

The Generalized Multiset Ordering is NP-Complete

René Thiemann Lukas Schmidinger

April 22, 2022

Abstract

We consider the problem of comparing two multisets via the generalized multiset ordering. We show that the corresponding decision problem is NP-complete. To be more precise, we encode multiset-comparisons into propositional formulas or into conjunctive normal forms of quadratic size; we further prove that satisfiability of conjunctive normal forms can be encoded as multiset-comparison problems of linear size.

As a corollary, we also show that the problem of deciding whether two terms are related by a recursive path order is NP-hard, provided the recursive path order is based on the generalized multiset ordering.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 2 |
| 2 | Properties of the Generalized Multiset Ordering | 2 |
| 3 | Propositional Formulas and CNFs | 9 |
| 3.1 | Propositional Formulas | 9 |
| 3.2 | Conjunctive Normal Forms | 10 |
| 4 | Deciding the Generalized Multiset Ordering is in NP | 11 |
| 4.1 | Locale for Generic Encoding | 11 |
| 4.2 | Definition of the Encoding | 12 |
| 4.3 | Soundness of the Encoding | 13 |
| 4.4 | Encoding into Propositional Formulas | 25 |
| 4.5 | Size of Propositional Formula Encoding is Quadratic | 26 |
| 4.6 | Encoding into Conjunctive Normal Form | 30 |
| 4.7 | Size of CNF-Encoding is Quadratic | 30 |
| 4.8 | Check Executability | 34 |

| | | |
|----------|--|-----------|
| 5 | Deciding the Generalized Multiset Ordering is NP-hard | 34 |
| 5.1 | Definition of the Encoding | 35 |
| 5.2 | Soundness of the Encoding | 35 |
| 5.3 | Size of Encoding is Linear | 40 |
| 5.4 | Check Executability | 40 |
| 6 | Deciding RPO-constraints is NP-hard | 41 |
| 6.1 | Definition of the Encoding | 41 |
| 6.2 | Soundness of the Encoding | 42 |
| 6.3 | Size of Encoding is Quadratic | 46 |
| 6.4 | Check Executability | 47 |

1 Introduction

Given a transitive and irreflexive relation \succ on elements, it can be extended to a relation on multisets (the *multiset ordering* \succ_{ms}) where for two multisets M and N the relation $M \succ_{ms} N$ is defined in a way that N is obtained from M by replacing some elements $a \in M$ by arbitrarily many elements b_1, \dots, b_n which are all smaller than a : $a \succ b_i$ for all $1 \leq i \leq n$.

Now, given \succ , M , and N , it is easy to decide $M \succ_{ms} N$: it is equivalent to demand $M \neq N$ and for each $b \in N \setminus M$ there must be some $a \in M \setminus N$ such that $a \succ b$.

The *generalized multiset ordering* is defined in terms of two relations \succ and \succsim . Here, one may additionally replace each element $a \in M$ by exactly one element b that satisfies $a \succsim b$. The multiset ordering is an instance of the generalized multiset ordering by choosing \succsim as the equality relation $=$.

The generalized multiset ordering is used in some definitions of the recursive path order (the original RPO [2] is defined via the multiset ordering, the variants of RPO [1, 4] use the generalized multiset ordering instead) so that more terms are in relation. A downside of the generalization is that the decision problem of whether two multisets are in relation becomes NP-complete, and also the decision problem for the RPO-variant in [4] is NP-complete.

In this AFP-entry we formalize NP-completeness of the generalized multiset ordering: we provide an $\mathcal{O}(n^2)$ encoding of multiset-comparisons into propositional formulas (using connectives $\vee, \wedge, \neg, \rightarrow, \leftrightarrow$), an $\mathcal{O}(n^2)$ encoding of multiset-comparisons into conjunctive normal forms (CNF), and an $\mathcal{O}(n)$ encoding of CNFs into multiset-comparisons. Moreover, we verify an $\mathcal{O}(n^2)$ encoding from a CNF into an RPO-constraint.

Our formalization is based on proofs in [1] (in NP) and [4] (NP-hardness).

2 Properties of the Generalized Multiset Ordering

```

theory Multiset-Ordering-More
  imports
    Weighted-Path-Order.Multiset-Extension2
begin

```

We provide characterizations of *s-mul-ext* and *ns-mul-ext* via introduction and elimination rules that are based on lists.

lemma *s-mul-ext-intro*:

```

  assumes  $xs = mset\ xs1 + mset\ xs2$ 
  and  $ys = mset\ ys1 + mset\ ys2$ 
  and  $length\ xs1 = length\ ys1$ 
  and  $\bigwedge i. i < length\ ys1 \implies (xs1\ !\ i, ys1\ !\ i) \in NS$ 
  and  $xs2 \neq []$ 
  and  $\bigwedge y. y \in set\ ys2 \implies \exists a \in set\ xs2. (a, y) \in S$ 
shows  $(xs, ys) \in s\text{-mul-ext}\ NS\ S$ 
  by (rule s-mul-extI[OF assms(1-2) multpw-listI[OF assms(3)]], insert assms(4-), auto)

```

lemma *ns-mul-ext-intro*:

```

  assumes  $xs = mset\ xs1 + mset\ xs2$ 
  and  $ys = mset\ ys1 + mset\ ys2$ 
  and  $length\ xs1 = length\ ys1$ 
  and  $\bigwedge i. i < length\ ys1 \implies (xs1\ !\ i, ys1\ !\ i) \in NS$ 
  and  $\bigwedge y. y \in set\ ys2 \implies \exists x \in set\ xs2. (x, y) \in S$ 
shows  $(xs, ys) \in ns\text{-mul-ext}\ NS\ S$ 
  by (rule ns-mul-extI[OF assms(1-2) multpw-listI[OF assms(3)]], insert assms(4-), auto)

