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

Leon Raffael Schulz Martin Desharnais-Schäfer
Jan Johannsen

June 16, 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 Schulzs 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

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 *card* (*Domain A*) ≤ *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** $\mathcal{X} :: 'a\ set$ **and** $\mathcal{Y} :: 'c\ set$
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-$\mathcal{Y}$: $\bigwedge x.\ x \in \mathcal{X} \Longrightarrow \exists\, y.\ f\ x\ y \in \mathcal{Y}$
shows *card* $\mathcal{X} \leq$ *card* $\mathcal{Y}$
proof −
have *f-XY-subset*: $(\lambda(x,\ y).\ f\ x\ y)\ `\ XY \subseteq \mathcal{Y}$
using *XY-def* **by** *auto*

then have *finite* $((\lambda(x,\ y).\ f\ x\ y)\ `\ XY)$
using ‹*finite* $\mathcal{Y}$› **by** (*rule finite-subset*)

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

moreover have *Domain XY* = $\mathcal{X}$
unfolding *XY-def*
using *ex-in-*$\mathcal{Y}$
by (*simp add*: *equalityI subsetI*)

ultimately have *card* $\mathcal{X} \leq$ *card XY*
using *card-Domain-le* **by** *iprover*

also have ... ≤ *card* $\mathcal{Y}$
using *inj-on-iff-card-le*[*OF* ‹*finite XY*› ‹*finite* $\mathcal{Y}$›]
using *f-XY-subset f-inj* **by** *blast*

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

2 Move to `Propositional_Proof_Systems`

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

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

lemma *cnf-in-nnf*: *is-cnf* $\varphi \Longrightarrow$ *is-nnf* φ
by (*induction* φ) (*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 φ $\psi \longleftrightarrow (\forall \alpha.\ (\alpha \models \varphi) \longleftrightarrow (\alpha \models \psi))$

lemma *equiv-reflexive*: $\bigwedge\varphi$. *equiv* φ φ
unfolding *equiv-def* **by** *simp*

lemma *equiv-symmetric*[*sym*]: $\bigwedge\varphi$ ψ. *equiv* φ $\psi \Longrightarrow$ *equiv* ψ φ
unfolding *equiv-def* **by** *simp*

lemma *equiv-transitive*[*trans*]: $\bigwedge\xi$ φ ψ. *equiv* ξ $\varphi \Longrightarrow$ *equiv* φ $\psi \Longrightarrow$ *equiv* ξ ψ
unfolding *equiv-def* **by** *simp*

lemma *equiv-Not-left-Not-right*[*simp*]: $\bigwedge\varphi$ ψ. *equiv* (*Not* φ) (*Not* ψ) $\longleftrightarrow$ *equiv* φ ψ
unfolding *equiv-def* **by** *simp-all*

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

lemma *equiv-Not-Not-right*[*simp*]: $\bigwedge\varphi$ ψ. *equiv* φ (*Not* (*Not* ψ)) $\longleftrightarrow$ *equiv* φ ψ
unfolding *equiv-def* **by** *simp-all*

3.2 Conjunctive Normal Form

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

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

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

lemma *length-unconj*: *length* (*unconj* φ) = *count-And* φ + *1*
by (*induction* φ) *simp-all*

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

3.3 Disjunctive Normal Form

fun *is-conj* :: *'a formula* ⇒ *bool* **where**
is-conj (*And* φ ψ) $\longleftrightarrow$ (*is-lit-plus* φ ∧ *is-conj* ψ) |
is-conj φ $\longleftrightarrow$ *is-lit-plus* φ

fun *is-dnf* :: *'a formula* ⇒ *bool* **where**
is-dnf (*Or* φ ψ) $\longleftrightarrow$ (*is-dnf* φ ∧ *is-dnf* ψ) |
is-dnf φ $\longleftrightarrow$ *is-conj* φ

lemma *conj-is-dnf*: *is-conj* φ $\Longrightarrow$ *is-dnf* φ
by (*induction* φ) *auto*

fun *undisj* :: *'a formula* ⇒ *'a formula list* **where**
undisj (*Or* φ ψ) = *undisj* φ @ *undisj* ψ |
undisj φ = [φ]

