

Simplicial complexes and Boolean functions*

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

February 6, 2026

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 $bool\ vec \Rightarrow bool$ and type $'a\ boolfunc$	28
10	Definition of <i>evasive</i> 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* ($\lambda i.$ *if vec-index v i then 1 else 0::nat*) {*0..<dim-vec v*}

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

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

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

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

lemma *count-true* (*vec* (5::*nat*)) ($\lambda i.$ *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* = ($\lambda v.$ *if i* \leq *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*: ($u::\text{bool vec}$) $\leq v$
 shows *count-true u* \leq *count-true v*
 unfolding *count-true-def*
 using *Groups-Big.ordered-comm-monoid-add-class.sum-mono*
 [*of* {*0..<dim-vec u*}

```

    ( $\lambda i.$  if vec-index u i then 1 else 0)
    ( $\lambda i.$  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: ( $u::\text{bool vec}$ )  $\leq v$ 
shows bool-fun-threshold n u  $\leq$  bool-fun-threshold n v
unfolding bool-fun-threshold-def
using monotone-count-true [OF ulev] by simp

```

```

lemma
assumes ( $u::\text{bool vec}$ )  $\leq v$ 
and  $n < \text{dim-vec } u$ 
shows bool-fun-threshold n u  $\leq$  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:  $\text{Pow } \{a\} = \{\{\}, \{a\}\}$  by auto

```

```

lemma Pow-pair:  $\text{Pow } \{a,b\} = \{\{\}, \{a\}, \{b\}, \{a,b\}\}$  by auto

```

```

locale simplicial-complex
  = fixes  $n::\text{nat}$ 
begin

```

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

```

definition simplices ::  $\text{nat set set}$ 

```

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 $\sigma \in \textit{simplices}$
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* ≡ ($\forall \sigma \in K. (\sigma \in \textit{simplices}) \wedge (\textit{Pow} \sigma) \subseteq K$)

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 $v \in K$
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 $K = \{\} \vee \{\} \in K$ **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*: $s \in K$ **and** *rs*: $r \subseteq s$
shows $r \in K$
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 \neg *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* = $\{x. x < \text{dim-vec } v \wedge \text{vec-index } v \ x = \text{False}\}$

lemma

ceros-of-boolean-input-l-dim:

assumes *a*: $a \in \text{ceros-of-boolean-input } v$

shows $a < \text{dim-vec } v$

using *a* **unfolding** *ceros-of-boolean-input-def* **by** *simp*

lemma *ceros-of-boolean-input* *v* = $\{x. x < \text{dim-vec } v \wedge \neg \text{vec-index } v \ x\}$

unfolding *ceros-of-boolean-input-def* **by** *simp*

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

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

We introduce here instantiations of the *typbool* type for the type classes *classzero* and *classone* that will simplify notation at some points:

instantiation $\text{bool} :: \{zero, one\}$
begin

definition
zero-bool-def: $0 == \text{False}$

definition
one-bool-def: $1 == \text{True}$

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
 $:: \text{nat} \Rightarrow (\text{bool vec} \Rightarrow \text{bool}) \Rightarrow \text{nat set set}$
where *simplicial-complex-induced-by-monotone-boolean-function* $n \ f =$
 $\{y. \exists x. \text{dim-vec } x = n \wedge f \ x \wedge \text{ceros-of-boolean-input } x = y\}$

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 ( $\lambda i. \text{if } i = 0 \text{ then False else True}$ ))
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 ( $\lambda x. \text{True}$ )])
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 ( $\lambda i. \text{if } i \in ?A \text{ then False else True}$ )],
  intro conjI)
show dim-vec (vec 4 ( $\lambda i. \text{if } i \in \{x\} \text{ then False else True}$ )) = 4 by simp
show bool-fun-threshold-2-3 (vec 4 ( $\lambda i. \text{if } i \in ?A \text{ then False else True}$ ))
  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 (λi. 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 (λi. if i ∈ ?A then False else True)],
  intro conjI)
  show dim-vec (vec 4 (λi. if i ∈ {x, y} then False else True)) = 4 by simp
  show bool-fun-threshold-2-3 (vec 4 (λi. 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 (λi. 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 $\text{card } ?\text{true} + \text{card } ?\text{false} = \text{card } (?\text{true} \cup ?\text{false}) + \text{card } (?\text{true} \cap ?\text{false})$
using *card-Un-Int* [*OF finite-True* [*of a*] *finite-False* [*of a*]] .
also have $\dots = \text{card } \{x. x < \text{dim-vec } a\}$
unfolding *UNIV-union UNIV-disjoint* **by** *simp*
finally show *?thesis* **by** *simp*
qed

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

corollary
card-ceros-of-boolean-input:
shows $\text{card } (\text{ceros-of-boolean-input } a) \leq \text{dim-vec } a$
using *card-complementary* [*of a*] **by** *simp*

lemma
vec-fun:
assumes $v \in \text{carrier-vec } n$
shows $\exists f. v = \text{vec } n \ f$ **using** *assms* **unfolding** *carrier-vec-def* **by** *fastforce*

corollary
assumes $\text{dim-vec } v = n$
shows $\exists f. v = \text{vec } n \ f$
using *carrier-vecI* [*OF assms*] **unfolding** *carrier-vec-def* **by** *fastforce*