```

lemma *ns-mul-ext-elim*: **assumes** $(xs, ys) \in ns\text{-mul-ext}\ NS\ S$

```

  shows  $\exists xs1\ xs2\ ys1\ ys2.$ 
     $xs = mset\ xs1 + mset\ xs2$ 
     $\wedge ys = mset\ ys1 + mset\ ys2$ 
     $\wedge length\ xs1 = length\ ys1$ 
     $\wedge (\forall i. i < length\ ys1 \implies (xs1\ !\ i, ys1\ !\ i) \in NS)$ 
     $\wedge (\forall y \in set\ ys2. \exists x \in set\ xs2. (x, y) \in S)$ 

```

proof –

from *ns-mul-extE*[*OF assms*] **obtain**

A1 A2 B1 B2 **where** $*$: $xs = A1 + A2\ ys = B1 + B2$

and *NS*: $(A1, B1) \in multpw\ NS$

and *S*: $\bigwedge b. b \in \# B2 \implies \exists a. a \in \# A2 \wedge (a, b) \in S$

by *blast*

from *multpw-listE*[*OF NS*] **obtain** *xs1 ys1* **where** $**$: $length\ xs1 = length\ ys1$

A1 = mset xs1 B1 = mset ys1

and *NS*: $\bigwedge i. i < length\ ys1 \implies (xs1\ !\ i, ys1\ !\ i) \in NS$ **by** *auto*

from *surj-mset* **obtain** *xs2* **where** *A2*: $A2 = mset\ xs2$ **by** *auto*

from *surj-mset* **obtain** *ys2* **where** *B2*: $B2 = mset\ ys2$ **by** *auto*

show *?thesis*

proof (rule $exI[of - xs1]$, rule $exI[of - xs2]$, rule $exI[of - ys1]$, rule $exI[of - ys2]$,
intro conjI)
show $xs = mset\ xs1 + mset\ xs2$ **using** $**\ A2\ B2$ **by** *auto*
show $ys = mset\ ys1 + mset\ ys2$ **using** $**\ A2\ B2$ **by** *auto*
show $length\ xs1 = length\ ys1$ **by** *fact*
show $\forall i < length\ ys1. (xs1\ !\ i, ys1\ !\ i) \in NS$ **using** $**\ A2\ B2\ NS$ **by** *auto*
show $\forall y \in set\ ys2. \exists x \in set\ xs2. (x, y) \in S$ **using** $**\ A2\ B2\ S$ **by** *auto*
qed
qed

lemma *s-mul-ext-elim*: **assumes** $(xs, ys) \in s\text{-mul-ext}\ NS\ S$

shows $\exists\ xs1\ xs2\ ys1\ ys2.$

$xs = mset\ xs1 + mset\ xs2$

$\wedge\ ys = mset\ ys1 + mset\ ys2$

$\wedge\ length\ xs1 = length\ ys1$

$\wedge\ xs2 \neq []$

$\wedge\ (\forall\ i. i < length\ ys1 \longrightarrow (xs1\ !\ i, ys1\ !\ i) \in NS)$

$\wedge\ (\forall\ y \in set\ ys2. \exists\ x \in set\ xs2. (x, y) \in S)$

proof –

from *s-mul-extE[OF assms]* **obtain**

$A1\ A2\ B1\ B2$ **where** $*$: $xs = A1 + A2\ ys = B1 + B2$

and NS : $(A1, B1) \in multpw\ NS$ **and** *nonempty*: $A2 \neq \{\#\}$

and S : $\wedge b. b \in \# B2 \implies \exists a. a \in \# A2 \wedge (a, b) \in S$

by *blast*

from *multpw-listE[OF NS]* **obtain** $xs1\ ys1$ **where** $**$: $length\ xs1 = length\ ys1$

$A1 = mset\ xs1\ B1 = mset\ ys1$

and NS : $\wedge\ i. i < length\ ys1 \implies (xs1\ !\ i, ys1\ !\ i) \in NS$ **by** *auto*

from *surj-mset* **obtain** $xs2$ **where** $A2$: $A2 = mset\ xs2$ **by** *auto*

from *surj-mset* **obtain** $ys2$ **where** $B2$: $B2 = mset\ ys2$ **by** *auto*

show *?thesis*

proof (rule $exI[of - xs1]$, rule $exI[of - xs2]$, rule $exI[of - ys1]$, rule $exI[of - ys2]$,
intro conjI)

show $xs = mset\ xs1 + mset\ xs2$ **using** $**\ A2\ B2$ **by** *auto*

show $ys = mset\ ys1 + mset\ ys2$ **using** $**\ A2\ B2$ **by** *auto*

show $length\ xs1 = length\ ys1$ **by** *fact*

show $\forall i < length\ ys1. (xs1\ !\ i, ys1\ !\ i) \in NS$ **using** $**\ A2\ B2\ NS$ **by** *auto*

show $\forall y \in set\ ys2. \exists x \in set\ xs2. (x, y) \in S$ **using** $**\ A2\ B2\ S$ **by** *auto*

show $xs2 \neq []$ **using** *nonempty A2* **by** *auto*

qed

qed

We further add a lemma that shows, that it does not matter whether one adds the strict relation to the non-strict relation or not.

lemma *ns-mul-ext-some-S-in-NS*: **assumes** $S' \subseteq S$

shows $ns\text{-mul-ext}\ (NS \cup S')\ S = ns\text{-mul-ext}\ NS\ S$

proof

show $ns\text{-mul-ext}\ NS\ S \subseteq ns\text{-mul-ext}\ (NS \cup S')\ S$

by (*simp add: ns-mul-ext-mono*)

