

# A Formalization of the Exponential Blowup in the Transformations between CNF and DNF

Leon Raffael Schulz      Martin Desharnais-Schäfer  
Jan Johannsen

June 25, 2026

## Abstract

A well-known result about propositional logic is that transforming a formula into disjunctive or conjunctive normal form can lead to an exponential blowup of the formula size. We formalize this result in the form of two theorems. This formalization originated in Schulz's bachelor thesis [2] supervised by Johannsen with the help of Desharnais-Schäfer; the thesis formalizes the theorems as found in Johannsen's lecture notes of SAT solving [1]. The formalization was later refactored and expanded by Desharnais-Schäfer in collaboration with Schulz.

## Contents

<b>1</b>	<b>Move to HOL</b>	<b>2</b>
<b>2</b>	<b>Move to Propositional_Proof_Systems</b>	<b>3</b>
<b>3</b>	<b>Functions, Predicates, and Datatypes</b>	<b>3</b>
3.1	Formula Equivalence . . . . .	3
3.2	Conjunctive Normal Form . . . . .	3
3.3	Disjunctive Normal Form . . . . .	4
3.4	Big Conjunction . . . . .	4
3.5	Big Disjunction . . . . .	5
3.6	Formula Size . . . . .	6
3.7	Fn function . . . . .	14
3.8	Dual Function . . . . .	15
3.9	Formula Contains Atom . . . . .	18
<b>4</b>	<b>CNF to DNF</b>	<b>19</b>

## 5 DNF to CNF

**theory** *CNF-DNF-Exp-Blowup*

**imports**

*Main*

*Propositional-Proof-Systems.Formulas*

*Propositional-Proof-Systems.Sema*

*Propositional-Proof-Systems.CNF-Formulas*

**begin**

### 1 Move to HOL

**lemma** *card-Domain-le*:

**assumes** *finite A*

**shows**  $\text{card } (\text{Domain } A) \leq \text{card } A$

**using** *assms* **by** (*metis card-image-le fst-eq-Domain*)

**lemma** *card-le-card-if-mem-imp-ex-mem*:

**fixes**  $f :: 'a \Rightarrow 'b \Rightarrow 'c$  **and**  $\mathcal{X} :: 'a \text{ set}$  **and**  $\mathcal{Y} :: 'c \text{ set}$

**defines**  $XY \equiv \{(x, y). x \in \mathcal{X} \wedge f x y \in \mathcal{Y}\}$

**assumes** *finite  $\mathcal{X}$*  **and** *finite  $\mathcal{Y}$*  **and**

*f-inj*: *inj-on*  $(\lambda(x, y). f x y)$   $XY$  **and**

*ex-in- $\mathcal{Y}$* :  $\bigwedge x. x \in \mathcal{X} \implies \exists y. f x y \in \mathcal{Y}$

**shows**  $\text{card } \mathcal{X} \leq \text{card } \mathcal{Y}$

**proof** –

**have** *f-XY-subset*:  $(\lambda(x, y). f x y) \text{ ' } XY \subseteq \mathcal{Y}$

**using** *XY-def* **by** *auto*

**then have** *finite*  $((\lambda(x, y). f x y) \text{ ' } XY)$

**using**  $\langle \text{finite } \mathcal{Y} \rangle$  **by** (*rule finite-subset*)

**then have** *finite*  $XY$

**by** (*rule finite-image-iff[THEN iffD1, OF f-inj]*)

**moreover have**  $\text{Domain } XY = \mathcal{X}$

**unfolding** *XY-def*

**using** *ex-in- $\mathcal{Y}$*

**by** (*simp add: equalityI subsetI*)

**ultimately have**  $\text{card } \mathcal{X} \leq \text{card } XY$

**using** *card-Domain-le* **by** *iprover*

**also have**  $\dots \leq \text{card } \mathcal{Y}$

**using** *inj-on-iff-card-le[OF  $\langle \text{finite } XY \rangle \langle \text{finite } \mathcal{Y} \rangle$ ]*

**using** *f-XY-subset f-inj* **by** *blast*

**finally show**  $\text{card } \mathcal{X} \leq \text{card } \mathcal{Y}$  .

**qed**

## 2 Move to Propositional\_Proof\_Systems

**lemma** *is-disj-if-is-lit-plus*: *is-lit-plus*  $\varphi \implies$  *is-disj*  $\varphi$   
by (*induction*  $\varphi$  rule: *is-lit-plus.induct*) *simp-all*

**lemma** *disj-is-cnf*: *is-disj*  $\varphi \implies$  *is-cnf*  $\varphi$   
by (*induction*  $\varphi$ ) *auto*

**lemma** *cnf-in-nnf*: *is-cnf*  $\varphi \implies$  *is-nnf*  $\varphi$   
by (*induction*  $\varphi$ ) (*simp-all add: disj-is-nnf is-disj-if-is-lit-plus*)

## 3 Functions, Predicates, and Datatypes

### 3.1 Formula Equivalence

**definition** *equiv* :: 'a formula  $\Rightarrow$  'a formula  $\Rightarrow$  bool **where**  
*equiv*  $\varphi \psi \longleftrightarrow (\forall \alpha. (\alpha \models \varphi) \longleftrightarrow (\alpha \models \psi))$

**lemma** *equiv-reflexive*:  $\bigwedge \varphi. \text{equiv } \varphi \varphi$   
**unfolding** *equiv-def* **by** *simp*

**lemma** *equiv-symmetric[sym]*:  $\bigwedge \varphi \psi. \text{equiv } \varphi \psi \implies \text{equiv } \psi \varphi$   
**unfolding** *equiv-def* **by** *simp*

**lemma** *equiv-transitive[trans]*:  $\bigwedge \xi \varphi \psi. \text{equiv } \xi \varphi \implies \text{equiv } \varphi \psi \implies \text{equiv } \xi \psi$   
**unfolding** *equiv-def* **by** *simp*

**lemma** *equiv-Not-left-Not-right[simp]*:  $\bigwedge \varphi \psi. \text{equiv } (\text{Not } \varphi) (\text{Not } \psi) \longleftrightarrow \text{equiv } \varphi \psi$   
**unfolding** *equiv-def* **by** *simp-all*

**lemma** *equiv-Not-Not-left[simp]*:  $\bigwedge \varphi \psi. \text{equiv } (\text{Not } (\text{Not } \varphi)) \psi \longleftrightarrow \text{equiv } \varphi \psi$   
**unfolding** *equiv-def* **by** *simp-all*

**lemma** *equiv-Not-Not-right[simp]*:  $\bigwedge \varphi \psi. \text{equiv } \varphi (\text{Not } (\text{Not } \psi)) \longleftrightarrow \text{equiv } \varphi \psi$   
**unfolding** *equiv-def* **by** *simp-all*

### 3.2 Conjunctive Normal Form

**fun** *unconj* :: 'a formula  $\Rightarrow$  'a formula list **where**  
*unconj* (*And*  $\varphi \psi$ ) = *unconj*  $\varphi$  @ *unconj*  $\psi$  |  
*unconj*  $\varphi$  = [ $\varphi$ ]

**lemma** *unconj-neq-Nil[simp]*: *unconj*  $\varphi \neq []$   
by (*induction*  $\varphi$ ) *simp-all*

**fun** *count-And* :: 'a formula  $\Rightarrow$  nat **where**  
*count-And* (*And*  $\varphi \psi$ ) = *count-And*  $\varphi$  + *count-And*  $\psi$  + 1 |  
*count-And* - = 0

**lemma** *length-unconj*:  $\text{length } (\text{unconj } \varphi) = \text{count-And } \varphi + 1$   
**by** (*induction*  $\varphi$ ) *simp-all*

**lemma** *ball-unconj-is-disj*:  
**fixes**  $\varphi :: 'a$  *formula*  
**assumes** *is-cnf*  $\varphi$   
**shows**  $\bigwedge C. C \in \text{set } (\text{unconj } \varphi) \implies \text{is-disj } C$   
**using** *assms*  
**by** (*induction*  $\varphi$  *rule: is-cnf.induct*) *auto*

### 3.3 Disjunctive Normal Form

**fun** *is-conj* :: *'a formula*  $\Rightarrow$  *bool* **where**  
*is-conj* (*And*  $\varphi \psi$ )  $\longleftrightarrow$  (*is-lit-plus*  $\varphi \wedge \text{is-conj } \psi$ ) |  
*is-conj*  $\varphi \longleftrightarrow \text{is-lit-plus } \varphi$