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

lemma
card-boolean-function:
assumes $d: v \in \text{carrier-vec } n$
shows $\text{card } \{x. x < n \ \wedge \ v \ \$ \ x = \text{True}\} = (\sum i = 0..<n. \text{if } v \ \$ \ i \ \text{then } 1 \ \text{else } 0)$
(*0::nat*)
using *d* **proof** (*induction n arbitrary: v rule: nat-less-induct*)
case (*1 n*)
assume *hyp*: $\forall m < n. \forall x. x \in \text{carrier-vec } m \longrightarrow$
 $\text{card } \{xa. xa < m \ \wedge \ x \ \$ \ xa = \text{True}\} = (\sum i = 0..<m. \text{if } x \ \$ \ i \ \text{then } 1 \ \text{else } 0)$
and $d: v \in \text{carrier-vec } n$
show $\text{card } \{x. x < n \ \wedge \ v \ \$ \ x = \text{True}\} = (\sum i = 0..<n. \text{if } v \ \$ \ i \ \text{then } 1 \ \text{else } 0)$
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..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: disjoint {x. x < m ∧ vec (Suc m) f $ x = True} {x. x = m ∧ (vec
(Suc m) f) $ x = True}
      using disjoint-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..by simp
    also have ... = (∑ i = 0..using vec-l-eq [of - m] True by simp
    finally have rhs: (∑ i = 0..show card {x. x < Suc m ∧ vec (Suc m) f $ x = True} =
      (∑ i = 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: $\text{disjnt } \{x. x < m \wedge \text{vec } (\text{Suc } m) f \$ x = \text{True}\} \{x. x = m \wedge (\text{vec } (\text{Suc } m) f) \$ x = \text{True}\}$
using disjnt-iff by blast
have card $\{x. x < \text{Suc } m \wedge \text{vec } (\text{Suc } m) f \$ x = \text{True}\}$
 $= \text{card } \{x. x < m \wedge (\text{vec } (\text{Suc } m) f) \$ x = \text{True}\} + \text{card } \{x. x = m \wedge (\text{vec } (\text{Suc } m) f) \$ x = \text{True}\}$
unfolding one
by $(\text{rule card-Un-disjnt } [\text{OF } - - \text{two}], \text{simp-all})$
also have $\dots = \text{card } \{x. x < m \wedge (\text{vec } m f) \$ x = \text{True}\} + 0$
proof -
have one: $\{x. x < m \wedge \text{vec } (\text{Suc } m) f \$ x = \text{True}\} = \{x. x < m \wedge \text{vec } m f \$ x = \text{True}\}$
using vec-l-eq [of - m] by auto
have eq: $\{x. x = m \wedge \text{vec } (\text{Suc } m) f \$ x = \text{True}\} = \{\}$ **using False by auto**
hence two: $\text{card } \{x. x = m \wedge \text{vec } (\text{Suc } m) f \$ x = \text{True}\} = 0$ **by simp**
show ?thesis using one two by simp
qed
finally have lhs: $\text{card } \{x. x < \text{Suc } m \wedge \text{vec } (\text{Suc } m) f \$ x = \text{True}\} = \text{card } \{x. x < m \wedge \text{vec } m f \$ x = \text{True}\} + 0$
have $(\sum i = 0..<\text{Suc } m. \text{if } \text{vec } (\text{Suc } m) f \$ i \text{ then } 1 \text{ else } 0) =$
 $(\sum i = 0..<m. \text{if } \text{vec } (\text{Suc } m) f \$ i \text{ then } 1 \text{ else } 0) + (\text{if } \text{vec } (\text{Suc } m) f \$ m \text{ then } 1 \text{ else } 0)$
by simp
also have $\dots = (\sum i = 0..<m. \text{if } \text{vec } m f \$ i \text{ then } 1 \text{ else } 0)$
using vec-l-eq [of - m] False by simp
finally have rhs: $(\sum i = 0..<\text{Suc } m. \text{if } \text{vec } (\text{Suc } m) f \$ i \text{ then } 1 \text{ else } 0) =$
 $(\sum i = 0..<m. \text{if } \text{vec } m f \$ i \text{ then } 1 \text{ else } 0)$
show $\text{card } \{x. x < \text{Suc } m \wedge \text{vec } (\text{Suc } m) f \$ x = \text{True}\} =$
 $(\sum i = 0..<\text{Suc } m. \text{if } \text{vec } (\text{Suc } m) f \$ i \text{ then } 1 \text{ else } 0)$
unfolding lhs rhs using hyp Suc by simp
qed
qed
qed

lemma card-ceros-count-UNIV:

shows $\text{card } (\text{ceros-of-boolean-input } a) + \text{count-true } ((a::\text{bool vec})) = \text{dim-vec } a$
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 $dx: \text{dim-vec } x = 2$