show $ns\text{-mul-ext}\ (NS \cup S')\ S \subseteq ns\text{-mul-ext}\ NS\ S$

```

proof
  fix as bs
  assume  $(as, bs) \in ns\text{-mul-ext } (NS \cup S') S$ 
  from ns-mul-extE[OF this] obtain nas sas nbs sbs where
    as: as = nas + sas and bs: bs = nbs + sbs
    and ns: (nas, nbs) \in multpw (NS \cup S')
    and s: (\bigwedge b. b \in\# sbs \implies \exists a. a \in\# sas \wedge (a, b) \in S) by blast
  from ns have  $\exists nas2\ sas2\ nbs2\ sbs2. nas = nas2 + sas2 \wedge nbs = nbs2 + sbs2$ 
 $\wedge (nas2, nbs2) \in multpw NS$ 
   $\wedge (\forall b \in\# sbs2. (\exists a. a \in\# sas2 \wedge (a, b) \in S))$ 
  proof (induct)
    case (add a b nas nbs)
    from add(3) obtain nas2 sas2 nbs2 sbs2 where  $*$ :  $nas = nas2 + sas2 \wedge nbs$ 
 $= nbs2 + sbs2 \wedge (nas2, nbs2) \in multpw NS$ 
   $\wedge (\forall b \in\# sbs2. (\exists a. a \in\# sas2 \wedge (a, b) \in S))$  by blast
    from add(1)
    show ?case
  proof
    assume  $(a, b) \in S'$ 
    with assms have ab: (a, b) \in S by auto
    have one: add-mset a nas = nas2 + (add-mset a sas2) using  $*$  by auto
    have two: add-mset b nbs = nbs2 + (add-mset b sbs2) using  $*$  by auto
    show ?thesis
    by (intro exI conjI, rule one, rule two, insert ab *, auto)
  next
    assume ab: (a, b) \in NS
    have one: add-mset a nas = (add-mset a nas2) + sas2 using  $*$  by auto
    have two: add-mset b nbs = (add-mset b nbs2) + sbs2 using  $*$  by auto
    show ?thesis
    by (intro exI conjI, rule one, rule two, insert ab *, auto intro: multpw.add)

  qed
qed auto
then obtain nas2 sas2 nbs2 sbs2 where  $*$ :  $nas = nas2 + sas2 \wedge nbs = nbs2$ 
 $+ sbs2 \wedge (nas2, nbs2) \in multpw NS$ 
   $\wedge (\forall b \in\# sbs2. (\exists a. a \in\# sas2 \wedge (a, b) \in S))$  by auto
  have as: as = nas2 + (sas2 + sas) and bs: bs = nbs2 + (sbs2 + sbs)
  unfolding as bs using  $*$  by auto
  show  $(as, bs) \in ns\text{-mul-ext } NS S$ 
  by (intro ns-mul-extI[OF as bs], insert * s, auto)
qed
qed

```

lemma *ns-mul-ext-NS-union-S*: $ns\text{-mul-ext } (NS \cup S) S = ns\text{-mul-ext } NS S$
by (*rule ns-mul-ext-some-S-in-NS, auto*)

Some further lemmas on multisets

lemma *mset-map-filter*: $mset (map\ v (filter (\lambda e. c\ e) t)) + mset (map\ v (filter$

$(\lambda e. \neg(c\ e))\ t)) = \text{mset } (\text{map } v\ t)$
by $(\text{induct } t, \text{auto})$

lemma *mset-map-split*: **assumes** $\text{mset } (\text{map } f\ xs) = \text{mset } ys1 + \text{mset } ys2$
shows $\exists\ zs1\ zs2. \text{mset } xs = \text{mset } zs1 + \text{mset } zs2 \wedge ys1 = \text{map } f\ zs1 \wedge ys2 = \text{map } f\ zs2$
using *assms*
proof $(\text{induct } xs\ \text{arbitrary: } ys1\ ys2)$
case $(\text{Cons } x\ xs\ ys1\ ys2)$
have $f\ x \in \# \text{mset } (\text{map } f\ (x \# xs))$ **by** *simp*
from $\text{this}[\text{unfolded } \text{Cons}(2)]$
have $f\ x \in \text{set } ys1 \cup \text{set } ys2$ **by** *auto*
thus $?case$
proof
let $?ys1 = ys1$ **let** $?ys2 = ys2$
assume $f\ x \in \text{set } ?ys1$
from $\text{split-list}[\text{OF } \text{this}]$ **obtain** $us1\ us2$ **where** $ys1: ?ys1 = us1 @ f\ x \# us2$
by *auto*
let $?us = us1 @ us2$
from $\text{Cons}(2)[\text{unfolded } ys1]$ **have** $\text{mset } (\text{map } f\ xs) = \text{mset } ?us + \text{mset } ?ys2$ **by** *auto*
from $\text{Cons}(1)[\text{OF } \text{this}]$ **obtain** $zs1\ zs2$ **where** $xs: \text{mset } xs = \text{mset } zs1 + \text{mset } zs2$
and $us: ?us = \text{map } f\ zs1$ **and** $ys: ?ys2 = \text{map } f\ zs2$
by *auto*
let $?zs1 = \text{take } (\text{length } us1)\ zs1$ **let** $?zs2 = \text{drop } (\text{length } us1)\ zs1$
show $?thesis$
apply $(\text{rule } \text{exI}[\text{of } -\ ?zs1 @ x \# ?zs2], \text{rule } \text{exI}[\text{of } -\ zs2])$
apply $(\text{unfold } ys1, \text{unfold } ys, \text{intro } \text{conjI } \text{refl})$
proof $-$
have $\text{mset } (x \# xs) = \{\# x \#\} + \text{mset } xs$ **by** *simp*
also **have** $\dots = \text{mset } (x \# zs1) + \text{mset } zs2$ **using** xs **by** *simp*
also **have** $zs1 = ?zs1 @ ?zs2$ **by** *simp*
also **have** $\text{mset } (x \# \dots) = \text{mset } (?zs1 @ x \# ?zs2)$ **by** $(\text{simp } \text{add: union-code})$
finally **show** $\text{mset } (x \# xs) = \text{mset } (?zs1 @ x \# ?zs2) + \text{mset } zs2$.
show $us1 @ f\ x \# us2 = \text{map } f\ (?zs1 @ x \# ?zs2)$ **using** us
by $(\text{smt } (\text{verit}, \text{best}) \langle zs1 = \text{take } (\text{length } us1)\ zs1 @ \text{drop } (\text{length } us1)\ zs1 \rangle$
add-diff-cancel-left' append-eq-append-conv length-append length-drop length-map list.simps(9) map-eq-append-conv)
qed
next
let $?ys1 = ys2$ **let** $?ys2 = ys1$
assume $f\ x \in \text{set } ?ys1$
from $\text{split-list}[\text{OF } \text{this}]$ **obtain** $us1\ us2$ **where** $ys1: ?ys1 = us1 @ f\ x \# us2$
by *auto*
let $?us = us1 @ us2$
from $\text{Cons}(2)[\text{unfolded } ys1]$ **have** $\text{mset } (\text{map } f\ xs) = \text{mset } ?us + \text{mset } ?ys2$ **by** *auto*
from $\text{Cons}(1)[\text{OF } \text{this}]$ **obtain** $zs1\ zs2$ **where** $xs: \text{mset } xs = \text{mset } zs1 + \text{mset } zs2$