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

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

fun *count-Or* :: *'a formula* ⇒ *nat* **where**
count-Or (*Or* φ ψ) = *count-Or* φ + *count-Or* ψ + *1* |
count-Or - = *0*

lemma *length-undisj*: *length* (*undisj* φ) = *count-Or* φ + *1*
by (*induction* φ) *simp-all*

3.4 Big Conjunction

fun *BigAnd'* :: *'a formula list* ⇒ *'a formula* **where**
BigAnd' [] = (¬⊥) |
BigAnd' [*F*] = *F* |

BigAnd′ (F # Fs) = F ∧ BigAnd′ Fs

lemma *atoms-BigAnd′*[*simp*]: *atoms (BigAnd′ Fs) = ⋃(atoms ‘ set Fs)*
by (*induction Fs rule*: *BigAnd′.induct*) *simp-all*

lemma *BigAnd′-semantics*[*simp*]: *A ⊨ BigAnd′ Ts ⟷ (∀ f ∈ set Ts. A ⊨ f)*
by (*induction Ts rule*: *BigAnd′.induct*) *simp-all*

lemma *is-cnf-BigAnd′*: *(∀ C ∈ set Cs. is-disj C ∧ ¬(∀ α. α ⊨ C)) ⟹ 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 ⇒ ′a formula* **where**
BigOr′ Nil = ⊥ |
BigOr′ [F] = F |
BigOr′ (F#Fs) = F ∨ BigOr′ Fs

lemma *atoms-BigOr′*[*simp*]: *atoms (BigOr′ Fs) = ⋃(atoms ‘ set Fs)*
by (*induction Fs rule*: *BigOr′.induct*) *simp-all*

lemma *BigOr′-semantics*[*simp*]: *A ⊨ BigOr′ Ts ⟷ (∃ f ∈ set Ts. A ⊨ f)*
by (*induction Ts rule*: *BigOr′.induct*) *simp-all*

lemma *is-dnf-BigOr′*: *(∀ T ∈ set Ts. is-conj T ∧ (∃ α. α ⊨ T)) ⟹ 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 *φ* :: *′a formula*
shows *equiv (BigOr′ (undisj φ)) φ*
using *equiv-BigOr′-append*

by (*induction* φ *rule*: *is-dnf.induct*) (*auto simp add*: *equiv-def*)

lemma *equiv-BigAnd'-unconj-if-cnf*:
fixes φ :: *'a formula*
shows *equiv* (*BigAnd'* (*unconj* φ)) φ
proof (*induction* φ *rule*: *unconj.induct*)
case (*1* φ ψ)
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 ¬ when calculating the size.

fun *sizef* :: *'a formula* ⇒ *nat* **where**
sizef Bot = *1* |
sizef (*Atom a*) = *1* |
sizef (*Not* φ) = *sizef* φ |
sizef (*And* φ ψ) = *sizef* φ + *sizef* ψ + *1* |
sizef (*Or* φ ψ) = *sizef* φ + *sizef* ψ + *1* |
sizef (*Imp* φ ψ) = *sizef* φ + *sizef* ψ + *1*

lemma *Suc-0-le-sizef*[*simp*]: *Suc 0* ≤ *sizef* φ
by (*induction* φ) *simp-all*

lemma *Suc-0-le-size*[*simp*]:
fixes φ :: *'a formula*
shows *Suc 0* ≤ *size* φ
by (*induction* φ) *simp-all*