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

lemma *tt-2*: **assumes** $dx: \text{dim-vec } x = 2$
and $be: x \$ 0 = \text{True} \wedge x \$ 1 = \text{True}$
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** $dx: \text{dim-vec } x = 2$
and $be: x \$ 0 = \text{True} \wedge x \$ 1 = \text{False}$
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** $dx: \text{dim-vec } x = 2$
and $be: x \$ 0 = \text{False} \wedge x \$ 1 = \text{True}$
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** $dx: \text{dim-vec } x = 2$
and $be: x \$ 0 = \text{False} \wedge x \$ 1 = \text{False}$
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 $dx: \text{dim-vec } x = 2$
shows *ceros-of-boolean-input* $x \in \{\{\}, \{0\}, \{1\}, \{0,1\}\}$
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** $n = 0 \vee n = 1 \vee n = (2::\text{nat})$
using n **by** *linarith*

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

lemma *fff-3*: **assumes** $dx: \text{dim-vec } x = 3$

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

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

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

lemma

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

Vectors of dimension 4.

lemma $less\text{-}4\text{-}cases$:

assumes n : $n < 4$
shows $n = 0 \vee n = 1 \vee n = 2 \vee n = (3::nat)$
using n **by** $linarith$

lemma

$dim\text{-}vec\text{-}4\text{-}cases$:

assumes dx : $dim\text{-}vec\ x = 4$
shows $(x\ \$\ 0 = x\ \$\ 1 = x\ \$\ 2 = x\ \$\ 3 = False) \vee (x\ \$\ 0 = x\ \$\ 1 = x\ \$\ 2 = False \wedge x\ \$\ 3 = True) \vee (x\ \$\ 0 = x\ \$\ 1 = x\ \$\ 3 = False \wedge x\ \$\ 2 = True) \vee (x\ \$\ 0 = x\ \$\ 1 = False \wedge x\ \$\ 2 = x\ \$\ 3 = True) \vee (x\ \$\ 0 = x\ \$\ 2 = x\ \$\ 3 = False \wedge x\ \$\ 1 = True) \vee (x\ \$\ 0 = x\ \$\ 2 = False \wedge x\ \$\ 1 = x\ \$\ 3 = True) \vee (x\ \$\ 0 = x\ \$\ 3 = False \wedge x\ \$\ 1 = x\ \$\ 2 = True) \vee (x\ \$\ 0 = False \wedge x\ \$\ 1 = x\ \$\ 2 = x\ \$\ 3 = True) \vee (x\ \$\ 0 = True \wedge x\ \$\ 1 = x\ \$\ 2 = x\ \$\ 3 = False) \vee (x\ \$\ 0 = x\ \$\ 3 = True \wedge x\ \$\ 1 = x\ \$\ 2 = False) \vee (x\ \$\ 0 = x\ \$\ 2 = True \wedge x\ \$\ 1 = x\ \$\ 3 = False) \vee (x\ \$\ 0 = x\ \$\ 2 = x\ \$\ 3 = True \wedge x\ \$\ 1 = False) \vee (x\ \$\ 0 = x\ \$\ 1 = True \wedge x\ \$\ 2 = x\ \$\ 3 = False) \vee (x\ \$\ 0 = x\ \$\ 1 = x\ \$\ 3 = True \wedge x\ \$\ 2 = False) \vee (x\ \$\ 0 = x\ \$\ 1 = x\ \$\ 2 = True \wedge x\ \$\ 3 = False) \vee (x\ \$\ 0 = x\ \$\ 1 = x\ \$\ 2 = x\ \$\ 3 = True)$
by $blast$

lemma $fff\text{-}4$: **assumes** dx : $dim\text{-}vec\ x = 4$

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

lemma $fft\text{-}4$: **assumes** dx : $dim\text{-}vec\ x = 4$

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

lemma $ftf\text{-}4$: **assumes** dx : $dim\text{-}vec\ x = 4$

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 $tfff-4$: **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 $tftt-4$: **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 $ttft-4$: **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] fttt-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  $y \in \text{set } [\{0,1,2\}, \{0,1,3\}, \{0,2,3\}, \{1,2,3\}, \{0,1,2,3\}]$ 
      using ceros-of-boolean-input-set [OF dx] y-nin
      unfolding cx by simp
      hence  $\text{card } y \geq 3$  by auto
      thus False using card-le by simp
  qed
  then show  $y \in \{\{\}, ?a, ?b, ?c, ?d, ?e, ?f, ?g, ?h, ?i, ?j\}$ 
    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:  $\sigma \in \text{simplices}$ 
  shows ceros-of-boolean-input ( $\text{vec } n \ (\lambda i. i \notin \sigma)$ ) =  $\sigma$ 
  unfolding ceros-of-boolean-input-def using s unfolding simplices-def by auto

```

```

lemma
  assumes sigma:  $\sigma \in \text{simplices}$ 
  and notEmpty:  $\sigma \neq \{\}$ 
  shows  $\text{Max } \sigma < n$ 
proof -
  have  $\text{Max } \sigma \in \sigma$  using linorder-class.Max-in [OF finite-simplex [OF sigma]
notEmpty].
  thus thesis using sigma unfolding simplices-def by auto
qed

```

```

definition bool-vec-from-simplice ::  $\text{nat set} \Rightarrow (\text{bool vec})$ 
  where bool-vec-from-simplice  $\sigma = \text{vec } n \ (\lambda i. i \notin \sigma)$ 

```