zs2

```
and us: ?us = map f zs1 and ys: ?ys2 = map f zs2
by auto
let ?zs1 = take (length us1) zs1 let ?zs2 = drop (length us1) zs1
show ?thesis
  apply (rule exI[of - zs2], rule exI[of - ?zs1 @ x # ?zs2])
  apply (unfold ys1, unfold ys, intro conjI refl)
proof -
  have mset (x # xs) = {# x #} + mset xs by simp
  also have ... = mset zs2 + mset (x # zs1) using xs by simp
  also have zs1 = ?zs1 @ ?zs2 by simp
  also have mset (x # ...) = mset (?zs1 @ x # ?zs2) by (simp add: union-code)
  finally show mset (x # xs) = mset zs2 + mset (?zs1 @ x # ?zs2) .
  show us1 @ f x # us2 = map f (?zs1 @ x # ?zs2) using us
    by (smt (verit, best) ⟨zs1 = take (length us1) zs1 @ drop (length us1) zs1⟩
      add-diff-cancel-left' append-eq-append-conv length-append length-drop length-map
      list.simps(9) map-eq-append-conv)
qed
qed
qed auto
```

lemma *deciding-mult*:

```
assumes tr: trans S and ir: irrefl S
shows (N,M) ∈ mult S = (M ≠ N ∧ (∀ b ∈# N - M. ∃ a ∈# M - N. (b,a)
∈ S))
proof -
  define I where I = M ∩# N
  have N: N = (N - M) + I unfolding I-def
    by (metis add commute diff-intersect-left-idem multiset-inter-commute subset-mset.add-diff-inverse subset-mset.inf-le1)
  have M: M = (M - N) + I unfolding I-def
    by (metis add commute diff-intersect-left-idem subset-mset.add-diff-inverse subset-mset.inf-le1)
  have (N,M) ∈ mult S ↔
    ((N - M) + I, (M - N) + I) ∈ mult S
    using N M by auto
  also have ... ↔ (N - M, M - N) ∈ mult S
    by (rule mult-cancel[OF tr ir])
  also have ... ↔ (M ≠ N ∧ (∀ b ∈# N - M. ∃ a ∈# M - N. (b,a) ∈ S))
  proof
    assume *: (M ≠ N ∧ (∀ b ∈# N - M. ∃ a ∈# M - N. (b,a) ∈ S))
    have ({#} + (N - M), {#} + (M - N)) ∈ mult S
      apply (rule one-step-implies-mult, insert *, auto)
      using M N by auto
    thus (N - M, M - N) ∈ mult S by auto
  next
    assume (N - M, M - N) ∈ mult S
    from mult-implies-one-step[OF tr this]
    obtain E J K
```

where *: $M - N = E + J \wedge$
 $N - M = E + K$ **and** $rel: J \neq \{\#\} \wedge (\forall k \in \#K. \exists j \in \#J. (k, j) \in S)$ **by**
auto
from * **have** $E = \{\#\}$
by (*metis* (*full-types*) *M N add-diff-cancel-right add-implies-diff cancel-ab-semigroup-add-class.diff-right-com*
diff-add-zero)
with * **have** $JK: J = M - N K = N - M$ **by** *auto*
show $(M \neq N \wedge (\forall b \in \# N - M. \exists a \in \# M - N. (b, a) \in S))$
using *rel* **unfolding** *JK* **by** *auto*
qed
finally show *?thesis* .
qed

lemma *s-mul-ext-map*: $(\bigwedge a b. a \in \text{set } as \implies b \in \text{set } bs \implies (a, b) \in S \implies (f a,$
 $f b) \in S') \implies$
 $(\bigwedge a b. a \in \text{set } as \implies b \in \text{set } bs \implies (a, b) \in NS \implies (f a, f b) \in NS') \implies$
 $(as, bs) \in \{(as, bs). (mset as, mset bs) \in s\text{-mul-ext } NS S\} \implies$
 $(map f as, map f bs) \in \{(as, bs). (mset as, mset bs) \in s\text{-mul-ext } NS' S'\}$
using *mult2-alt-map*[*of - - NS⁻¹ ff (NS')⁻¹ S⁻¹ S'⁻¹ False*] **unfolding** *s-mul-ext-def*
by *fastforce*

lemma *fst-mul-ext-imp-fst*: **assumes** *fst* (*mul-ext* *f xs ys*)
and $\text{length } xs \leq \text{length } ys$
shows $\exists x y. x \in \text{set } xs \wedge y \in \text{set } ys \wedge \text{fst } (f x y)$
proof -
from *assms(1)*[*unfolded mul-ext-def Let-def fst-conv*]
have $(mset xs, mset ys) \in s\text{-mul-ext } \{(x, y). \text{snd } (f x y)\} \{(x, y). \text{fst } (f x y)\}$ **by**
auto
from *s-mul-ext-elim*[*OF this*] **obtain** *xs1 xs2 ys1 ys2*
where *: $mset xs = mset xs1 + mset xs2$
 $mset ys = mset ys1 + mset ys2$
 $\text{length } xs1 = \text{length } ys1$
 $xs2 \neq []$
 $(\forall y \in \text{set } ys2. \exists x \in \text{set } xs2. (x, y) \in \{(x, y). \text{fst } (f x y)\})$ **by** *auto*
from *(1-3) *assms(2)* **have** $\text{length } xs2 \leq \text{length } ys2$
by (*metis* *add-le-cancel-left size-mset size-union*)
with *(4) **have** $\text{hd } ys2 \in \text{set } ys2$ **by** (*cases* *ys2*, *auto*)
with *(5,1,2) **show** *?thesis*
by (*metis* *Un-iff mem-Collect-eq prod.simps(2) set-mset-mset set-mset-union*)
qed