lemma *sizef-le-size*: *sizef* φ ≤ *size* φ
by (*induction* φ) *simp-all*

lemma *card-atoms-le-sizef*: *card* (*atoms* φ) ≤ *sizef* φ
proof (*induction* φ)
case (*And F1 F2*)
have *card* (*atoms* (*F1* ∧ *F2*)) = *card* (*atoms F1* ∪ *atoms F2*)
by *simp*
also have ... ≤ *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 ... ≤ *Suc* (*sizef F1* + *sizef F2*)
using *And.IH* **by** *presburger*
also have ... = *sizef* (*F1* ∧ *F2*)
by *simp*
finally show *?case*
by *presburger*

next
case (*Or F1 F2*)
have *card* (*atoms* (*F1* ∨ *F2*)) = *card* (*atoms F1* ∪ *atoms F2*)
by *simp*
also have ... ≤ *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 ... ≤ *Suc* (*sizef F1* + *sizef F2*)
using *Or.IH* **by** *presburger*
also have ... = *sizef* (*F1* ∨ *F2*)
by *simp*
finally show *?case*
by *presburger*
next
case (*Imp F1 F2*)
have *card* (*atoms* (*F1* → *F2*)) = *card* (*atoms F1* ∪ *atoms F2*)
by *simp*
also have ... ≤ *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 ... ≤ *Suc* (*sizef F1* + *sizef F2*)
using *Imp.IH* **by** *presburger*
also have ... = *sizef* (*F1* → *F2*)
by *simp*
finally show *?case*
by *presburger*
qed *simp-all*

lemma *card-atoms-le-size*: *card* (*atoms* φ) ≤ *size* φ
by (*metis card-atoms-le-sizef le-trans sizef-le-size*)

lemma *aux-exp-sizef*: *length Ts* = *n* ⟹ ∀ *T* ∈ *set Ts*. *sizef T* ≥ *m* ⟹ *sizef* (*BigOr′ Ts*) ≥ *n* ∗ *m*
by (*induction Ts arbitrary*: *m n rule*: *BigOr′.induct*; *fastforce*)

lemma *aux-exp-size*: *length Ts* = *n* ⟹ ∀ *T* ∈ *set Ts*. *size T* ≥ *m* ⟹ *size* (*BigOr′ Ts*) ≥ *n* ∗ *m*
by (*induction Ts arbitrary*: *m n rule*: *BigOr′.induct*; *fastforce*)

lemma *exp-sizef*:
assumes *n* > *0* **and** *length Ts* ≥ *2^n* **and** ∀ *T* ∈ *set Ts*. *sizef T* ≥ *m*
shows *sizef* (*BigOr′ Ts*) ≥ *2^n* ∗ *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 2 ^ n ≤ length Ts ∨ 2 ^ n = Suc (length Ts)
    unfolding Ts-def using 3.prems by auto
  then have 2 ^ n * m ≤ Suc (sizef T + sizef (BigOr' Ts))
  proof (elim disjE)
    assume asm1: 2 ^ n ≤ length Ts

    have 2 ^ n * m ≤ sizef (BigOr' (T' # Ts'))
    proof (rule 3.IH)
      show 0 < n
        by (metis 3.prems(1))
    next
      show 2 ^ n ≤ length (T' # Ts')
        using Ts-def asm1 by blast
    next
      show (∀ T∈set (T' # Ts'). m ≤ sizef T)
        by (simp add: 3.prems(3) Ts-def)
    qed

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

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

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

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

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

lemma exp-size:
  assumes n > 0 and length Ts ≥ 2^n and ∀ T ∈ set Ts. size T ≥ m
```

```
  shows size (BigOr' Ts) ≥ 2^n * 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 2 ^ n ≤ length Ts ∨ 2 ^ n = Suc (length Ts)
    unfolding Ts-def using 3.prems by auto
  then have 2 ^ n * m ≤ Suc (size T + size (BigOr' Ts))
  proof (elim disjE)
    assume asm1: 2 ^ n ≤ length Ts

    have 2 ^ n * m ≤ size (BigOr' (T' # Ts'))
    proof (rule 3.IH)
      show 0 < n
        by (metis 3.prems(1))
    next
      show 2 ^ n ≤ length (T' # Ts')
        using Ts-def asm1 by blast
    next
      show (∀ T∈set (T' # Ts'). m ≤ size T)
        by (simp add: 3.prems(3) Ts-def)
    qed