**fun** *is-dnf* :: *'a formula*  $\Rightarrow$  *bool* **where**  
*is-dnf* (*Or*  $\varphi \psi$ )  $\longleftrightarrow$  (*is-dnf*  $\varphi \wedge \text{is-dnf } \psi$ ) |  
*is-dnf*  $\varphi \longleftrightarrow \text{is-conj } \varphi$

**lemma** *conj-is-dnf*:  $\text{is-conj } \varphi \implies \text{is-dnf } \varphi$   
**by** (*induction*  $\varphi$ ) *auto*

**fun** *undisj* :: *'a formula*  $\Rightarrow$  *'a formula list* **where**  
*undisj* (*Or*  $\varphi \psi$ ) = *undisj*  $\varphi$  @ *undisj*  $\psi$  |  
*undisj*  $\varphi$  = [ $\varphi$ ]

**lemma** *undisj-neq-Nil[simp]*:  $\text{undisj } \varphi \neq []$   
**by** (*induction*  $\varphi$ ) *simp-all*

**lemma** *ball-undisj-is-conj*:  
**fixes**  $\varphi :: 'a$  *formula*  
**assumes** *is-dnf*  $\varphi$   
**shows**  $\bigwedge T. T \in \text{set } (\text{undisj } \varphi) \implies \text{is-conj } T$   
**using** *assms*  
**by** (*induction*  $\varphi$  *rule: is-dnf.induct*) *auto*

**fun** *count-Or* :: *'a formula*  $\Rightarrow$  *nat* **where**  
*count-Or* (*Or*  $\varphi \psi$ ) = *count-Or*  $\varphi$  + *count-Or*  $\psi$  + 1 |  
*count-Or* - = 0

**lemma** *length-undisj*:  $\text{length } (\text{undisj } \varphi) = \text{count-Or } \varphi + 1$   
**by** (*induction*  $\varphi$ ) *simp-all*

### 3.4 Big Conjunction

**fun** *BigAnd'* :: *'a formula list*  $\Rightarrow$  *'a formula* **where**  
*BigAnd'* [] = ( $\neg \perp$ ) |  
*BigAnd'* [*F*] = *F* |

$$BigAnd' (F \# Fs) = F \wedge BigAnd' Fs$$

**lemma** *atoms-BigAnd'[simp]*:  $atoms (BigAnd' Fs) = \bigcup (atoms \text{ ' set } Fs)$   
**by** (*induction Fs rule: BigAnd'.induct*) *simp-all*

**lemma** *BigAnd'-semantics[simp]*:  $A \models BigAnd' Ts \iff (\forall f \in set Ts. A \models f)$   
**by** (*induction Ts rule: BigAnd'.induct*) *simp-all*

**lemma** *is-cnf-BigAnd'*:  $(\forall C \in set Cs. is-disj C \wedge \neg(\forall \alpha. \alpha \models C)) \implies is-cnf (BigAnd' Cs)$   
**by** (*induction Cs rule: BigAnd'.induct*) (*simp-all add: disj-is-cnf*)

**lemma** *equiv-BigAnd'-append*:  $equiv (BigAnd' (xs @ ys)) (And (BigAnd' xs) (BigAnd' ys))$   
**by** (*induction xs*) (*simp-all add: equiv-def*)

### 3.5 Big Disjunction

**fun** *BigOr'* :: 'a formula list  $\Rightarrow$  'a formula **where**  
*BigOr' Nil* =  $\perp$  |  
*BigOr' [F]* =  $F$  |  
*BigOr' (F # Fs)* =  $F \vee BigOr' Fs$

**lemma** *atoms-BigOr'[simp]*:  $atoms (BigOr' Fs) = \bigcup (atoms \text{ ' set } Fs)$   
**by** (*induction Fs rule: BigOr'.induct*) *simp-all*

**lemma** *BigOr'-semantics[simp]*:  $A \models BigOr' Ts \iff (\exists f \in set Ts. A \models f)$   
**by** (*induction Ts rule: BigOr'.induct*) *simp-all*

**lemma** *is-dnf-BigOr'*:  $(\forall T \in set Ts. is-conj T \wedge (\exists \alpha. \alpha \models T)) \implies is-dnf (BigOr' Ts)$

**proof** (*induction Ts*)

**case** *Nil*

**then show** *?case by simp*

**next**

**case** (*Cons T Ts*)

**then show** *?case*

**by** (*metis conj-is-dnf list.set-intros(1) BigOr'.simps(2) BigOr'.simps(3) list.set-intros(2) is-dnf.simps(1) neq-Nil-conv*)

**qed**

**lemma** *equiv-BigOr'-append*:  $equiv (BigOr' (xs @ ys)) (Or (BigOr' xs) (BigOr' ys))$   
**by** (*induction xs*) (*simp-all add: equiv-def*)

**lemma** *equiv-BigOr'-undisj-if-dnf*:

**fixes**  $\varphi :: \text{'a formula}$

**shows**  $equiv (BigOr' (undisj \varphi)) \varphi$

**using** *equiv-BigOr'-append*

by (induction  $\varphi$  rule: is-dnf.induct) (auto simp add: equiv-def)

**lemma** *equiv-BigAnd'-unconj-if-cnf*:  
**fixes**  $\varphi :: 'a$  formula  
**shows** *equiv (BigAnd' (unconj  $\varphi$ ))  $\varphi$*   
**proof** (induction  $\varphi$  rule: unconj.induct)  
**case** (1  $\varphi \psi$ )  
**then show** ?case  
**using** *equiv-BigAnd'-append*  
**by** (auto simp add: equiv-def)  
**qed** (simp-all add: equiv-def)

### 3.6 Formula Size

Similar to *size*, but ignores  $\neg$  when calculating the size.

**fun** *sizef* :: 'a formula  $\Rightarrow$  nat **where**  
*sizef Bot* = 1 |  
*sizef (Atom a)* = 1 |  
*sizef (Not  $\varphi$ )* = *sizef  $\varphi$*  |  
*sizef (And  $\varphi \psi$ )* = *sizef  $\varphi$*  + *sizef  $\psi$*  + 1 |  
*sizef (Or  $\varphi \psi$ )* = *sizef  $\varphi$*  + *sizef  $\psi$*  + 1 |  
*sizef (Imp  $\varphi \psi$ )* = *sizef  $\varphi$*  + *sizef  $\psi$*  + 1

**lemma** *Suc-0-le-sizef[simp]*: *Suc 0  $\leq$  sizef  $\varphi$*   
**by** (induction  $\varphi$ ) *simp-all*

**lemma** *Suc-0-le-size[simp]*:  
**fixes**  $\varphi :: 'a$  formula  
**shows** *Suc 0  $\leq$  size  $\varphi$*   
**by** (induction  $\varphi$ ) *simp-all*

**lemma** *sizef-le-size*: *sizef  $\varphi \leq$  size  $\varphi$*   
**by** (induction  $\varphi$ ) *simp-all*

**lemma** *card-atoms-le-sizef*: *card (atoms  $\varphi$ )  $\leq$  sizef  $\varphi$*   
**proof** (induction  $\varphi$ )  
**case** (And  $F1 F2$ )  
**have** *card (atoms (F1  $\wedge$  F2)) = card (atoms F1  $\cup$  atoms F2)*  
**by** *simp*  
**also have**  $\dots \leq$  *card (atoms F1) + card (atoms F2)*  
**using** *card-Un-le* **by** *metis*  
**also have**  $\dots <$  *Suc (card (atoms F1) + card (atoms F2))*  
**by** *presburger*  
**also have**  $\dots \leq$  *Suc (sizef F1 + sizef F2)*  
**using** *And.IH* **by** *presburger*  
**also have**  $\dots =$  *sizef (F1  $\wedge$  F2)*  
**by** *simp*  
**finally show** ?case  
**by** *presburger*

```

next
  case (Or F1 F2)
  have card (atoms (F1  $\vee$  F2)) = card (atoms F1  $\cup$  atoms F2)
    by simp
  also have ...  $\leq$  card (atoms F1) + card (atoms F2)
    using card-Un-le by metis
  also have ...  $<$  Suc (card (atoms F1) + card (atoms F2))
    by presburger
  also have ...  $\leq$  Suc (sizef F1 + sizef F2)
    using Or.IH by presburger
  also have ... = sizef (F1  $\vee$  F2)
    by simp
  finally show ?case
    by presburger
next
  case (Imp F1 F2)
  have card (atoms (F1  $\rightarrow$  F2)) = card (atoms F1  $\cup$  atoms F2)
    by simp
  also have ...  $\leq$  card (atoms F1) + card (atoms F2)
    using card-Un-le by metis
  also have ...  $<$  Suc (card (atoms F1) + card (atoms F2))
    by presburger
  also have ...  $\leq$  Suc (sizef F1 + sizef F2)
    using Imp.IH by presburger
  also have ... = sizef (F1  $\rightarrow$  F2)
    by simp
  finally show ?case
    by presburger
qed simp-all