lemma *ns-mul-ext-point*: **assumes** $(as, bs) \in ns\text{-mul-ext } NS S$
and $b \in \# bs$
shows $\exists a \in \# as. (a, b) \in NS \cup S$
proof -
from *ns-mul-ext-elim*[*OF assms(1)*]
obtain *xs1 xs2 ys1 ys2*
where *: $as = mset xs1 + mset xs2$
 $bs = mset ys1 + mset ys2$


```

    length xs1 = length ys1
    ( $\forall i < \text{length } ys1. (xs1 ! i, ys1 ! i) \in NS$ ) ( $\forall y \in \text{set } ys2. \exists x \in \text{set } xs2. (x, y) \in S$ )
  by auto
  from assms(2)[unfolded *] have  $b \in \text{set } ys1 \vee b \in \text{set } ys2$  by auto
  thus ?thesis
  proof
    assume  $b \in \text{set } ys2$ 
    with * obtain a where  $a \in \text{set } xs2$  and  $(a, b) \in S$  by auto
    with *(1) show ?thesis by auto
  next
    assume  $b \in \text{set } ys1$ 
    from this[unfolded set-conv-nth] obtain i where  $i < \text{length } ys1$  and  $b = ys1$ 
    ! i by auto
    with *(4) have  $(xs1 ! i, b) \in NS$  by auto
    moreover from i *(3) have  $xs1 ! i \in \text{set } xs1$  by auto
    ultimately show ?thesis using *(1) by auto
  qed
qed

```

```

lemma s-mul-ext-point: assumes  $(as, bs) \in s\text{-mul-ext } NS S$ 
  and  $b \in \# bs$ 
shows  $\exists a \in \# as. (a, b) \in NS \cup S$ 
  by (rule ns-mul-ext-point, insert assms s-ns-mul-ext, auto)

```

end

3 Propositional Formulas and CNFs

We provide a straight-forward definition of propositional formulas, defined as arbitrary formulas using variables, negations, conjunctions and disjunctions. CNFs are represented as lists of lists of literals and then converted into formulas.

```

theory Propositional-Formula
  imports Main
begin

```

3.1 Propositional Formulas

```

datatype 'a formula =
  Prop 'a |
  Conj 'a formula list |
  Disj 'a formula list |
  Neg 'a formula |
  Impl 'a formula 'a formula |
  Equiv 'a formula 'a formula

```

```

fun eval :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a formula  $\Rightarrow$  bool where

```

```

    eval v (Prop x) = v x
| eval v (Neg f) = (¬ eval v f)
| eval v (Conj fs) = (∀ f ∈ set fs. eval v f)
| eval v (Disj fs) = (∃ f ∈ set fs. eval v f)
| eval v (Impl f g) = (eval v f → eval v g)
| eval v (Equiv f g) = (eval v f ↔ eval v g)

```

Definition of propositional formula size: number of connectives

```

fun size-pf :: 'a formula ⇒ nat where
  size-pf (Prop x) = 1
| size-pf (Neg f) = 1 + size-pf f
| size-pf (Conj fs) = 1 + sum-list (map size-pf fs)
| size-pf (Disj fs) = 1 + sum-list (map size-pf fs)
| size-pf (Impl f g) = 1 + size-pf f + size-pf g
| size-pf (Equiv f g) = 1 + size-pf f + size-pf g

```

3.2 Conjunctive Normal Forms

```

type-synonym 'a clause = ('a × bool) list
type-synonym 'a cnf = 'a clause list

```

```

fun formula-of-lit :: 'a × bool ⇒ 'a formula where
  formula-of-lit (x, True) = Prop x
| formula-of-lit (x, False) = Neg (Prop x)

```

```

definition formula-of-cnf :: 'a cnf ⇒ 'a formula where
  formula-of-cnf = (Conj o map (Disj o map formula-of-lit))

```

```

definition eval-cnf :: ('a ⇒ bool) ⇒ 'a cnf ⇒ bool where
  eval-cnf α cnf = eval α (formula-of-cnf cnf)

```

```

lemma eval-cnf-alt-def: eval-cnf α cnf = Ball (set cnf) (λ c. Bex (set c) (λ l. α
(fst l) = snd l))

```

```

unfolding eval-cnf-def formula-of-cnf-def o-def eval.simps set-map Ball-image-comp
bex-simps
apply (intro ball-cong bex-cong refl)
subgoal for c l by (cases l; cases snd l, auto)
done

```

The size of a CNF is the number of literals + the number of clauses, i.e., the sum of the lengths of all clauses + the length.

```

definition size-cnf :: 'a cnf ⇒ nat where
  size-cnf cnf = sum-list (map length cnf) + length cnf

```

end

4 Deciding the Generalized Multiset Ordering is in NP

We first define a SAT-encoding for the comparison of two multisets w.r.t. two relations S and NS, then show soundness of the encoding and finally show that the size of the encoding is quadratic in the input.

```

theory
  Multiset-Ordering-in-NP
imports
  Multiset-Ordering-More
  Propositional-Formula
begin