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

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

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

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

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

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

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

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

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

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

lemma *sizef-conv-sum-list-undisj*: *sizef* φ = *sum-list* (*map sizef* (*undisj* φ)) + *count-Or* φ
by (*induction* φ) *simp-all*

lemma *size-conv-sum-list-undisj*: *size* φ = *sum-list* (*map size* (*undisj* φ)) + *count-Or* φ
by (*induction* φ) *simp-all*

lemma *sizef-conv-sum-list-unconj*: *sizef* φ = *sum-list* (*map sizef* (*unconj* φ)) + *count-And* φ
by (*induction* φ) *simp-all*

lemma *size-conv-sum-list-unconj*: *size* φ = *sum-list* (*map size* (*unconj* φ)) + *count-And* φ
by (*induction* φ) *simp-all*

lemma *sizef-BigOr'-undisj*:
fixes φ :: *'a formula*
shows *sizef* (*BigOr'* (*undisj* φ)) = *sizef* φ
proof −
have *sizef* φ + *1* = *sum-list* (*map sizef* (*undisj* φ)) + *count-Or* φ + *1*
using *sizef-conv-sum-list-undisj*[*of* φ] **by** *presburger*

also have ... = *sum-list* (*map sizef* (*undisj* φ)) + *length* (*undisj* φ)
using *length-undisj*[*of* φ] **by** *presburger*

also have ... = *sizef* (*BigOr'* (*undisj* φ)) + *1*
using *sizef-BigOr'*[*of undisj* φ, *simplified*] **by** *presburger*

```
  finally show ?thesis
    by presburger
qed

lemma size-BigOr'-undisj:
  fixes φ :: 'a formula
  shows size (BigOr' (undisj φ)) = size φ
proof −
  have size φ + 1 = sum-list (map size (undisj φ)) + count-Or φ + 1
    using size-conv-sum-list-undisj[of φ] by presburger

  also have ... = sum-list (map size (undisj φ)) + length (undisj φ)
    using length-undisj[of φ] by presburger

  also have ... = size (BigOr' (undisj φ)) + 1
    using size-BigOr'[of undisj φ, simplified] by presburger

  finally show ?thesis
    by presburger
qed

lemma sizef-BigAnd'-unconj:
  fixes φ :: 'a formula
  shows sizef (BigAnd' (unconj φ)) = sizef φ
proof −
  have sizef φ + 1 = sum-list (map sizef (unconj φ)) + count-And φ + 1
    using sizef-conv-sum-list-unconj[of φ] by presburger

  also have ... = sum-list (map sizef (unconj φ)) + length (unconj φ)
    using length-unconj[of φ] by presburger

  also have ... = sizef (BigAnd' (unconj φ)) + 1
    using sizef-BigAnd'[of unconj φ, simplified] by presburger

  finally show ?thesis
    by presburger
qed

lemma size-BigAnd'-unconj:
  fixes φ :: 'a formula
  shows size (BigAnd' (unconj φ)) = size φ
proof −
  have size φ + 1 = sum-list (map size (unconj φ)) + count-And φ + 1
    using size-conv-sum-list-unconj[of φ] by presburger

  also have ... = sum-list (map size (unconj φ)) + length (unconj φ)
    using length-unconj[of φ] by presburger

  also have ... = size (BigAnd' (unconj φ)) + 1
```

```
    using size-BigAnd'[of unconj φ, simplified] by presburger

  finally show ?thesis
    by presburger
qed

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

  have sizef (BigOr' (filter P (F # v # va))) ≤ 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 ... ≤ 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 ... ≤ sizef (BigOr' (F # v # va))
      using 3 by simp
    finally show ?thesis .
  qed

  finally show ?case .
qed

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