```

**lemma** *card-atoms-le-size*:  $\text{card (atoms } \varphi) \leq \text{size } \varphi$   
 by (metis card-atoms-le-sizef le-trans sizef-le-size)

**lemma** *aux-exp-sizef*:  $\text{length } Ts = n \implies \forall T \in \text{set } Ts. \text{sizef } T \geq m \implies \text{sizef (BigOr' } Ts) \geq n * m$   
 by (induction Ts arbitrary: m n rule: BigOr'.induct; fastforce)

**lemma** *aux-exp-size*:  $\text{length } Ts = n \implies \forall T \in \text{set } Ts. \text{size } T \geq m \implies \text{size (BigOr' } Ts) \geq n * m$   
 by (induction Ts arbitrary: m n rule: BigOr'.induct; fastforce)

**lemma** *exp-sizef*:  
 assumes  $n > 0$  and  $\text{length } Ts \geq 2^n$  and  $\forall T \in \text{set } Ts. \text{sizef } T \geq m$   
 shows  $\text{sizef (BigOr' } Ts) \geq 2^n * m$   
 using assms  
**proof** (induction Ts arbitrary: n m rule: BigOr'.induct)  
 case 1  
 then show ?case  
 by simp

```

next
  case (2  $\varphi$ )
  then have False
  by (metis list.size(3) One-nat-def length-Cons leD one-less-power less-2-cases-iff)
  then show ?case ..
next
  case (3  $T T' Ts'$ )
  define  $Ts$  where
     $Ts = T' \# Ts'$ 
  have  $2^{\wedge} n \leq \text{length } Ts \vee 2^{\wedge} n = \text{Suc } (\text{length } Ts)$ 
  unfolding  $Ts\text{-def}$  using 3.prem1 by auto
  then have  $2^{\wedge} n * m \leq \text{Suc } (\text{sizef } T + \text{sizef } (\text{BigOr}' Ts))$ 
  proof (elim disjE)
    assume asm1:  $2^{\wedge} n \leq \text{length } Ts$ 

    have  $2^{\wedge} n * m \leq \text{sizef } (\text{BigOr}' (T' \# Ts'))$ 
    proof (rule 3.IH)
      show  $0 < n$ 
      by (metis 3.prem1)
    next
      show  $2^{\wedge} n \leq \text{length } (T' \# Ts')$ 
      using  $Ts\text{-def}$  asm1 by blast
    next
      show  $(\forall T \in \text{set } (T' \# Ts'). m \leq \text{sizef } T)$ 
      by (simp add: 3.prem3)  $Ts\text{-def}$ 
  qed

  also have  $\dots \leq \text{Suc } (\text{sizef } T + \text{sizef } (\text{BigOr}' Ts))$ 
  by (simp add:  $Ts\text{-def}$ )

  finally show ?thesis .
next
  assume  $2^{\wedge} n = \text{Suc } (\text{length } Ts)$ 

  then have  $2^{\wedge} n = \text{length } (T \# Ts)$ 
  by simp

  moreover have  $\forall T \in \text{set } (T \# Ts). \text{sizef } T \geq m$ 
  by (metis 3.prem3)  $Ts\text{-def}$ 

  ultimately show ?thesis
  using aux-exp-sizef by fastforce
qed
then show ?case
  by (simp add:  $Ts\text{-def}$ )
qed

lemma exp-size:
  assumes  $n > 0$  and  $\text{length } Ts \geq 2^{\wedge} n$  and  $\forall T \in \text{set } Ts. \text{sizef } T \geq m$ 

```

```

shows size (BigOr' Ts) ≥ 2n * m
using assms
proof (induction Ts arbitrary: n m rule: BigOr'.induct)
  case 1
  then show ?case
    by simp
next
  case (2 φ)
  then have False
    by (metis list.size(3) One-nat-def length-Cons leD one-less-power less-2-cases-iff)
  then show ?case ..
next
  case (3 T T' Ts')
  define Ts where
    Ts = T' # Ts'
  have 2n ≤ length Ts ∨ 2n = Suc (length Ts)
    unfolding Ts-def using 3.prem1 by auto
  then have 2n * m ≤ Suc (size T + size (BigOr' Ts))
  proof (elim disjE)
    assume asm1: 2n ≤ length Ts

    have 2n * m ≤ size (BigOr' (T' # Ts'))
    proof (rule 3.IH)
      show 0 < n
        by (metis 3.prem1)
    next
      show 2n ≤ length (T' # Ts')
        using Ts-def asm1 by blast
    next
      show (∀ T ∈ set (T' # Ts'). m ≤ size T)
        by (simp add: 3.prem2) Ts-def
    qed

    also have ... ≤ Suc (size T + size (BigOr' Ts))
      by (simp add: Ts-def)

    finally show ?thesis .
  next
    assume 2n = Suc (length Ts)

    then have 2n = length (T # Ts)
      by simp

    moreover have ∀ T ∈ set (T # Ts). size T ≥ m
      by (metis 3.prem3) Ts-def

    ultimately show ?thesis
      using aux-exp-size by fastforce
  qed

```

**then show** *?case*  
**by** (*simp add: Ts-def*)  
**qed**

**lemma** *sizef-BigOr'*:  $xs \neq [] \implies \text{sizef } (\text{BigOr}' xs) + 1 = \text{sum-list } (\text{map sizef } xs) + \text{length } xs$   
**by** (*induction xs rule: BigOr'.induct*) *simp-all*

**lemma** *size-BigOr'*:  $xs \neq [] \implies \text{size } (\text{BigOr}' xs) + 1 = \text{sum-list } (\text{map size } xs) + \text{length } xs$   
**by** (*induction xs rule: BigOr'.induct*) *simp-all*

**lemma** *sizef-BigAnd'*:  $xs \neq [] \implies \text{sizef } (\text{BigAnd}' xs) + 1 = \text{sum-list } (\text{map sizef } xs) + \text{length } xs$   
**by** (*induction xs rule: BigAnd'.induct*) *simp-all*

**lemma** *size-BigAnd'*:  $xs \neq [] \implies \text{size } (\text{BigAnd}' xs) + 1 = \text{sum-list } (\text{map size } xs) + \text{length } xs$   
**by** (*induction xs rule: BigAnd'.induct*) *simp-all*

**lemma** *sizef-conv-sum-list-undisj*:  $\text{sizef } \varphi = \text{sum-list } (\text{map sizef } (\text{undisj } \varphi)) + \text{count-Or } \varphi$   
**by** (*induction \varphi*) *simp-all*

**lemma** *size-conv-sum-list-undisj*:  $\text{size } \varphi = \text{sum-list } (\text{map size } (\text{undisj } \varphi)) + \text{count-Or } \varphi$   
**by** (*induction \varphi*) *simp-all*

**lemma** *sizef-conv-sum-list-unconj*:  $\text{sizef } \varphi = \text{sum-list } (\text{map sizef } (\text{unconj } \varphi)) + \text{count-And } \varphi$   
**by** (*induction \varphi*) *simp-all*

**lemma** *size-conv-sum-list-unconj*:  $\text{size } \varphi = \text{sum-list } (\text{map size } (\text{unconj } \varphi)) + \text{count-And } \varphi$   
**by** (*induction \varphi*) *simp-all*

**lemma** *sizef-BigOr'-undisj*:  
**fixes**  $\varphi :: 'a \text{ formula}$   
**shows**  $\text{sizef } (\text{BigOr}' (\text{undisj } \varphi)) = \text{sizef } \varphi$

**proof** –  
**have**  $\text{sizef } \varphi + 1 = \text{sum-list } (\text{map sizef } (\text{undisj } \varphi)) + \text{count-Or } \varphi + 1$   
**using** *sizef-conv-sum-list-undisj*[of  $\varphi$ ] **by** *presburger*

**also have**  $\dots = \text{sum-list } (\text{map sizef } (\text{undisj } \varphi)) + \text{length } (\text{undisj } \varphi)$   
**using** *length-undisj*[of  $\varphi$ ] **by** *presburger*