```

4.1 Locale for Generic Encoding

We first define a generic encoding which may be instantiated for both propositional formulas and for CNFs. Here, we require some encoding primitives with the semantics specified in the enc-sound assumptions.

```

locale encoder =
  fixes eval :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'f  $\Rightarrow$  bool
  and enc-False :: 'f
  and enc-True :: 'f
  and enc-pos :: 'a  $\Rightarrow$  'f
  and enc-neg :: 'a  $\Rightarrow$  'f
  and enc-different :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'f
  and enc-equiv-and-not :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  'f
  and enc-equiv-ite :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  'f
  and enc-ite :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  'f
  and enc-impl :: 'a  $\Rightarrow$  'f  $\Rightarrow$  'f
  and enc-var-impl :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'f
  and enc-not-and :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'f
  and enc-not-all :: 'a list  $\Rightarrow$  'f
  and enc-conj :: 'f list  $\Rightarrow$  'f
assumes enc-sound[simp]:
  eval  $\alpha$  (enc-False) = False
  eval  $\alpha$  (enc-True) = True
  eval  $\alpha$  (enc-pos x) =  $\alpha$  x
  eval  $\alpha$  (enc-neg x) = ( $\neg$   $\alpha$  x)
  eval  $\alpha$  (enc-different x y) = ( $\alpha$  x  $\neq$   $\alpha$  y)
  eval  $\alpha$  (enc-equiv-and-not x y z) = ( $\alpha$  x  $\longleftrightarrow$   $\alpha$  y  $\wedge$   $\neg$   $\alpha$  z)
  eval  $\alpha$  (enc-equiv-ite x y z u) = ( $\alpha$  x  $\longleftrightarrow$  (if  $\alpha$  y then  $\alpha$  z else  $\alpha$  u))
  eval  $\alpha$  (enc-ite x y z) = (if  $\alpha$  x then  $\alpha$  y else  $\alpha$  z)
  eval  $\alpha$  (enc-impl x f) = ( $\alpha$  x  $\longrightarrow$  eval  $\alpha$  f)
  eval  $\alpha$  (enc-var-impl x y) = ( $\alpha$  x  $\longrightarrow$   $\alpha$  y)
  eval  $\alpha$  (enc-not-and x y) = ( $\neg$  ( $\alpha$  x  $\wedge$   $\alpha$  y))
  eval  $\alpha$  (enc-not-all xs) = ( $\neg$  (Ball (set xs)  $\alpha$ ))
  eval  $\alpha$  (enc-conj fs) = (Ball (set fs) (eval  $\alpha$ ))
begin

```

4.2 Definition of the Encoding

We need to encode formulas of the shape that exactly one variable is evaluated to true. Here, we use the linear encoding of [3, Section 5.3] that requires some auxiliary variables. More precisely, for each propositional variable that we want to count we require two auxiliary variables.

```
fun encode-sum-0-1-main :: ('a × 'a × 'a) list ⇒ 'f list × 'a × 'a where
  encode-sum-0-1-main [(x, zero, one)] = ([enc-different zero x], zero, x)
| encode-sum-0-1-main ((x, zero, one) # rest) = (case encode-sum-0-1-main rest
of
  (conds, fzero, fone) ⇒ let
    czero = enc-equiv-and-not zero fzero x;
    cone = enc-equiv-ite one x fzero fone
  in (czero # cone # conds, zero, one))
```

```
definition encode-exactly-one :: ('a × 'a × 'a) list ⇒ 'f × 'f list where
  encode-exactly-one vars = (case vars of [] ⇒ (enc-False, [])
| [(x,-,-)] ⇒ (enc-pos x, [])
| ((x,-,-) # vars) ⇒ (case encode-sum-0-1-main vars of (conds, zero, one)
⇒ (enc-ite x zero one, conds)))
```

```
fun encodeGammaCond :: 'a ⇒ 'a ⇒ bool ⇒ bool ⇒ 'f where
  encodeGammaCond gam eps True True = enc-True
| encodeGammaCond gam eps False False = enc-neg gam
| encodeGammaCond gam eps False True = enc-var-impl gam eps
| encodeGammaCond gam eps True False = enc-not-and gam eps
end
```

The encoding of the multiset comparisons is based on [1, Sections 3.6 and 3.7]. It uses propositional variables γ_{ij} and ϵ_i . We further add auxiliary variables that are required for the exactly-one-encoding.

```
datatype PropVar = Gamma nat nat | Epsilon nat
| AuxZeroJI nat nat | AuxOneJI nat nat
| AuxZeroIJ nat nat | AuxOneIJ nat nat
```

At this point we define a new locale as an instance of *encoder* where the type of propositional variables is fixed to *PropVar*.

```
locale ms-encoder = encoder eval for eval :: (PropVar ⇒ bool) ⇒ 'f ⇒ bool
begin
```

```
definition formula14 :: nat ⇒ nat ⇒ 'f list where
formula14 n m = (let
  inner-left = λ j. case encode-exactly-one (map (λ i. (Gamma i j, AuxZeroJI i
j, AuxOneJI i j)) [0 ..< n])
of (one, conds) ⇒ one # conds;
  left = List.maps inner-left [0 ..< m];
  inner-right = λ i. encode-exactly-one (map (λ j. (Gamma i j, AuxZeroIJ i j,
AuxOneIJ i j)) [0 ..< m]);
```

$right = List.maps (\lambda i. case inner-right\ i\ of\ (one, cands) \Rightarrow enc-impl\ (Epsilon\ i)\ one\ \# cands)\ [0 \dots n]$
 $in\ left\ @\ right)$

definition $formula15 :: (nat \Rightarrow nat \Rightarrow bool) \Rightarrow (nat \Rightarrow nat \Rightarrow bool) \Rightarrow nat \Rightarrow nat \Rightarrow 'f\ list\ \mathbf{where}$
 $formula15\ cs\ cns\ n\ m = (let$
 $conjs = List.maps (\lambda i. List.maps (\lambda j. let\ s = cs\ i\ j; ns = cns\ i\ j\ in$
 $if\ s \wedge ns\ then\ []\ else\ [encodeGammaCond\ (Gamma\ i\ j)\ (Epsilon\ i)\ s\ ns])\ [0 \dots m])\ [0 \dots n]$
 $in\ conjs\ @\ formula14\ n\ m)$

definition $formula16 :: (nat \Rightarrow nat \Rightarrow bool) \Rightarrow (nat \Rightarrow nat \Rightarrow bool) \Rightarrow nat \Rightarrow nat \Rightarrow 'f\ list\ \mathbf{where}$
 $formula16\ cs\ cns\ n\ m = (enc-not-all\ (map\ Epsilon\ [0 \dots n])\ \# formula15\ cs\ cns\ n\ m)$

The main encoding function. It takes a function as input that returns for each pair of elements a pair of Booleans, and these indicate whether the elements are strictly or weakly decreasing. Moreover, two input lists are given. Finally two formulas are returned, where the first is satisfiable iff the two lists are strictly decreasing w.r.t. the multiset ordering, and second is satisfiable iff there is a weak decrease w.r.t. the multiset ordering.