```

lemma [simp]:
  assumes  $\sigma \in \text{simplices}$ 
  shows ceros-of-boolean-input (bool-vec-from-simplice  $\sigma$ ) =  $\sigma$ 
  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* ($\lambda i. i \notin \{0\}$)
unfolding *bool-vec-from-simplice-def* **by** *auto*

lemma *bool-vec-from-simplice* {*1,2*} =
vec *n* ($\lambda i. i \notin \{1,2\}$)
unfolding *bool-vec-from-simplice-def* **by** *auto*

lemma *simplicial-complex-implies-true*:
assumes $\sigma \in$ *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* =
 $\{\sigma. (\text{dim-vec } \sigma = n) \wedge (\exists k \in K. \sigma = \text{bool-vec-from-simplice } k)\}$

definition *boolean-function-from-simplicial-complex* :: *nat* *set* *set* => (*bool* *vec* =>
bool)
where *boolean-function-from-simplicial-complex* *K* =
 $(\lambda x. x \in (\text{bool-vec-set-from-simplice-set } K))$

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* \implies *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 ≤ s
  show boolean-function-from-simplicial-complex K r
    ≤ 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 ⊆ 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 (λi. i ∉ k)
    and k: k ∈ 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 ∈ 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 ∃ k∈K. s = bool-vec-from-simplice k
  proof (rule bexI [of- ceros-of-boolean-input s], unfold bool-vec-from-simplice-def)
    show ceros-of-boolean-input s ∈ K
      using simplicial-complex-monotone [OF mon r-in-K ce] .
    show s = vec n (λi. i ∉ 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*) ∈

```

      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  $x \in K$ 
hence  $x: x \in \text{simplices}$  using  $K$  unfolding simplicial-complex-def by simp
have  $bvs: \text{ceros-of-boolean-input} (\text{bool-vec-from-simplice } x) = x$ 
  unfolding one-bool-def
  unfolding bool-vec-from-simplice-def
  using ceros-of-boolean-input-in-set [ $OF\ x$ ] .
show  $x \in \text{simplicial-complex-induced-by-monotone-boolean-function } n$ 
  (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)  $\langle x \in K \rangle$  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  $\Rightarrow$  'a list  $\Rightarrow$  'a ifex where
mk-ifex  $f\ [] = (\text{if } f (\text{const } \text{False}) \text{ then } \text{Trueif} \text{ else } \text{Falseif}) |$ 
mk-ifex  $f\ (v\#\text{vs}) = \text{ifex-ite}$ 
  (IF  $v\ \text{Trueif}\ \text{Falseif}$ )
  (mk-ifex (bf-restrict  $v\ \text{True}\ f$ )  $\text{vs}$ )
  (mk-ifex (bf-restrict  $v\ \text{False}\ f$ )  $\text{vs}$ )

```

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

```

lemma mk-ifex-ro: ro-ifex (mk-ifex  $f\ \text{vs}$ )
  by (induction  $\text{vs}$  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: $\lambda as. \text{finite } \{x. as\ x\}$.” 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* $f\ x \equiv$
 $(\forall \text{assmt1 assmt2}. (\forall p. p \in x \longrightarrow \text{assmt1 } p = \text{assmt2 } p) \longrightarrow f \text{ assmt1} = f \text{ assmt2})$

lemma *reads-inside-set-subset*: $\text{reads-inside-set } f\ a \implies a \subseteq b \implies \text{reads-inside-set } f\ b$

unfolding *reads-inside-set-def* **by** *blast*

lemma *reads-inside-set-restrict*: $\text{reads-inside-set } f\ s \implies \text{reads-inside-set } (\text{bf-restrict } i\ v\ f)$ (*Set.remove* $i\ s$)

unfolding *reads-inside-set-def* *bf-restrict-def* **by** *force*

lemma *collect-upd-true*: $\text{Collect } (x(y:= \text{True})) = \text{insert } y\ (\text{Collect } x)$ **by** *auto*

lemma *collect-upd-false*: $\text{Collect } (x(y:= \text{False})) = \text{Set.remove } y\ (\text{Collect } x)$ **by** *auto metis*

lemma *reads-none*: $\text{reads-inside-set } f\ \{\} \implies f = \text{bf-True} \vee f = \text{bf-False}$

unfolding *reads-inside-set-def* **by** *fast*

lemma *val-ifex-ite-subst*: $\llbracket \text{ro-ifex } i; \text{ro-ifex } t; \text{ro-ifex } e \rrbracket \implies \text{val-ifex } (\text{ifex-ite } i\ t\ e) = \text{bf-ite } (\text{val-ifex } i)\ (\text{val-ifex } t)\ (\text{val-ifex } e)$

using *val-ifex-ite* **by** *blast*

theorem

val-ifex-mk-ifex-equal:

$\text{reads-inside-set } f\ (\text{set } vs) \implies \text{val-ifex } (\text{mk-ifex } f\ vs)\ \text{assmt} = f\ \text{assmt}$

proof(*induction* *vs* *arbitrary*: $f\ \text{assmt}$)

case *Nil*

then show *?case* **using** *reads-none* **by** *auto*

next

case (*Cons* $v\ vs$)

have $\text{reads-inside-set } (\text{bf-restrict } v\ x\ f)\ (\text{set } vs)$ **for** x

using *reads-inside-set-restrict*[*OF* *Cons.prem*s] *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 $bool\ vec \Rightarrow bool$ and type $'a\ boolfunc$