**also have**  $\dots = \text{sizef } (\text{BigOr}' (\text{undisj } \varphi)) + 1$   
**using** *sizef-BigOr'*[of *undisj*  $\varphi$ , *simplified*] **by** *presburger*

**finally show** *?thesis*  
 by *presburger*  
**qed**

**lemma** *size-BigOr'-undisj*:  
 fixes  $\varphi :: 'a \text{ formula}$   
 shows  $\text{size } (\text{BigOr}' (\text{undisj } \varphi)) = \text{size } \varphi$   
**proof** –  
 have  $\text{size } \varphi + 1 = \text{sum-list } (\text{map } \text{size } (\text{undisj } \varphi)) + \text{count-Or } \varphi + 1$   
 using *size-conv-sum-list-undisj*[of  $\varphi$ ] **by** *presburger*  
  
 also have  $\dots = \text{sum-list } (\text{map } \text{size } (\text{undisj } \varphi)) + \text{length } (\text{undisj } \varphi)$   
 using *length-undisj*[of  $\varphi$ ] **by** *presburger*  
  
 also have  $\dots = \text{size } (\text{BigOr}' (\text{undisj } \varphi)) + 1$   
 using *size-BigOr'*[of *undisj*  $\varphi$ , *simplified*] **by** *presburger*

**finally show** *?thesis*  
 by *presburger*  
**qed**

**lemma** *sizef-BigAnd'-unconj*:  
 fixes  $\varphi :: 'a \text{ formula}$   
 shows  $\text{sizef } (\text{BigAnd}' (\text{unconj } \varphi)) = \text{sizef } \varphi$   
**proof** –  
 have  $\text{sizef } \varphi + 1 = \text{sum-list } (\text{map } \text{sizef } (\text{unconj } \varphi)) + \text{count-And } \varphi + 1$   
 using *sizef-conv-sum-list-unconj*[of  $\varphi$ ] **by** *presburger*  
  
 also have  $\dots = \text{sum-list } (\text{map } \text{sizef } (\text{unconj } \varphi)) + \text{length } (\text{unconj } \varphi)$   
 using *length-unconj*[of  $\varphi$ ] **by** *presburger*  
  
 also have  $\dots = \text{sizef } (\text{BigAnd}' (\text{unconj } \varphi)) + 1$   
 using *sizef-BigAnd'*[of *unconj*  $\varphi$ , *simplified*] **by** *presburger*

**finally show** *?thesis*  
 by *presburger*  
**qed**

**lemma** *size-BigAnd'-unconj*:  
 fixes  $\varphi :: 'a \text{ formula}$   
 shows  $\text{size } (\text{BigAnd}' (\text{unconj } \varphi)) = \text{size } \varphi$   
**proof** –  
 have  $\text{size } \varphi + 1 = \text{sum-list } (\text{map } \text{size } (\text{unconj } \varphi)) + \text{count-And } \varphi + 1$   
 using *size-conv-sum-list-unconj*[of  $\varphi$ ] **by** *presburger*  
  
 also have  $\dots = \text{sum-list } (\text{map } \text{size } (\text{unconj } \varphi)) + \text{length } (\text{unconj } \varphi)$   
 using *length-unconj*[of  $\varphi$ ] **by** *presburger*  
  
 also have  $\dots = \text{size } (\text{BigAnd}' (\text{unconj } \varphi)) + 1$

```

    using size-BigAnd'[of unconj  $\varphi$ , simplified] by presburger

  finally show ?thesis
    by presburger
qed

lemma sizef-BigOr'-filter-le: sizef (BigOr' (filter P xs))  $\leq$  sizef (BigOr' xs)
proof (induction xs rule: BigOr'.induct)
  case 1
  then show ?case
    by simp
next
  case (2  $\varphi$ )
  then show ?case
    by simp
next
  case (3 F v va)

  have sizef (BigOr' (filter P (F # v # va)))  $\leq$  sizef (BigOr' (F # filter P (v #
va)))
  proof (cases P F)
    case True
    then show ?thesis
      by simp
  next
    case False
    then show ?thesis
      by (cases filter P (v # va)) simp-all
  qed

  also have ...  $\leq$  sizef (BigOr' (F # v # va))
  proof (cases filter P (v # va))
    case Nil
    then show ?thesis
      by simp
  next
    case (Cons G Gs)
    then have sizef (BigOr' (F # filter P (v # va))) =
      sizef F + sizef (BigOr' (filter P (v # va))) + 1
      by simp
    also have ...  $\leq$  sizef (BigOr' (F # v # va))
      using 3 by simp
    finally show ?thesis .
  qed

  finally show ?case .
qed

lemma size-BigOr'-filter-le-if:

```

```

assumes  $\exists x \in \text{set } xs. P x$ 
shows  $\text{size } (\text{BigOr}' (\text{filter } P \text{ } xs)) \leq \text{size } (\text{BigOr}' xs)$ 
using assms
proof (induction xs rule: BigOr'.induct)
  case 1
  then show ?case
    by simp
  next
  case (2 F)
  then show ?case
    by simp
  next
  case (3 F v va)

  then have IH:  $\text{size } (\text{BigOr}' (\text{filter } P (v \# va))) \leq \text{size } (\text{BigOr}' (v \# va))$ 
    by (metis BigOr'.simps(1) One-nat-def empty-filter-conv formula.size(8) formula.size-neq
      leI less-one)

  have  $\text{size } (\text{BigOr}' (\text{filter } P (F \# v \# va))) \leq \text{size } (\text{BigOr}' (F \# \text{filter } P (v \# va)))$ 
  proof (cases P F)
    case True
    then show ?thesis
      by simp
    next
    case False
    then show ?thesis
      by (cases filter P (v # va) simp-all)
  qed

  also have  $\dots \leq \text{size } (\text{BigOr}' (F \# v \# va))$ 
  proof (cases filter P (v # va))
    case Nil
    then show ?thesis
      by simp
    next
    case (Cons G Gs)
    then have  $\text{size } (\text{BigOr}' (F \# \text{filter } P (v \# va))) =$ 
       $\text{size } F + \text{size } (\text{BigOr}' (\text{filter } P (v \# va))) + 1$ 
      by simp
    also have  $\dots \leq \text{size } (\text{BigOr}' (F \# v \# va))$ 
      using IH by auto
    finally show ?thesis .
  qed

  finally show ?case .
qed

```

```

lemma sizef-BigAnd'-filter-le: sizef (BigAnd' (filter P xs)) ≤ sizef (BigAnd' xs)
proof (induction xs rule: BigAnd'.induct)
  case 1
  then show ?case
    by simp
next
  case (2 F)
  then show ?case
    by simp
next
  case (3 F v va)
  then show ?case
    by (metis BigAnd'.simps(1) BigOr'.simps(1) add-diff-cancel-right diff-is-0-eq
sizef.simps(3)
sizef-BigAnd' sizef-BigOr' sizef-BigOr'-filter-le)
qed

```

### 3.7 Fn function

```

datatype var = Var nat bool

```

```

lemma inj-on-Var[simp]: inj-on (λ(x, y). Var x y) A for A
  by (rule inj-onI) (simp add: case-prod-beta prod-eq-iff)

```

```

fun Fn :: nat ⇒ var formula where
  Fn 0 = ( $\neg \perp$ )|
  Fn (Suc n) =
    And
      (And
        (Or
          (Atom (Var (Suc n) False))
          (Atom (Var (Suc n) True))))
        (Or
          (Not (Atom (Var (Suc n) False)))
          (Not (Atom (Var (Suc n) True)))))))
  (Fn n)

```

```

lemma sizef-Fn: sizef (Fn n) = 8 * n + 1
  by (induction n) auto

```

```

lemma size-Fn: size (Fn n) = 10 * n + 2
  by (induction n) auto

```

```

lemma is-cnf-Fn: is-cnf (Fn n)
  by (induction n; auto)

```

```

lemma is-nnf-Fn: is-nnf (Fn n)
  using is-cnf-Fn[THEN cnf-in-nnf] .