definition $encode-mul-ext :: ('a \Rightarrow 'a \Rightarrow bool \times bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow 'f \times 'f\ \mathbf{where}$
 $encode-mul-ext\ s-ns\ xs\ ys = (let$
 $n = length\ xs;$
 $m = length\ ys;$
 $cs = (\lambda\ i\ j. fst\ (s-ns\ (xs\ !\ i)\ (ys\ !\ j)));$
 $cns = (\lambda\ i\ j. snd\ (s-ns\ (xs\ !\ i)\ (ys\ !\ j)));$
 $f15 = formula15\ cs\ cns\ n\ m;$
 $f16 = enc-not-all\ (map\ Epsilon\ [0 \dots n])\ \# f15$
 $in\ (enc-conj\ f16,\ enc-conj\ f15))$
end

4.3 Soundness of the Encoding

context $encoder$
begin

abbreviation $eval-all :: ('a \Rightarrow bool) \Rightarrow 'f\ list \Rightarrow bool\ \mathbf{where}$
 $eval-all\ \alpha\ fs \equiv (Ball\ (set\ fs)\ (eval\ \alpha))$

lemma $encode-sum-0-1-main$: **assumes** $encode-sum-0-1-main\ vars = (conds, zero, one)$

and $\bigwedge i\ x\ ze\ on\ re. prop \Longrightarrow i < length\ vars \Longrightarrow drop\ i\ vars = ((x, ze, on) \# re)$
 \Longrightarrow
 $(\alpha\ ze \longleftrightarrow \neg (\exists\ y \in insert\ x\ (fst\ 'set\ re). \alpha\ y))$
 $\wedge (\alpha\ on \longleftrightarrow (\exists!\ y \in insert\ x\ (fst\ 'set\ re). \alpha\ y))$

```

and  $\neg prop \implies eval\text{-}all \ \alpha \ cond$ s
and distinct (map fst vars)
and vars  $\neq []$ 
shows eval-all  $\alpha \ cond$ s
   $\wedge (\alpha \ zero \longleftrightarrow \neg (\exists x \in fst \ ' \ set \ vars. \ \alpha \ x))$ 
   $\wedge (\alpha \ one \longleftrightarrow (\exists! x \in fst \ ' \ set \ vars. \ \alpha \ x))$ 
  using assms
proof (induct vars arbitrary: conds zero one rule: encode-sum-0-1-main.induct)
  case (1 x zero' one' conds zero one)
  from 1(1,3-) 1(2)[of 0] show ?case by (cases prop, auto)
next
  case Cons: (2 x zero one r rr conds' zero' one')
  let ?triple = (x,zero,one)
  let ?rest = r # rr
  obtain conds fzero fone where res: encode-sum-0-1-main ?rest = (conds, fzero,
fone)
  by (cases encode-sum-0-1-main ?rest, auto)
  from Cons(2)[unfolded encode-sum-0-1-main.simps res split Let-def]
  have zero: zero' = zero and one: one' = one and
    conds': conds' = enc-equiv-and-not zero fzero x # enc-equiv-ite one x fzero
fone # conds
  by auto
  from Cons(5) have x: x  $\notin$  fst ' set ?rest
    and dist: distinct (map fst ?rest) by auto
  have eval-all  $\alpha \ conds \wedge \alpha \ fzero = (\neg (\exists a \in fst \ ' \ set \ ?rest. \ \alpha \ a)) \wedge \alpha \ fone = (\exists!x. x \in fst \ ' \ set \ ?rest \wedge \alpha \ x)$ 
  apply (rule Cons(1)[OF res - - dist])
  subgoal for i x ze on re using Cons(3)[of Suc i x ze on re] by auto
  subgoal using Cons(4) unfolding conds' by auto
  subgoal by auto
  done
  hence IH: eval-all  $\alpha \ conds \wedge \alpha \ fzero = (\neg (\exists a \in fst \ ' \ set \ ?rest. \ \alpha \ a))$ 
 $\wedge \alpha \ fone = (\exists!x. x \in fst \ ' \ set \ ?rest \wedge \alpha \ x)$  by auto
  show ?case
  proof (cases prop)
  case True
  from Cons(3)[of 0 x zero one ?rest, OF True]
  have id:  $\alpha \ zero = (\forall y \in insert \ x \ (fst \ ' \ set \ ?rest). \ \neg \ \alpha \ y)$ 
 $\alpha \ one = (\exists!y. y \in insert \ x \ (fst \ ' \ set \ ?rest) \wedge \alpha \ y)$  by auto
  show ?thesis unfolding zero one conds' eval.simps using x IH(1)
  apply (simp add: IH id)
  by blast
next
  case False
  from Cons(4)[OF False, unfolded conds']
  have id:  $\alpha \ zero = (\neg \ \alpha \ x \wedge \alpha \ fzero)$ 
 $\alpha \ one = (\alpha \ x \wedge \alpha \ fzero \vee \neg \ \alpha \ x \wedge \alpha \ fone)$  by auto
  show ?thesis unfolding zero one conds' eval.simps using x IH(1)
  apply (simp add: IH id)