```

definition vec-to-boolfunc ::  $nat \Rightarrow (bool\ vec \Rightarrow bool) \Rightarrow (nat\ boolfunc)$ 
  where vec-to-boolfunc  $n\ f = (\lambda i. f\ (vec\ n\ i))$ 

```

```

lemma
  ris: reads-inside-set  $(\lambda i. bool-fun-threshold-2-3\ (vec\ 4\ i))\ (set\ [0,1,2,3])$ 
  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  $f :: bool\ vec \Rightarrow bool$ 
  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  $f :: bool\ vec \Rightarrow bool$ 
  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 $f :: \text{bool vec} \Rightarrow \text{bool}$

shows $\text{val-ifex } (\text{mk-ifex } (\text{vec-to-boolfunc } n f) [0..n])$
 $= \text{vec-to-boolfunc } n f$

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 $\text{bcount-true} :: \text{nat} \Rightarrow (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{nat}$

where $\text{bcount-true } n f = (\sum i = 0..<n. \text{if } f i \text{ then } 1 \text{ else } (0::\text{nat}))$

definition $\text{boolfunc-threshold-2-3} :: (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{bool}$

where $\text{boolfunc-threshold-2-3} = (\lambda v. 2 \leq \text{bcount-true } 4 v)$

definition $\text{proj-2} :: (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{bool}$

where $\text{proj-2} = (\lambda v. v 2)$

definition $\text{proj-2-n3} :: (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{bool}$

where $\text{proj-2-n3} = (\lambda v. v 2 \wedge \neg v 3)$

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

fun $\text{height} :: 'a \text{ ifex} \Rightarrow \text{nat}$

where $\text{height Trueif} = 0$

| $\text{height Falseif} = 0$

| $\text{height (IF } v \text{ va } vb) = 1 + \max (\text{height } va) (\text{height } vb)$

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

lemma $\text{height } (\text{mk-ifex } (\text{boolfunc-threshold-2-3}) [0,1,2,3]) = 4$

by *eval*

lemma $\text{height } (\text{mk-ifex } (\text{proj-2}) [0,1,2,3]) = 1$

by *eval*

lemma $\text{mk-ifex } (\text{proj-2}) [0] = \text{Falseif}$ **by** *eval*

lemma $\text{height } (\text{mk-ifex } (\text{proj-2}) [0]) = 0$ **by** *eval*

lemma $\text{mk-ifex } (\text{proj-2}) [3,2,1,0] = \text{IF } 2 \text{ Trueif Falseif}$

by *eval*

lemma $\text{mk-ifex } (\text{proj-2}) [0,1,2,3] = \text{IF } 2 \text{ Trueif Falseif}$

by *eval*

lemma $\text{height } (\text{mk-ifex } (\text{proj-2}) [0,1,2,3]) = 1$ **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 \neg *evasive 4 proj-2* **by eval**

lemma \neg *evasive 4 proj-2-n3* **by eval**

lemma \neg *evasive 4 bf-True* **by eval**

lemma \neg *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 (∃ va vb)
    then show ?case
      using less-eq-linorder-list-pre.elims(2) by blast
  next
    case (∗ a1 as b1 bs)
    obtain y1 ys where y: y = LinorderList (y1 # ys)
      using 4.prem1 less-eq-linorder-list-pre.elims(2) by blast
    then show ?case proof(cases a1 = b1)
      case True
        have prem1: less-eq-linorder-list-pre (LinorderList as) (LinorderList ys)
less-eq-linorder-list-pre (LinorderList ys) (LinorderList bs)
        by (metis 4.prem1 True y less-eq-linorder-list-pre.simp1(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.prem1 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* $L \equiv$
map linorder-list-unwrap (sorted-list-of-set (LinorderList ‘ L))

lemma *set-sorted-list-of-list-set[simp]*:
finite L \implies 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 \implies (bool vec \implies bool)*
where *bf-from-sc K \equiv ($\lambda v. \{i. i < dim-vec v \wedge \neg (vec-index v i)\} \in K$)*

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* $\equiv \lambda p. \{i. i < n \wedge \neg p\ i\} \in K$

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* $\equiv \{\{\}, \{0::nat\}, \{1\}, \{2\}, \{3\}, \{0,1\}, \{0,2\}, \{0,3\}, \{1,2\}, \{1,3\}, \{2,3\}\}$

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

lemma *hlp1*: $\{i. i < 4 \wedge \neg (f(0 := a0, 1 := a1, 2 := a2, 3 := a3))\ i\} =$
(if a0 then {} else {0::nat})
 \cup *(if a1 then {} else {1})*
 \cup *(if a2 then {} else {2})*
 \cup *(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-ffft*:
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-tfff*:

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
 $\implies \text{boolfunc-from-sc } n \ K = (\lambda p. \text{Pow } \{i. i < n \wedge \neg p \ i\} \subseteq K)$

unfolding *simplicial-complex.simplicial-complex-def boolfunc-from-sc-def*
by *auto*

primrec *boolfunc-from-vertex-list :: nat list \Rightarrow nat list \Rightarrow (nat \Rightarrow bool) \Rightarrow 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 \in set n then bf-True else bf-lit f)

lemma *boolfunc-from-vertex-list-Cons*:

boolfunc-from-vertex-list (a # as) lUNIV =
 $(\lambda v. (\text{boolfunc-from-vertex-list } as \ lUNIV) (v(a:=True)))$