```

```

lemma Fn-sat:  $\exists \alpha. \alpha \models \text{Fn } n$ 
proof –
  define  $\alpha$  where  $\alpha = (\lambda x. \text{case } x \text{ of } (\text{Var } i \ b) \Rightarrow b)$ 
  then have  $\alpha \models \text{Fn } n$ 
    by (induction n; simp)
  then show ?thesis
    by auto
qed

lemma semantics-Fn-iff:  $\alpha \models \text{Fn } n \iff (\forall i \in \{1..n\}. \alpha (\text{Var } i \ \text{False}) \neq \alpha (\text{Var } i \ \text{True}))$ 
proof (induction n)
  case 0
  show ?case
    by simp
next
  case (Suc n)

  let ?F =
    And
    (Or
      (Atom (Var (Suc n) False))
      (Atom (Var (Suc n) True)))
    (Or
      (Not (Atom (Var (Suc n) False)))
      (Not (Atom (Var (Suc n) True))))

  have  $\alpha \models \text{Fn } (\text{Suc } n) \iff \alpha \models \text{And } ?F (\text{Fn } n)$ 
    by simp
  also have  $\dots \iff \alpha \models ?F \wedge \alpha \models (\text{Fn } n)$ 
    by simp
  also have  $\dots \iff (\alpha (\text{Var } (\text{Suc } n) \ \text{False}) \neq \alpha (\text{Var } (\text{Suc } n) \ \text{True})) \wedge \alpha \models (\text{Fn } n)$ 
    by force
  also have  $\dots \iff (\alpha (\text{Var } (\text{Suc } n) \ \text{False}) \neq \alpha (\text{Var } (\text{Suc } n) \ \text{True})) \wedge (\forall i \in \{1..n\}. \alpha (\text{Var } i \ \text{False}) \neq \alpha (\text{Var } i \ \text{True}))$ 
    unfolding Suc.IH ..
  also have  $\dots \iff (\forall i \in \text{insert } (\text{Suc } n) \ \{1..n\}. \alpha (\text{Var } i \ \text{False}) \neq \alpha (\text{Var } i \ \text{True}))$ 
    by simp
  also have  $\dots \iff (\forall i \in \{1.. \text{Suc } n\}. \alpha (\text{Var } i \ \text{False}) \neq \alpha (\text{Var } i \ \text{True}))$ 
    by (simp add: atLeastAtMostSuc-conv)
  finally show ?case .
qed

```

### 3.8 Dual Function

```

primrec dual :: 'a formula  $\Rightarrow$  'a formula where
  dual Bot = Not Bot |
  dual (Atom v) = Not (Atom v) |

```

$dual (Not v) = v \mid$   
 $dual (And \varphi \psi) = Or (dual \varphi) (dual \psi) \mid$   
 $dual (Or \varphi \psi) = And (dual \varphi) (dual \psi) \mid$   
 $dual (Imp \varphi \psi) = And \varphi (dual \psi)$

**lemma** *sizef-dual-Fn*:  $sizef (dual (Fn n)) = 8 * n + 1$   
**by** (*induction n*) *simp-all*

**lemma** *size-dual-Fn*:  $size (dual (Fn n)) = 10 * n + 1$   
**by** (*induction n*) *simp-all*

**lemma** *is-dnf-dual-Fn*:  $is-dnf (dual (Fn n))$   
**by** (*induction n*) *simp-all*

**lemma** *sizef-dual*:  $sizef (dual \varphi) = sizef \varphi$   
**by** (*induction \varphi*) *simp-all*

**lemma** *size-dual-le*:  $size (dual \varphi) \leq 2 * size \varphi$   
**by** (*induction \varphi*) *simp-all*

**lemma** *equiv-dual*:  $equiv (dual \varphi) (Not \varphi)$   
**by** (*induction \varphi*) (*simp-all add: equiv-def*)

**lemma** *is-disj-dual-if-is-conj*:  $is-conj \varphi \implies is-disj (dual \varphi)$

**proof** (*induction \varphi*)

**case** (*Not \varphi*)

**then show** *?case*

**by** (*metis is-conj.simps(4) is-disj.simps(2,3) is-lit-plus.simps(1,3)*  
*is-nnf.simps(6) is-nnf-NotD dual.simps(3)*)

**next**

**case** (*And F1 F2*)

**then show** *?case*

**unfolding** *dual.simps is-conj.simps is-disj.simps*

**by** (*metis dual.simps(1,2,3) is-lit-plus.elims(2) is-lit-plus.simps(1,2,3,4)*)

**qed** *simp-all*

**lemma** *is-conj-dual-if-is-disj*:  $is-disj \varphi \implies is-conj (dual \varphi)$

**proof** (*induction \varphi*)

**case** (*Not \varphi*)

**then show** *?case*

**using** *is-nnf-NotD* **by** *force*

**next**

**case** (*Or F1 F2*)

**then show** *?case*

**unfolding** *dual.simps is-conj.simps is-disj.simps*

**by** (*metis dual.simps(1,2,3) is-lit-plus.elims(2) is-lit-plus.simps(1,2,3,4)*)

**qed** *simp-all*

**lemma** *is-dnf-dual-if-is-cnf*:  $is-cnf \varphi \implies is-dnf (dual \varphi)$

```

proof (induction  $\varphi$ )
  case (Not  $\varphi$ )
  then show ?case
    using conj-is-dnf is-cnf.simps(4) is-conj-dual-if-is-disj by blast
next
  case (Or F1 F2)
  then show ?case
    using conj-is-dnf is-cnf.simps(5) is-conj-dual-if-is-disj by blast
qed simp-all

```

```

lemma is-cnf-dual-if-is-dnf: is-dnf  $\varphi \implies$  is-cnf (dual  $\varphi$ )
proof (induction  $\varphi$ )
  case (Not  $\varphi$ )
  then show ?case
    using disj-is-cnf is-disj-dual-if-is-conj is-dnf.simps(4) by blast
next
  case (And F1 F2)
  then show ?case
    using disj-is-cnf is-disj-dual-if-is-conj is-dnf.simps(5) by blast
qed auto

```

```

lemma dual-disj-not-taut-impl-sat: is-disj  $\varphi \implies \exists \alpha. \neg \alpha \models \varphi \implies \exists \alpha. \alpha \models$  dual
 $\varphi$ 
proof (induction  $\varphi$ )
  case (Or F1 F2)
  have F-is-nnf: is-nnf (F1  $\vee$  F2)
    using Or.prem(1) disj-is-nnf by blast
  then have equiv: equiv (Not (F1  $\vee$  F2))(dual (F1  $\vee$  F2))
    using equiv-dual equiv-def by blast
  obtain  $\alpha$  where  $\alpha$ -def:  $\neg \alpha \models F1 \vee F2$ 
    using Or.prem(2) by auto
  then have  $\alpha \models$  Not (F1  $\vee$  F2)
    by auto
  then have  $\alpha \models$  dual (F1  $\vee$  F2)
    using equiv by (simp add: equiv-def)
  then show ?case
    by auto
qed auto

```

```

lemma dual-conj-of-disjs-is-disj-of-conjs:
  fixes Cs
  assumes  $\forall C \in \text{set } Cs. \text{is-disj } C \wedge (\exists \alpha. \neg(\alpha \models C))$ 
  defines Ts  $\equiv \text{map dual } Cs$ 
  shows dual (BigAnd' Cs) = BigOr' Ts  $\forall T \in \text{set } Ts. \text{is-conj } T \wedge (\exists \alpha. \alpha \models T)$ 
  unfolding atomize-conj Ts-def
  using assms(1)
proof (induction Cs)
  case Nil
  then show ?case

```

```

    by simp
next
case (Cons C Cs)

then have IH:
  dual (BigAnd' Cs) = BigOr' (map dual Cs)
  (∀ T ∈ set (map dual Cs). is-conj T ∧ (∃ α. α ⊨ T))
  by (auto simp add: Cons.IH Cons.prem)

show ?case
proof (intro conjI ballI)
  show dual (BigAnd' (C # Cs)) = BigOr' (map dual (C # Cs))
    by (cases Cs) (use IH(1) in simp-all)
next
have is-disj C
  using Cons.prem by simp
then have is-conj (dual C)
  by (rule is-conj-dual-if-is-disj)
then show ∧ T. T ∈ set (map dual (C # Cs)) ⇒ is-conj T
  using IH(2) by auto
next
have ∃ α. ¬ α ⊨ C
  using Cons.prem by simp
then have ∃ α. α ⊨ dual C
  using equiv-dual[of C, unfolded equiv-def] by simp
then show ∧ T. T ∈ set (map dual (C # Cs)) ⇒ ∃ α. α ⊨ T
  using IH(2) by auto
qed
qed

```

### 3.9 Formula Contains Atom

Should only be applied to a formula for which *is-nnf* holds.

```

fun cont-pos :: 'a formula ⇒ 'a ⇒ bool where
  cont-pos Bot l = False |
  cont-pos (Atom v) l = (v = l) |
  cont-pos (Not (Atom v)) l = False |
  cont-pos (Not φ) l = False |
  cont-pos (And φ ψ) l = (cont-pos φ l ∨ cont-pos ψ l) |
  cont-pos (Or φ ψ) l = (cont-pos φ l ∨ cont-pos ψ l) |
  cont-pos (Imp φ ψ) l = False

```

Should only be applied to a formula for which *is-nnf* holds.

```

fun cont-neg :: 'a formula ⇒ 'a ⇒ bool where
  cont-neg Bot l = False |
  cont-neg (Atom v) l = False |
  cont-neg (Not (Atom v)) l = (v = l) |
  cont-neg (Not φ) l = False |
  cont-neg (And φ ψ) l = (cont-neg φ l ∨ cont-neg ψ l) |

```

$cont\_neg (Or \ \varphi \ \psi) \ l = (cont\_neg \ \varphi \ l \ \vee \ cont\_neg \ \psi \ l) \ |$   
 $cont\_neg (Imp \ \varphi \ \psi) \ l = False$

Should only be applied to a formula for which *is-nnf* holds.