```
  assumes ∃ x ∈ set xs. P x
  shows size (BigOr' (filter P xs)) ≤ size (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: size (BigOr' (filter P (v # va))) ≤ size (BigOr' (v # va))
    by (metis BigOr'.simps(1) One-nat-def empty-filter-conv formula.size(8) for-
mula.size-neq
        leI less-one)

  have size (BigOr' (filter P (F # v # va))) ≤ size (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 ... ≤ size (BigOr' (F # v # va))
  proof (cases filter P (v # va))
    case Nil
    then show ?thesis
      by simp
  next
    case (Cons G Gs)
    then have size (BigOr' (F # filter P (v # va))) =
      size F + size (BigOr' (filter P (v # va))) + 1
      by simp
    also have ... ≤ size (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 = (¬⊥)|
  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*: $\exists\,\alpha.\ \alpha \models Fn\ n$
proof −
  define α **where** $\alpha = (\lambda x.\ case\ x\ of\ (Var\ i\ b) \Rightarrow b)$
  then have $\alpha \models Fn\ n$
    by (*induction n*; *simp*)
  then show *?thesis*
    by *auto*
qed

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

3.8 Dual Function

primrec *dual* :: *'a formula* ⇒ *'a formula* **where**
  dual Bot = *Not Bot* |
  dual (*Atom v*) = *Not* (*Atom v*) |

```
  dual (Not v) = v |
  dual (And φ ψ) = Or (dual φ) (dual ψ) |
  dual (Or φ ψ) = And (dual φ) (dual ψ) |
  dual (Imp φ ψ) = And φ (dual ψ)

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 φ) = sizef φ
  by (induction φ) simp-all

lemma size-dual-le: size (dual φ) ≤ 2 * size φ
  by (induction φ) simp-all

lemma equiv-dual: equiv (dual φ) (Not φ)
  by (induction φ) (simp-all add: equiv-def)

lemma is-disj-dual-if-is-conj: is-conj φ ⟹ is-disj (dual φ)
proof (induction φ)
  case (Not φ)
  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 φ ⟹ is-conj (dual φ)
proof (induction φ)
  case (Not φ)
  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 φ ⟹ is-dnf (dual φ)
```

proof (*induction* φ)
  case (*Not* φ)
  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* $\varphi \Longrightarrow$ *is-cnf* (*dual* φ)
proof (*induction* φ)
  case (*Not* φ)
  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* $\varphi \Longrightarrow \exists\,\alpha.\ \neg\ \alpha \models \varphi \Longrightarrow \exists\,\alpha.\ \alpha \models$ *dual* φ
proof (*induction* φ)
  case (*Or F1 F2*)
  have *F-is-nnf*: *is-nnf* (*F1* $\vee$ *F2*)
    using *Or.prems*(*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 α **where** α-*def*: $\neg\ \alpha \models$ *F1* $\vee$ *F2*
    using *Or.prems*(*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 $\forall\ C \in$ *set Cs*. *is-disj C* $\wedge$ ($\exists\ \alpha.\ \neg(\alpha \models C)$)
  defines *Ts* $\equiv$ *map dual Cs*
  shows *dual* (*BigAnd*$'$ *Cs*) = *BigOr*$'$ *Ts* $\forall\, T \in$*set Ts*. *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.prems)

  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.prems 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.prems 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* φ ψ) l = (*cont-neg* φ l $\vee$ *cont-neg* ψ l) |
cont-neg (*Imp* φ ψ) 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* φ) *l* = *False* |
cont (*And* φ ψ) *l* = (*cont* φ *l* $\vee$ *cont* ψ *l*) |
cont (*Or* φ ψ) *l* = (*cont* φ *l* $\vee$ *cont* ψ *l*) |
cont (*Imp* φ ψ) *l* = *False*