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} $\implies \text{boolfunc-from-vertex-list } a \ lUNIV = (\lambda p. \{i. i < n \wedge \neg$

$p\ i\} \subseteq \text{set } a)$
by (*induction a; fastforce*
simp add: boolfunc-from-vertex-list-Empty boolfunc-from-vertex-list-Cons)

primrec *boolfunc-from-sc-list* :: $\text{nat list} \Rightarrow \text{nat list list} \Rightarrow (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{bool}$
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]:* $\text{bf-ite } \text{bf-True } a\ b = a\ \text{bf-ite } \text{bf-False } a\ b = b$
by (*simp-all*)

lemma *Pow-subset-Pow:* $\text{Pow } a \subseteq \text{Pow } b = (a \subseteq b)$
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:*
 $a \in \text{set } as \Longrightarrow \text{boolfunc-from-sc-list } l\ (a \# as) = \text{boolfunc-from-sc-list } l\ as$
proof(*induction as*)
case (*Cons a1s as*) **then show** *?case* **by** (*cases a1s = a; simp*) *metis*
qed *simp*

primrec *remove* :: $'a \Rightarrow 'a\ \text{list} \Rightarrow 'a\ \text{list}$
where
remove a [] = [] |
remove a (a1 # as) = (if a = a1 then [] else [a1]) @ remove a as

lemma *set-remove[simp]:* $\text{set } (\text{remove } a\ as) = \text{set } as - \{a\}$
by(*induction as; auto*)

lemma *remove-concat[simp]:* $\text{remove } a\ (a1 @ a2) = \text{remove } a\ a1 @ \text{remove } a\ a2$
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  $\implies$  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:  $\bigwedge a$  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  $\in$  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.prem 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}
   $\implies$  simplicial-complex.simplicial-complex n (set ' set L)
   $\implies$  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  $\equiv$  sorted-wrt ( $\lambda a$  b :: nat list. card (set b)  $\leq$  card
(set a))

```

```

  have i: sorted L
     $\implies$  simplicial-complex.simplicial-complex n (set ' set L)
     $\implies$  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.prem(2) have p: Pow (set a)  $\subseteq$  (set ' set (a # L))
  unfolding simplicial-complex.simplicial-complex-def by simp
hence pun: insert (set a) (set ' set L) = Pow (set a)  $\cup$  (set ' set L) by auto
from p and Cons.prem(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.prem 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.prem(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  $\equiv$  rev (sort-key ( $\lambda l$ . 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 \Rightarrow nat list \Rightarrow bddi \Rightarrow (nat \times 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* $v \equiv$ *sorted-list-of-set* v

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

definition $ex\text{-false} \equiv \text{do } \{$
 $s \leftarrow \text{emptyci};$
 $(ex, s) \leftarrow \text{bdd-from-sc-list } [0, 1, 2, 3] (\text{nat-list-from-sc } \{\}) s;$
 $\text{graphifyci "false" } ex\ s$
 $\}$

definition $ex\text{-true} \equiv \text{do } \{$
 $s \leftarrow \text{emptyci};$
 $(ex, s) \leftarrow \text{bdd-from-sc-list } [0, 1, 2, 3]$
 $(\text{nat-list-from-sc}$
 $\{\{\}, \{0\}, \{1\}, \{2\}, \{3\},$
 $\{0, 1\}, \{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 3\}, \{2, 3\},$
 $\{0, 1, 2\}, \{0, 1, 3\}, \{0, 2, 3\}, \{1, 2, 3\}, \{0, 1, 2, 3\}\}) s;$
 $\text{graphifyci "true" } ex\ s$
 $\}$

definition $another\text{-ex} \equiv \text{do } \{$
 $s \leftarrow \text{emptyci};$
 $(ex, s) \leftarrow \text{bdd-from-sc-list } [0, 1, 2, 3]$
 $(\text{nat-list-from-sc}$
 $\{\{\}, \{0\}, \{1\}, \{2\}, \{3\},$
 $\{0, 1\}, \{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 3\}, \{2, 3\},$
 $\{0, 1, 2\}, \{0, 1, 3\}, \{0, 2, 3\}, \{1, 2, 3\}\}) s;$
 $\text{graphifyci "another-ex" } ex\ s$
 $\}$

definition $one\text{-another-ex} \equiv \text{do } \{$
 $s \leftarrow \text{emptyci};$
 $(ex, s) \leftarrow \text{bdd-from-sc-list } [0, 1, 2, 3]$
 $(\text{nat-list-from-sc}$
 $\{\{\}, \{0\}, \{1\}, \{2\}, \{3\},$
 $\{0, 1\}, \{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 3\}, \{2, 3\},$
 $\{0, 1, 2\}, \{0, 1, 3\}, \{1, 2, 3\}\}) s;$
 $\text{graphifyci "one-another-ex" } ex\ s$
 $\}$

lemma $bf\text{-ite-direct}[simp]: bf\text{-ite } i\ bf\text{-True } bf\text{-False} = i\ \text{by } simp$

lemma $andciI: \text{node-relator } (tb, tc)\ rp \implies \text{node-relator } (eb, ec)\ rp \implies rq \subseteq rp$
 \implies