**fun** *cont* :: 'a formula  $\Rightarrow$  'a  $\Rightarrow$  bool **where**  
 $cont \ Bot \ l = False \ |$   
 $cont \ (Atom \ v) \ l = (v = l) \ |$   
 $cont \ (Not \ (Atom \ v)) \ l = (v = l) \ |$   
 $cont \ (Not \ \varphi) \ l = False \ |$   
 $cont \ (And \ \varphi \ \psi) \ l = (cont \ \varphi \ l \ \vee \ cont \ \psi \ l) \ |$   
 $cont \ (Or \ \varphi \ \psi) \ l = (cont \ \varphi \ l \ \vee \ cont \ \psi \ l) \ |$   
 $cont \ (Imp \ \varphi \ \psi) \ l = False$

**lemma** *impl-not-cont-pos*:  $\neg \ cont\_pos \ \varphi \ v \Longrightarrow \ cont\_neg \ \varphi \ v \ \vee \ \neg \ (cont \ \varphi \ v)$   
**by** (*induction*  $\varphi$ ) (*auto elim: cont.elims*)

**lemma** *impl-not-cont*:  $\neg \ cont \ \varphi \ v \Longrightarrow \neg \ cont\_pos \ \varphi \ v \ \wedge \ \neg \ cont\_neg \ \varphi \ v$   
**by** (*induction*  $\varphi$ ) (*auto elim: cont.elims*)

**lemma** *mem-atoms-if-cont-pos*:  
**assumes** *cont-pos*  $T \ v$   
**shows**  $v \in \ atoms \ T$   
**using** *assms* **by** (*induction*  $T \ v$  *rule: cont-pos.induct*) *auto*

**lemma**  
**assumes** *is-conj*  $\varphi$   
**shows**  
*not-sat-conj-neg-true*:  $\exists v. \ cont\_neg \ \varphi \ v \ \wedge \ \alpha \ v \Longrightarrow \neg (\alpha \models \varphi)$  **and**  
*not-sat-conj-pos-false*:  $\exists v. \ cont\_pos \ \varphi \ v \ \wedge \ \neg (\alpha \ v) \Longrightarrow \neg (\alpha \models \varphi)$   
**using** *assms*  
**by** (*induction*  $\varphi$ ) (*auto elim: cont-pos.elims is-lit-plus.elims*)

**lemma** *sat-conj-val-cont-ident*:  
**assumes**  $Val1 \models \varphi$  **and**  $\forall v \in \{v. \ cont \ \varphi \ v\}. \ Val1 \ v = Val2 \ v$  **and** *is-conj*  $\varphi$   
**shows**  $Val2 \models \varphi$   
**using** *assms*  
**by** (*induction*  $\varphi$ ) (*auto elim: cont-neg.elims is-lit-plus.elims*)

## 4 CNF to DNF

**proposition** *exp-blowup-from-Fn-to-BigOr'*:  
**fixes**  $n :: nat$  **and**  $Ts :: var \ formula \ list$   
**defines**  $\varphi \equiv Fn \ n$  **and**  $\psi \equiv BigOr' \ Ts$   
**assumes**  
*n-greater-0*:  $n > 0$  **and**  
*Ts-spec*:  $(\forall T \in \ set \ Ts. \ is\_conj \ T \ \wedge \ (\exists \alpha. \ \alpha \models T))$  **and**  
*equiv*  $\varphi \ \psi$   
**shows**  $sizef \ \psi \geq n * 2^{\wedge} n$   
**proof** –

```

have occ-var-bool-diff: cont-pos T (Var i False) ≠ cont-pos T (Var i True)
  if T ∈ set Ts and i ∈ {1..n}
  for T :: var formula and i :: nat
proof (rule notI)
  assume cont-pos T (Var i False) = cont-pos T (Var i True)

then consider
  (both-absent)  $\neg(\text{cont-pos } T \text{ (Var } i \text{ False)}) \neg(\text{cont-pos } T \text{ (Var } i \text{ True)}) \mid$ 
  (both-present) cont-pos T (Var i False) cont-pos T (Var i True)
  by satx

then show False
proof cases
  case both-absent
  then have  $\exists \alpha. \alpha \models T$ 
    by (simp add: <T ∈ set Ts> Ts-spec)
  then obtain ValsatT where Valsat: ValsatT ⊨ T
    by auto

  define  $\alpha$  where
     $\alpha = (\lambda v. (\text{if } v = \text{Var } i \text{ False} \vee v = \text{Var } i \text{ True} \text{ then False else ValsatT } v))$ 

  have  $\forall v \in \{v. \text{cont-pos } T \ v\}. \text{ValsatT } v = \alpha \ v$ 
    by (simp add: α-def both-absent)
  then have  $\alpha \models T$ 
    using not-sat-conj-neg-true impl-not-cont-pos sat-conj-val-cont-ident
    by (metis <T ∈ set Ts> α-def Ts-spec mem-Collect-eq Valsat)
  then have  $\exists \alpha. \alpha \models T \wedge \alpha \text{ (Var } i \text{ False)} = \text{False} \wedge \alpha \text{ (Var } i \text{ True)} = \text{False}$ 
    unfolding  $\alpha$ -def by auto
  then have  $\exists \alpha. \alpha \models \psi \wedge \neg(\alpha \models \varphi)$ 
    unfolding  $\psi$ -def  $\varphi$ -def
    using BigOr'-semantics <T ∈ set Ts> <i ∈ {1..n}> semantics-Fn-iff by metis
  then have  $\neg \text{equiv } \varphi \ \psi$ 
    unfolding equiv-def by auto
  then show False
    using  $\langle \text{equiv } \varphi \ \psi \rangle$  by contradiction
next
  case both-present
  then have  $\exists \alpha. \alpha \models T$ 
    by (simp add: <T ∈ set Ts> Ts-spec)
  then have  $\exists \alpha. \alpha \models T \wedge \alpha \text{ (Var } i \text{ False)} = \text{True} \wedge \alpha \text{ (Var } i \text{ True)} = \text{True}$ 
    using  $\langle T \in \text{set } Ts \rangle$  both-present not-sat-conj-pos-false Ts-spec by blast
  then have  $\exists \alpha. \alpha \models \psi \wedge \neg(\alpha \models \varphi)$ 
    unfolding  $\psi$ -def  $\varphi$ -def
    using BigOr'-semantics <T ∈ set Ts> <i ∈ {1..n}> semantics-Fn-iff by metis
  then have  $\neg \text{equiv } \varphi \ \psi$ 
    unfolding equiv-def by auto
  then show False
    using  $\langle \text{equiv } \varphi \ \psi \rangle$  by contradiction

