qed
qed

end
```

References

[1] E. J. Chastain and N. A. Scoville. Homology of Boolean functions and the complexity of simplicial homology. https://nanopdf.com/download/homology-of-boolean-functions-and-the-complexity-of-simplicial pdf.

[2] J. Michaelis, M. Haslbeck, P. Lammich, and L. Hupel. Algorithms for reduced ordered binary decision diagrams. *Archive of Formal Proofs*, Apr. 2016. https://isa-afp.org/entries/ROBDD.html, Formal proof development.

[3] N. A. Scoville. *Discrete Morse Theory*, volume 90 of *Student Mathematical Library*. American Mathematical Society, 2019.