$\langle \text{bdd-relator } rp \ s \rangle \text{ andci } tc \ ec \ s \ \langle \lambda(r,s'). \text{ bdd-relator } (\text{insert } (\text{bf-and } tb \ eb, r) \text{ } rq) \ s' \rangle$

by *sep-auto*

lemma *bdd-from-vertex-list[sep-heap-rules]*:

shows $\langle \text{bdd-relator } rp \ s \rangle$

$\text{bdd-from-vertex-list } n \ l \ s$

$\langle \lambda(r,s'). \text{ bdd-relator } (\text{insert } (\text{boolfunc-from-vertex-list } n \ l, r) \ rp) \ s' \rangle$

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 $\langle \text{bdd-relator } rp \ s \rangle$

$\text{bdd-from-sc-list } LUNIV \ K \ s$

$\langle \lambda(r,s'). \text{ bdd-relator } (\text{insert } (\text{boolfunc-from-sc-list } LUNIV \ K, r) \ rp) \ s' \rangle$

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*: $(\bigwedge x. x \in \bigcup (\text{set } ' \text{ set } l) \implies f \ x = x) \implies \text{map } (\text{map } f) \ l = l$

by(*induct l; simp; meson map-idI*)

definition

bdd-from-sc $K\ n \equiv \text{bdd-from-sc-list } (\text{nat-list-from-vertex } \{..<n\}) (\text{nat-list-from-sc } K)$

theorem *bdd-from-sc*:

assumes *simplicial-complex.simplicial-complex* n ($K :: \text{nat set set}$)

shows $\langle \text{bdd-relator } rp\ s \rangle$

bdd-from-sc $K\ n\ s$

$\langle \lambda(r,s'). \text{bdd-relator } (\text{insert } (\text{vec-to-boolfunc } n (\text{bf-from-sc } K), r) rp) s' \rangle$

proof –

have fK : *finite* K

using *simplicial-complex.finite-simplicial-complex* [*OF assms*] .

have fv : $\forall v \in K. \text{finite } v$

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* $\equiv (\text{nat-list-from-sc } K)$

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* \rightarrow (*SML*) *IBDD*

and (*OCaml*) *IBDD* **and** (*Haskell*) *IBDD*

| **code-module** *Typerep* \rightarrow (*SML*) *IBDD*

and (*OCaml*) *IBDD* **and** (*Haskell*) *IBDD*

| **code-module** *String* \rightarrow (*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* :: $\text{nat} \Rightarrow (\text{bool vec} \Rightarrow \text{bool}) \Rightarrow (\text{bool vec} \Rightarrow \text{bool}) \Rightarrow (\text{bool vec} \Rightarrow \text{bool})$
where $(\text{bool-fun-or } n \ f \ g) \equiv (\lambda x. f \ x \ \vee \ g \ x)$

definition *bool-fun-and* :: $\text{nat} \Rightarrow (\text{bool vec} \Rightarrow \text{bool}) \Rightarrow (\text{bool vec} \Rightarrow \text{bool}) \Rightarrow (\text{bool vec} \Rightarrow \text{bool})$
where $(\text{bool-fun-and } n \ f \ g) \equiv (\lambda x. f \ x \ \wedge \ g \ x)$

lemma *eq-union-or*:

simplicial-complex-induced-by-monotone-boolean-function n $(\text{bool-fun-or } n \ f \ g)$
 $=$ *simplicial-complex-induced-by-monotone-boolean-function* $n \ f$
 \cup *simplicial-complex-induced-by-monotone-boolean-function* $n \ g$
(is $?sc \ n \ (?bf\text{-or } n \ f \ g) = ?sc \ n \ f \cup ?sc \ n \ g$ **)**

proof

show $?sc \ n \ f \cup ?sc \ n \ g \subseteq ?sc \ n \ (?bf\text{-or } n \ f \ g)$

proof

fix $\sigma :: \text{nat set}$

assume $\sigma \in (?sc \ n \ f \cup ?sc \ n \ g)$

hence *sigma*: $\sigma \in ?sc \ n \ f \vee \sigma \in ?sc \ n \ g$ **by** *auto*

have f (*simplicial-complex.bool-vec-from-simplice* $n \ \sigma$)
 $\vee g$ (*simplicial-complex.bool-vec-from-simplice* $n \ \sigma$)

proof (*cases* $\sigma \in ?sc \ n \ f$)

case *True*

from *simplicial-complex.simplicial-complex-implies-true* [*OF True*]

show f (*simplicial-complex.bool-vec-from-simplice* $n \ \sigma$)

$\vee g$ (*simplicial-complex.bool-vec-from-simplice* $n \ \sigma$) **by** *fast*

next

case *False*

hence *sigma*: $\sigma \in ?sc \ n \ g$ **using** *sigma* **by** *fast*

from *simplicial-complex.simplicial-complex-implies-true* [*OF sigma*]

show f (*simplicial-complex.bool-vec-from-simplice* $n \ \sigma$)