```

```

qed
qed

have ex-T-cont-pos-var-eps:  $\exists T \in \text{set } Ts. \forall i \in \{1..n\}. \text{cont-pos } T ( \text{Var } i ( \text{nth } \text{eps } (i-1)))$ 
if length eps = n for eps :: bool list
proof (rule ccontr)
assume assm:  $\neg (\exists T \in \text{set } Ts. \forall i \in \{1..n\}. \text{cont-pos } T ( \text{Var } i ( \text{nth } \text{eps } (i-1))))$ 

define  $\alpha$  where
   $\alpha = (\lambda x. \text{case } x \text{ of } ( \text{Var } i b) \Rightarrow b = \text{nth } \text{eps } (i-1))$ 

have  $\alpha \models \varphi$ 
proof –
  have  $\alpha \models Fn\ n$ 
    using  $\alpha\text{-def}$  by (induction n) simp-all
  then show ?thesis
    by (simp add:  $\varphi\text{-def}$ )
qed

moreover have  $\neg(\alpha \models \psi)$ 
proof –
  have  $\forall T \in \text{set } Ts. \exists i \in \{1..n\}. \text{cont-pos } T ( \text{Var } i ( \neg(\text{nth } \text{eps } (i-1))))$ 
    using assm occ-var-bool-diff by (metis (full-types))
  then have  $\forall T \in \text{set } Ts. \neg(\alpha \models T)$ 
    by (metis (mono-tags, lifting) Ts-spec  $\alpha\text{-def not-sat-conj-pos-false var.case}$ )
  then show ?thesis
    unfolding  $\psi\text{-def}$  by auto
qed

ultimately have  $\neg \text{equiv } \varphi \ \psi$ 
  by (auto simp add: equiv-def)

then show False
  using  $\langle \text{equiv } \varphi \ \psi \rangle$  by contradiction
qed

have n-le-card-atoms:  $n \leq \text{card } (\text{atoms } T)$  if T-in:  $T \in \text{set } Ts$  for  $T$ 
  using card-le-card-if-mem-imp-ex-mem[of  $\{1..n\}$  atoms T Var, simplified]
  using occ-var-bool-diff[rule-format, OF T-in]
  by (metis One-nat-def atLeastAtMost-iff mem-atoms-if-cont-pos)

define conj-of-eps  $:: \text{bool list} \Rightarrow \text{var formula}$  where
  conj-of-eps =  $(\lambda \text{eps.}$ 
     $(\text{SOME } T. T \in \text{set } Ts \wedge (\forall i \in \{1..(\text{length } \text{eps})\}. \text{cont-pos } T ( \text{Var } i ( \text{nth } \text{eps } (i - 1))))))$ )

have T-of-conj-of-eps-in-Ts: conj-of-eps eps  $\in \text{set } Ts$  if length eps = n for eps
  unfolding conj-of-eps-def

```

```

by (smt (verit, best) ex-T-cont-pos-var-eps that verit-sko-ex1)

have  $2^{\hat{n}} = \text{card } \{ \text{eps} :: \text{bool list. length eps} = n \}$ 
  using card-lists-length-eq[of UNIV :: bool set n, simplified, symmetric] .

also have  $\dots \leq \text{card } (\text{set } Ts)$ 
proof (rule card-inj-on-le[of conj-of-eps])
  show inj-on conj-of-eps {eps. length eps = n}
  proof (rule inj-onI)
    fix xs ys :: bool list
    assume
      xs-in: xs  $\in$  {eps. length eps = n} and
      ys-in: ys  $\in$  {eps. length eps = n}

    have length xs = n
      using xs-in by simp

    moreover have length ys = n
      using ys-in by simp

    ultimately have length xs = length ys
      by simp

    show conj-of-eps xs = conj-of-eps ys  $\implies$  xs = ys
    proof (erule contrapos-pp)
      assume xs  $\neq$  ys
      then obtain i where i < n and nth xs i  $\neq$  nth ys i
        using  $\langle \text{length xs} = \text{length ys} \rangle$ 
        using  $\langle \text{length xs} = n \rangle$  nth-equalityI by blast

      have cont-pos (conj-of-eps xs) (Var (Suc i) (xs ! i))
      proof -
        have conj-of-eps xs  $\in$  set Ts  $\wedge$ 
          ( $\forall i \in \{1..n\}. \text{cont-pos } (\text{conj-of-eps } xs) (\text{Var } i (xs ! (i - 1)))$ )
          by (smt (verit, ccfv-SIG)  $\langle \text{length xs} = n \rangle$  conj-of-eps-def
            ex-T-cont-pos-var-eps someI-ex)
        then show ?thesis
          using  $\langle i < n \rangle$  by force
      qed

      moreover have cont-pos (conj-of-eps ys) (Var (Suc i) (ys ! i))
      proof -
        have conj-of-eps ys  $\in$  set Ts  $\wedge$ 
          ( $\forall i \in \{1..n\}. \text{cont-pos } (\text{conj-of-eps } ys) (\text{Var } i (ys ! (i - 1)))$ )
          by (smt (verit, ccfv-SIG)  $\langle \text{length ys} = n \rangle$  conj-of-eps-def
            ex-T-cont-pos-var-eps someI-ex)
        then show ?thesis
          using  $\langle i < n \rangle$  by force
      qed
    qed
  qed

```

**moreover have**  
*cont-pos (conj-of-eps ys) (Var (Suc i) False) ≠*  
*cont-pos (conj-of-eps ys) (Var (Suc i) True)*  
**using**  $\langle i < n \rangle$  *occ-var-bool-diff*[OF *T-of-conj-of-eps-in-Ts*[OF  $\langle \text{length } ys = n \rangle$ ], of *Suc i*]  
**by** *simp*

**ultimately show** *conj-of-eps xs ≠ conj-of-eps ys*  
**using**  $\langle xs ! i \neq ys ! i \rangle$   
**by** (*metis (mono-tags)*)  
**qed**  
**qed**  
**next**  
**show** *conj-of-eps* ‘ $\{ \text{eps. length } \text{eps} = n \} \subseteq \text{set } Ts$ ’  
**using** *T-of-conj-of-eps-in-Ts* **by** *auto*  
**next**  
**show** *finite (set Ts)*  
**by** *simp*  
**qed**

**also have**  $\dots \leq \text{length } Ts$   
**using** *card-length*[of *Ts*] .

**finally have**  $\text{length } Ts \geq 2^n$  .

**moreover have**  $\forall T \in \text{set } Ts. \text{size } T \geq n$   
**using** *n-le-card-atoms card-atoms-le-sizef*  
**using** *le-trans* **by** *blast*

**ultimately show** *?thesis*  
**unfolding** *ψ-def*  
**using** *exp-sizef*[OF *n-greater-0*]  
**by** (*metis mult.commute*)  
**qed**

**lemma** *ex-equiv-disj-list-if-is-dnf*:  
**fixes**  $\varphi :: 'a \text{ formula}$   
**assumes** *dnf: is-dnf*  $\varphi$  **and** *sat*:  $\exists \alpha. \alpha \models \varphi$   
**shows**  $\exists (Ts :: 'a \text{ formula list}). \text{equiv } \varphi (\text{BigOr}' Ts) \wedge$   
 $\text{size } (\text{BigOr}' Ts) \leq \text{size } \varphi \wedge$   
 $(\forall T \in \text{set } Ts. \text{is-conj } T) \wedge$   
 $(\forall T \in \text{set } Ts. \exists \alpha. \alpha \models T)$

**proof** –  
**have** *bex-sat*:  $\exists T \in \text{set } (\text{undisj } \varphi). \exists \alpha. \alpha \models T$   
**using** *sat*  
**by** (*metis equiv-def BigOr'-semantics equiv-BigOr'-undisj-if-dnf*)

**define**  $Ts :: 'a \text{ formula list}$  **where**

```

Ts = filter (λT. ∃α. α ⊨ T) (undisj φ)

show ?thesis
proof (intro exI[of - Ts] conjI ballI)
  have equiv φ (BigOr' (undisj φ))
    using equiv-BigOr'-undisj-if-dnf[symmetric] .
  then show equiv φ (BigOr' Ts)
    unfolding Ts-def
    by (auto simp add: equiv-def)
next
  have size (BigOr' (undisj φ)) = size φ
    using size-BigOr'-undisj .
  then show size (BigOr' Ts) ≤ size φ
    unfolding Ts-def
    using size-BigOr'-filter-le-if[OF bex-sat]
    by presburger
next
  show ∧T. T ∈ set Ts ⇒ is-conj T
    using ball-undisj-is-conj[OF dnf]
    by (simp add: Ts-def)
next
  show ∧T. T ∈ set Ts ⇒ ∃α. α ⊨ T
    by (simp add: Ts-def)
qed
qed

theorem exp-blowup-from-CNF-to-DNF:
  fixes n :: nat
  shows ∃(φ :: var formula).
    is-cnf φ ∧
    size φ = 10 * n + 2 ∧
    (∀(ψ :: var formula). equiv φ ψ → is-dnf ψ → size ψ ≥ n * 2 ^ n)
proof (cases n)
  case 0
  then show ?thesis
    using is-cnf-Fn size-Fn by fastforce
next
  case (Suc n')

  then have 0 < n
    by presburger

  show ?thesis
  proof (intro exI conjI allI impI)
    show is-cnf (Fn n)
      using is-cnf-Fn .
  next
    show size (Fn n) = 10 * n + 2
      using size-Fn .
  next