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

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

lemma *sat-conj-val-cont-ident*:
assumes *Val1* $\models$ φ **and** $\forall$ *v* $\in$ {*v*. *cont* φ *v*}. *Val1 v* = *Val2 v* **and** *is-conj* φ
shows *Val2* $\models$ φ
using *assms*
by (*induction* φ) (*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 φ ψ
shows *sizef* $\psi \geq n*2\hat{}n$
proof −

have *occ-var-bool-diff*: *cont-pos T* (*Var i False*) ≠ *cont-pos T* (*Var i True*)
if $T \in$ *set Ts* **and** $i \in \{1..n\}$
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*) ¬(*cont-pos T* (*Var i False*)) ¬(*cont-pos T* (*Var i True*)) |
(*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 ∃ α. $\alpha \models T$
by (*simp add*: ‹*T* ∈ *set Ts*› *Ts-spec*)
then obtain *ValsatT* **where** *Valsat*: *ValsatT* ⊨ *T*
by *auto*

define α **where**
α = (λ *v*. (*if v* = *Var i False* ∨ *v* = *Var i True then False else ValsatT v*))

have ∀ *v* ∈ {*v*. *cont-pos T v*}. *ValsatT v* = α *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 ∃ α. $\alpha \models T \wedge \alpha$ (*Var i False*) = *False* ∧ α (*Var i True*) = *False*
unfolding α-*def* **by** *auto*
then have ∃ α. $\alpha \models \psi \wedge \neg(\alpha \models \varphi)$
unfolding ψ-*def* φ-*def*
using *BigOr′-semantics* ‹*T* ∈ *set Ts*› ‹*i* ∈ {*1..n*}› *semantics-Fn-iff* **by** *metis*
then have ¬ *equiv* φ ψ
unfolding *equiv-def* **by** *auto*
then show *False*
using ‹*equiv* φ ψ› **by** *contradiction*
next
case *both-present*
then have ∃ α. $\alpha \models T$
by (*simp add*: ‹*T* ∈ *set Ts*› *Ts-spec*)
then have ∃ α. $\alpha \models T \wedge \alpha$ (*Var i False*) = *True* ∧ α (*Var i True*) = *True*
using ‹*T* ∈ *set Ts*› *both-present not-sat-conj-pos-false Ts-spec* **by** *blast*
then have ∃ α. $\alpha \models \psi \wedge \neg(\alpha \models \varphi)$
unfolding ψ-*def* φ-*def*
using *BigOr′-semantics* ‹*T* ∈ *set Ts*› ‹*i* ∈ {*1..n*}› *semantics-Fn-iff* **by** *metis*
then have ¬ *equiv* φ ψ
unfolding *equiv-def* **by** *auto*
then show *False*
using ‹*equiv* φ ψ› **by** *contradiction*

```
    qed
  qed

  have ex-T-cont-pos-var-eps: ∃ T ∈ set Ts. ∀ i ∈ {1..n}. cont-pos T (Var i (nth
eps (i−1)))
    if length eps = n for eps :: bool list
  proof (rule ccontr)
   assume assm: ¬ (∃ T ∈ set Ts. ∀ i ∈ {1..n}. cont-pos T (Var i (nth eps (i−1))))

    define α where
      α = (λx. case x of (Var i b) ⇒ b = nth eps (i−1))

    have α ⊨ φ
    proof −
      have α ⊨ Fn n
        using α-def by (induction n) simp-all
      then show ?thesis
        by (simp add: φ-def)
    qed

    moreover have ¬(α ⊨ ψ)
    proof −
      have ∀ T ∈ set Ts. ∃ i ∈ {1..n}. cont-pos T (Var i (¬(nth eps (i−1))))
        using assm occ-var-bool-diff by (metis (full-types))
      then have ∀ T ∈ set Ts. ¬(α ⊨ T)
        by (metis (mono-tags, lifting) Ts-spec α-def not-sat-conj-pos-false var.case)
      then show ?thesis
        unfolding ψ-def by auto
    qed

    ultimately have ¬ equiv φ ψ