$\vee g$ (*simplicial-complex.bool-vec-from-simplice* $n \ \sigma$) **by** *fast*

qed

thus $\sigma \in ?sc \ n \ (?bf\text{-or } n \ f \ g)$

```

    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)  $\subseteq$  ?sc n f  $\cup$  ?sc n g
proof
  fix  $\sigma :: \text{nat set}$ 
  assume sigma:  $\sigma \in ?sc n (?bf-or n f g)$ 
  hence bool-fun-or n f g (simplicial-complex.bool-vec-from-simplice n  $\sigma$ )
    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  $\sigma$ ))
     $\vee$  (g (simplicial-complex.bool-vec-from-simplice n  $\sigma$ ))
    unfolding bool-fun-or-def
    by auto
  hence  $\sigma \in ?sc n f \vee \sigma \in ?sc n g$ 
    by (smt (z3) sigma bool-fun-or-def mem-Collect-eq
        simplicial-complex-induced-by-monotone-boolean-function-def)
  thus  $\sigma \in \text{simplicial-complex-induced-by-monotone-boolean-function } n f$ 
     $\cup \text{simplicial-complex-induced-by-monotone-boolean-function } n g$ 
    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
     $\cap$  simplicial-complex-induced-by-monotone-boolean-function n g
  (is ?sc n (?bf-and n f g) = ?sc n f  $\cap$  ?sc n g)
proof
  show ?sc n f  $\cap$  ?sc n g  $\subseteq$  ?sc n (?bf-and n f g)
  proof
    fix  $\sigma :: \text{nat set}$ 
    assume  $\sigma \in (?sc n f \cap ?sc n g)$ 
    hence sigma:  $\sigma \in ?sc n f \wedge \sigma \in ?sc n g$  by auto
    have f (simplicial-complex.bool-vec-from-simplice n  $\sigma$ )
       $\wedge$  g (simplicial-complex.bool-vec-from-simplice n  $\sigma$ )
    proof -
      from sigma have sigmaf:  $\sigma \in ?sc n f$  and sigmag:  $\sigma \in ?sc n g$ 
      by auto
      have f (simplicial-complex.bool-vec-from-simplice n  $\sigma$ )
        using simplicial-complex.simplicial-complex-implies-true [OF sigmaf] .
      moreover have g (simplicial-complex.bool-vec-from-simplice n  $\sigma$ )
        using simplicial-complex.simplicial-complex-implies-true [OF sigmag] .
      ultimately show ?thesis by fast
    qed
  qed
  thus  $\sigma \in ?sc n (?bf-and n f g)$ 

```

```

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)
     $\wedge$  (g (simplicial-complex.bool-vec-from-simplice n σ))
  )
  unfolding bool-fun-and-def
  by auto
  hence σ ∈ ?sc n f  $\wedge$  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
     $\cap$  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)
   $\Rightarrow$  (bool vec × bool vec ⇒ bool)
  where (bool-fun-ast n f)  $\equiv$  ( $\lambda$  (x,y). (fst f x)  $\wedge$  (snd f y))

```

```

definition
  simplicial-complex-induced-by-monotone-boolean-function-ast
   $::$  (nat × nat)  $\Rightarrow$  ((bool vec × bool vec ⇒ bool))  $\Rightarrow$  (nat set * nat set) set
  where simplicial-complex-induced-by-monotone-boolean-function-ast n f =
    {z.  $\exists$  x y. dim-vec x = fst n  $\wedge$  dim-vec y = snd n  $\wedge$  f (x, y)
       $\wedge$  ((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
   $\implies$  ( $\exists$  x y. f (x, y)  $\wedge$  (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

```

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

definition *set-ast* :: (*nat set*) *set* \Rightarrow (*nat set*) *set* \Rightarrow ((*nat set***nat set*) *set*)
where *set-ast* *A B* $\equiv \{c. \exists a \in A. \exists b \in B. c = (a, b)\}$

definition *set-fst* :: (*nat***nat*) *set* \Rightarrow *nat set*
where *set-fst* *AB* = {*a. \exists ab \in AB. a = fst ab*}

lemma *set-fst-simp* [*simp*]:

assumes $y \neq \{\}$
shows *set-fst* ($x \times y$) = *x*

proof

show *set-fst* ($x \times y$) \subseteq *x*
by (*smt* (*verit*) *SigmaE mem-Collect-eq prod.sel(1) set-fst-def subsetI*)
show $x \subseteq$ *set-fst* ($x \times y$)

proof

fix *a::nat*
assume $a \in x$
then obtain *b* **where** $b \in y$ **and** $(a, b) \in (x \times y)$
using *assms* **by** *blast*
then show $a \in$ *set-fst* ($x \times y$)
using *set-fst-def* **by** *fastforce*

qed

qed

definition *set-snd* :: (*nat***nat*) *set* \Rightarrow *nat set*
where *set-snd* *AB* = {*b. \exists ab \in AB. b = snd(ab)*}

lemma

simplicial-complex-ast-implies-fst-true:

assumes $\gamma \in$ *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 $\gamma \in$ *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  $\gamma :: \text{nat set} * \text{nat set}$ 
assume pert:  $\gamma \in \text{simplicial-complex-induced-by-monotone-boolean-function-ast}$ 
(n, m)
  (bool-fun-ast (n, m) f)
have sigma: (fst  $\gamma$ ) ∈ 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  $\gamma$ )], 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  $\gamma$ ) ∈ 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  $\gamma$ )],
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  $\gamma \in \text{set-ast}$ 
  (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.