```

```

next
  fix  $\psi :: \text{var formula}$ 
  assume equiv (Fn n)  $\psi$ 
  then have sat- $G_n$ :  $\exists \alpha. \alpha \models \psi$ 
    unfolding equiv-def
    using Fn-sat[of n] by metis

  assume is-dnf  $\psi$ 
  then obtain Ts :: var formula list where
     $\forall T \in \text{set } Ts. \text{is-conj } T$  and
     $\forall T \in \text{set } Ts. \exists \alpha. \alpha \models T$  and
    equiv  $\psi$  (BigOr' Ts) and
    size: size (BigOr' Ts)  $\leq \text{size } \psi$ 
    using ex-equiv-disj-list-if-is-dnf[OF - sat- $G_n$ ] by metis

  moreover have equiv (Fn n) (BigOr' Ts)
    using equiv-transitive[OF  $\langle \text{equiv} (Fn n) \psi \rangle \langle \text{equiv } \psi (BigOr' Ts) \rangle$ ].

  ultimately have  $n * 2^{\wedge} n \leq \text{sizef} (BigOr' Ts)$ 
    using exp-blowup-from-Fn-to-BigOr'[OF  $\langle 0 < n \rangle$ , of Ts] by metis

  then show  $n * 2^{\wedge} n \leq \text{size } \psi$ 
    using size sizef-le-size[of BigOr' Ts] by presburger
qed
qed

```

## 5 DNF to CNF

```

proposition exp-blowup-from-dual-Fn-to-BigAnd':
  fixes  $n :: \text{nat}$  and Cs :: var formula list
  defines  $\varphi \equiv \text{dual} (Fn n)$  and  $\psi \equiv \text{BigAnd}' Cs$ 
  assumes
    n-greater-0:  $n > 0$  and
    Cs-spec:  $(\forall C \in \text{set } Cs. \text{is-disj } C \wedge \neg(\forall \alpha. \alpha \models C))$  and
    equiv  $\varphi \psi$ 
  shows sizef  $\psi \geq n * 2^{\wedge} n$ 
proof –
  have G-in-cnf: is-cnf  $\psi$ 
    using  $\psi$ -def Cs-spec is-cnf-BigAnd' by auto
  have G-in-nnf: is-nnf  $\psi$ 
    using G-in-cnf cnf-in-nnf by auto

  show ?thesis
proof (rule ccontr)
  assume  $\neg n * 2^{\wedge} n \leq \text{sizef } \psi$ 
  then have sizef  $\psi < n * 2^{\wedge} n$ 
    by presburger
  then have sizef (dual  $\psi$ )  $< n * 2^{\wedge} n$ 
    using sizef-dual[of  $\psi$ ] by presburger

```

**obtain**  $Ts$  **where**  
 $dual\ \psi = BigOr'\ Ts\ \forall T \in set\ Ts.\ is-conj\ T \wedge (\exists\ \alpha.\ \alpha \models T)$   
**using**  $\psi-def\ Cs-spec\ dual-conj-of-disjs-is-disj-of-conjs$  **by** *metis*

**moreover have**  $equiv\ (Fn\ n)\ (dual\ \psi)$

**proof** –

**have**  $equiv\ (dual\ \psi)\ (Not\ \psi)$

**using** *equiv-dual* .

**also have**  $equiv\ \dots\ (Not\ (dual\ (Fn\ n)))$

**unfolding** *equiv-Not-left-Not-right*

**using**  $\langle equiv\ \varphi\ \psi \rangle [symmetric,\ unfolded\ \varphi-def]$  .

**also have**  $equiv\ \dots\ (Not\ (Not\ (Fn\ n)))$

**unfolding** *equiv-Not-left-Not-right*

**using** *equiv-dual* .

**also have**  $equiv\ \dots\ (Fn\ n)$

**unfolding** *equiv-Not-Not-left*

**using** *equiv-reflexive* .

**finally show** *?thesis*

**by** *(rule equiv-symmetric)*

**qed**

**ultimately have**  $n * 2^{\hat{n}} \leq sizef\ (dual\ \psi)$

**using** *exp-blowup-from-Fn-to-BigOr'*[of  $n\ Ts$ ] **by** *force*

**then show** *False*

**using**  $\langle sizef\ (dual\ \psi) < n * 2^{\hat{n}} \rangle$  **by** *presburger*

**qed**

**qed**

**lemma** *ex-equiv-conj-list-if-is-cnf*:

**fixes**  $\varphi :: 'a\ formula$

**assumes** *cnf: is-cnf  $\varphi$*

**shows**  $\exists (Cs :: 'a\ formula\ list).\ equiv\ \varphi\ (BigAnd'\ Cs) \wedge$

$sizef\ (BigAnd'\ Cs) \leq sizef\ \varphi \wedge$

$(\forall C \in set\ Cs.\ is-disj\ C) \wedge$

$(\forall C \in set\ Cs.\ \neg \models C)$

**proof** –

**define**  $Cs :: 'a\ formula\ list$  **where**

$Cs = filter\ (\lambda C.\ \neg \models C)\ (unconj\ \varphi)$

**show** *?thesis*

**proof** *(intro exI[of - Cs] conjI ballI)*

**have**  $equiv\ \varphi\ (BigAnd'\ (unconj\ \varphi))$

**using** *equiv-BigAnd'-unconj-if-cnf[symmetric]* .

**then show**  $equiv\ \varphi\ (BigAnd'\ Cs)$

**unfolding** *Cs-def*

**by** *(auto simp add: equiv-def)*

**next**

**have**  $sizef\ (BigAnd'\ (unconj\ \varphi)) = sizef\ \varphi$

```

    using sizef-BigAnd'-unconj .
  then show sizef (BigAnd' Cs) ≤ sizef φ
    unfolding Cs-def
    using sizef-BigAnd'-filter-le[of λC. ¬ ⊨ C unconj φ]
    by presburger
next
  show ∧ C. C ∈ set Cs ⇒ is-disj C
    using ball-unconj-is-disj[OF cnf]
    by (simp add: Cs-def)
next
  show ∧ C. C ∈ set Cs ⇒ ¬ ⊨ C
    by (simp add: Cs-def)
qed
qed

theorem exp-blowup-from-DNF-to-CNF:
  fixes n :: nat
  shows ∃ (φ :: var formula).
    is-dnf φ ∧
    size φ = 10 * n + 1 ∧
    (∀ (ψ :: var formula). equiv φ ψ → is-cnf ψ → size ψ ≥ n * 2 ^ n)
proof (cases n)
  case 0
  then show ?thesis
    using is-dnf-dual-Fn size-dual-Fn
    by fastforce
next
  case (Suc n')

  then have n > 0
    by presburger

  show ?thesis
  proof (intro exI conjI allI impI)
    show is-dnf (dual (Fn n))
      using is-dnf-dual-Fn .
    next
      show size (dual (Fn n)) = 10 * n + 1
        using size-dual-Fn .
    next
      fix ψ :: var formula
      assume equiv (dual (Fn n)) ψ
      assume is-cnf ψ

  then obtain Cs :: var formula list where
    ∀ C ∈ set Cs. is-disj C and
    ∀ C ∈ set Cs. ¬ ⊨ C and
    equiv ψ (BigAnd' Cs) and
    sizef: sizef (BigAnd' Cs) ≤ sizef ψ

```

```

using ex-equiv-conj-list-if-is-cnf[of  $\psi$ ] by metis

moreover have equiv (dual (Fn  $n$ )) (BigAnd' Cs)
using equiv-transitive[OF  $\langle$ equiv (dual (Fn  $n$ ))  $\psi$  $\rangle$   $\langle$ equiv  $\psi$  (BigAnd' Cs) $\rangle$ ].

ultimately have  $n * 2^{\wedge} n \leq \text{sizef}$  (BigAnd' Cs)
using exp-blowup-from-dual-Fn-to-BigAnd'[OF  $\langle 0 < n \rangle$ , of Cs] by metis

then show  $n * 2^{\wedge} n \leq \text{size}$   $\psi$ 
using sizef sizef-le-size[of  $\psi$ ] by presburger
qed
qed

end

```

## References

- [1] Jan Johannsen. SAT Solving: Skript zur Vorlesung im WS 2023/24, 2023.
- [2] Leon Raffael Schulz. Formalisierung einer unteren Schranke an die Formelgröße in der Aussagenlogik mit Isabelle, 2026.