      by (auto simp add: equiv-def)

    then show False
      using ‹equiv φ ψ› by contradiction
  qed

  have n-le-card-atoms: n ≤ card (atoms T) if T-in: T ∈ 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 :: bool list ⇒ var formula where
    conj-of-eps = (λeps.
      (SOME T. T ∈ set Ts ∧ (∀ i ∈ {1..(length eps)}. cont-pos T (Var i (nth eps
(i − 1))))))

  have T-of-conj-of-eps-in-Ts: conj-of-eps eps ∈ 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-ex')

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

also have ... ≤ card (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 ∈ {eps. length eps = n} and
      ys-in: ys ∈ {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 ⟹ xs = ys
    proof (erule contrapos-pp)
      assume xs ≠ ys
      then obtain i where i < n and nth xs i ≠ nth ys i
        using ‹length xs = length ys›
        using ‹length xs = n› nth-equalityI by blast

      have cont-pos (conj-of-eps xs) (Var (Suc i) (xs ! i))
      proof −
        have conj-of-eps xs ∈ set Ts ∧
              (∀ i∈{1..n}. cont-pos (conj-of-eps xs) (Var i (xs ! (i − 1))))
          by (smt (verit, ccfv-SIG) ‹length xs = n› conj-of-eps-def
              ex-T-cont-pos-var-eps someI-ex)
        then show ?thesis
          using ‹i < n› by force
      qed

      moreover have cont-pos (conj-of-eps ys) (Var (Suc i) (ys ! i))
      proof −
        have conj-of-eps ys ∈ set Ts ∧
              (∀ i∈{1..n}. cont-pos (conj-of-eps ys) (Var i (ys ! (i − 1))))
          by (smt (verit, ccfv-SIG) ‹length ys = n› conj-of-eps-def
              ex-T-cont-pos-var-eps someI-ex)
        then show ?thesis
          using ‹i < n› by force
      qed
```

```
      moreover have
        cont-pos (conj-of-eps ys) (Var (Suc i) False) ≠
        cont-pos (conj-of-eps ys) (Var (Suc i) True)
       using ‹i < n› occ-var-bool-diff[OF T-of-conj-of-eps-in-Ts[OF ‹length ys =
n›], of Suc i]
        by simp

      ultimately show conj-of-eps xs ≠ conj-of-eps ys
        using ‹xs ! i ≠ ys ! i›
        by (metis (mono-tags))
    qed
  qed
 next
  show conj-of-eps ‘ {eps. length eps = n} ⊆ set Ts
    using T-of-conj-of-eps-in-Ts by auto
 next
  show finite (set Ts)
    by simp
 qed

 also have ... ≤ length Ts
  using card-length[of Ts] .

 finally have length Ts ≥ 2^n .

 moreover have ∀ T ∈ set Ts. sizef T ≥ 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 φ :: 'a formula
 assumes dnf: is-dnf φ and sat: ∃ α. α ⊨ φ
 shows ∃ (Ts :: 'a formula list). equiv φ (BigOr' Ts) ∧
  size (BigOr' Ts) ≤ size φ ∧
  (∀ T ∈ set Ts. is-conj T) ∧
  (∀ T ∈ set Ts. ∃ α. α ⊨ T)
proof −
 have bex-sat: ∃ T ∈ set (undisj φ). ∃ α. α ⊨ T
  using sat
  by (metis equiv-def BigOr'-semantics equiv-BigOr'-undisj-if-dnf)

 define Ts :: 'a 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
    fix ψ :: var formula
    assume equiv (Fn n) ψ
    then have sat-G_n: ∃α. α ⊨ ψ
      unfolding equiv-def
      using Fn-sat[of n] by metis

    assume is-dnf ψ
    then obtain Ts :: var formula list where
      ∀ T ∈ set Ts. is-conj T and
      ∀ T ∈ set Ts. ∃α. α ⊨ T and
      equiv ψ (BigOr' Ts) and
      size: size (BigOr' Ts) ≤ size ψ
      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 ‹equiv (Fn n) ψ› ‹equiv ψ (BigOr' Ts)›] .

    ultimately have n * 2 ^ n ≤ sizef (BigOr' Ts)
      using exp-blowup-from-Fn-to-BigOr'[OF ‹0 < n›, of Ts] by metis

    then show n * 2 ^ n ≤ size ψ
      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 :: nat and Cs :: var formula list
  defines φ ≡ dual (Fn n) and ψ ≡ BigAnd' Cs
  assumes
    n-greater-0: n > 0 and
    Cs-spec: (∀ C ∈ set Cs. is-disj C ∧ ¬(∀ α. α ⊨ C)) and
    equiv φ ψ
  shows sizef ψ ≥ n*2^n
proof −
  have G-in-cnf: is-cnf ψ
    using ψ-def Cs-spec is-cnf-BigAnd' by auto
  have G-in-nnf: is-nnf ψ
    using G-in-cnf cnf-in-nnf by auto

  show ?thesis
  proof (rule ccontr)
    assume ¬ n*2^n ≤ sizef ψ
    then have sizef ψ < n*2^n
      by presburger
    then have sizef (dual ψ) < n*2^n
      using sizef-dual[of ψ] by presburger
```

```
    obtain Ts where
      dual ψ = BigOr' Ts ∀ T∈set Ts. is-conj T ∧ (∃ α. α ⊨ T)
      using ψ-def Cs-spec dual-conj-of-disjs-is-disj-of-conjs by metis

    moreover have equiv (Fn n) (dual ψ)
    proof −
      have equiv (dual ψ) (Not ψ)
        using equiv-dual .
      also have equiv ... (Not (dual (Fn n)))
        unfolding equiv-Not-left-Not-right
        using ‹equiv φ ψ›[symmetric, unfolded φ-def] .
      also have equiv ... (Not (Not (Fn n)))
        unfolding equiv-Not-left-Not-right
        using equiv-dual .
      also have equiv ... (Fn n)
        unfolding equiv-Not-Not-left
        using equiv-reflexive .
      finally show ?thesis
        by (rule equiv-symmetric)
    qed

    ultimately have n * 2 ^ n ≤ sizef (dual ψ)
      using exp-blowup-from-Fn-to-BigOr'[of n Ts] by force
    then show False
      using ‹sizef (dual ψ) < n*2^n› by presburger
  qed
qed

lemma ex-equiv-conj-list-if-is-cnf:
  fixes φ :: 'a formula
  assumes cnf: is-cnf φ
  shows ∃(Cs :: 'a formula list). equiv φ (BigAnd' Cs) ∧
    sizef (BigAnd' Cs) ≤ sizef φ ∧
    (∀ C ∈ set Cs. is-disj C) ∧
    (∀ C ∈ set Cs. ¬ ⊨ C)
proof −
  define Cs :: 'a formula list where
    Cs = filter (λC. ¬ ⊨ C) (unconj φ)

  show ?thesis
  proof (intro exI[of - Cs] conjI ballI)
    have equiv φ (BigAnd' (unconj φ))
      using equiv-BigAnd'-unconj-if-cnf[symmetric] .
    then show equiv φ (BigAnd' Cs)
      unfolding Cs-def
      by (auto simp add: equiv-def)
  next
    have sizef (BigAnd' (unconj φ)) = sizef φ
```

```
    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 ψ] by metis

  moreover have equiv (dual (Fn n)) (BigAnd' Cs)
    using equiv-transitive[OF ‹equiv (dual (Fn n)) ψ› ‹equiv ψ (BigAnd' Cs)›] .

  ultimately have n * 2 ^ n ≤ sizef (BigAnd' Cs)
    using exp-blowup-from-dual-Fn-to-BigAnd'[OF ‹0 < n›, of Cs] by metis

  then show n * 2 ^ n ≤ size ψ
    using sizef sizef-le-size[of ψ] 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.