```

```

    by blast
  qed
qed auto

lemma encode-exactly-one-complete: assumes encode-exactly-one vars = (one,
conds)
and  $\bigwedge i x ze on. i < length\ vars \implies$ 
  vars ! i = (x,ze,on)  $\implies$ 
  ( $\alpha\ ze \longleftrightarrow \neg (\exists y \in fst\ 'set\ (drop\ i\ vars). \alpha\ y)$ )
   $\wedge (\alpha\ on \longleftrightarrow (\exists! y \in fst\ 'set\ (drop\ i\ vars). \alpha\ y))$ 
and distinct (map fst vars)
shows eval-all  $\alpha\ conds \wedge (eval\ \alpha\ one \longleftrightarrow (\exists! x \in fst\ 'set\ vars. \alpha\ x))$ 
proof -
  consider (empty) vars = [] | (single) x ze on where vars = [(x,ze,on)]
  | (other) x ze on v vs where vars = (x,ze,on) # v # vs
  by (cases vars; cases tl vars; auto)
  thus ?thesis
proof cases
  case (other x ze' on' v vs)
  obtain on zero where res: encode-sum-0-1-main (v # vs) = (conds, zero, on)

    and one: one = enc-ite x zero on
    using assms(1) unfolding encode-exactly-one-def other split list.simps
    by (cases encode-sum-0-1-main (v # vs), auto)
  let ?vars = v # vs
  define vars' where vars' = ?vars
  from assms(3) other have dist: distinct (map fst ?vars) by auto
  have main: eval-all  $\alpha\ conds \wedge (\alpha\ zero \longleftrightarrow \neg (\exists x \in fst\ 'set\ ?vars. \alpha\ x))$ 
   $\wedge (\alpha\ on \longleftrightarrow (\exists! x \in fst\ 'set\ ?vars. \alpha\ x))$ 
  apply (rule encode-sum-0-1-main[OF res - - dist, of True])
  subgoal for i x ze on re using assms(2)[of Suc i x ze on] unfolding other
  by (simp add: nth-via-drop)
  by auto
  hence conds: eval-all  $\alpha\ conds$  and zero:  $\alpha\ zero \longleftrightarrow \neg (\exists x \in fst\ 'set\ ?vars.$ 
 $\alpha\ x)$ 
  and on:  $\alpha\ on \longleftrightarrow (\exists! x \in fst\ 'set\ ?vars. \alpha\ x)$  by auto
  have one: eval  $\alpha\ one \longleftrightarrow (\exists! x \in fst\ 'set\ vars. \alpha\ x)$ 
  unfolding one
  apply (simp)
  using assms(3)
  unfolding zero on other vars'-def[symmetric] by simp blast
  show ?thesis using one conds by auto
next
  case empty
  with assms have one = enc-False by (auto simp: encode-exactly-one-def)
  hence eval  $\alpha\ one = False$  by auto
  with assms empty show ?thesis by (auto simp: encode-exactly-one-def)
qed (insert assms, auto simp: encode-exactly-one-def)
qed

```

lemma *encode-exactly-one-sound*: **assumes** *encode-exactly-one vars = (one, conds)*
and *distinct (map fst vars)*
and *eval α one*
and *eval-all α conds*
shows $\exists! x \in \text{fst } \text{' set vars. } \alpha x$
proof –
consider (*empty*) *vars = []* | (*single*) *x ze on* **where** *vars = [(x,ze,on)]*
| (*other*) *x ze on v vs* **where** *vars = (x,ze,on) # v # vs*
by (*cases vars; cases tl vars; auto*)
thus *?thesis*
proof *cases*
case (*other x ze' on' v vs*)
obtain *on zero* **where** *res: encode-sum-0-1-main (v # vs) = (conds, zero, on)*

and *one: one = enc-ite x zero on*
using *assms(1) unfolding encode-exactly-one-def other split list.simps*
by (*cases encode-sum-0-1-main (v # vs), auto*)
let *?vars = v # vs*
define *vars' where vars' = ?vars*
from *assms(2) other* **have** *dist: distinct (map fst ?vars)* **by** *auto*
have *main: eval-all α conds \wedge (α zero \longleftrightarrow \neg ($\exists x \in \text{fst } \text{' set ?vars. } \alpha x$))*
 \wedge (α on \longleftrightarrow ($\exists! x \in \text{fst } \text{' set ?vars. } \alpha x$))
by (*rule encode-sum-0-1-main[OF res - assms(4) dist, of False], auto*)
hence *conds: eval-all α conds* **and** *zero: α zero \longleftrightarrow \neg ($\exists x \in \text{fst } \text{' set ?vars.$*
 αx)
and *on: α on \longleftrightarrow ($\exists! x \in \text{fst } \text{' set ?vars. } \alpha x$)* **by** *auto*
have *one: eval α one \longleftrightarrow ($\exists! x \in \text{fst } \text{' set vars. } \alpha x$)*
unfolding *one*
apply (*simp*)
using *assms(2)*
unfolding *zero on other vars'-def[symmetric]* **by** *simp blast*
with *assms* **show** *?thesis* **by** *auto*
next
case *empty*
with *assms* **have** *one = enc-False* **by** (*auto simp: encode-exactly-one-def*)
hence *eval α one = False* **by** *auto*
with *assms empty* **show** *?thesis* **by** (*auto simp: encode-exactly-one-def*)
qed (*insert assms, auto simp: encode-exactly-one-def*)
qed

lemma *encodeGammaCond[simp]*: *eval α (encodeGammaCond gam eps s ns) =*
 $(\alpha \text{ gam} \longrightarrow (\alpha \text{ eps} \longrightarrow \text{ns}) \wedge (\neg \alpha \text{ eps} \longrightarrow \text{s}))$
by (*cases ns; cases s, auto*)

lemma *eval-all-append[simp]*: *eval-all α (fs @ gs) = (eval-all α fs \wedge eval-all α gs)*

by *auto*


```

lemma eval-all-Cons[simp]: eval-all  $\alpha$  (f # gs) = (eval  $\alpha$  f  $\wedge$  eval-all  $\alpha$  gs)
  by auto

lemma eval-all-concat[simp]: eval-all  $\alpha$  (concat fs) = ( $\forall$  f  $\in$  set fs. eval-all  $\alpha$  f)
  by auto

lemma eval-all-maps[simp]: eval-all  $\alpha$  (List.maps f fs) = ( $\forall$  g  $\in$  set fs. eval-all  $\alpha$  (f g))
  unfolding List.maps-def eval-all-concat by auto
end

context ms-encoder
begin

context
  fixes s t :: nat  $\Rightarrow$  'a
    and n m :: nat
    and S NS :: 'a rel
    and cs cns
assumes cs:  $\bigwedge$  i j. cs i j = ((s i, t j)  $\in$  S)
  and cns:  $\bigwedge$  i j. cns i j = ((s i, t j)  $\in$  NS)
begin

lemma encoding-sound:
  assumes eval15: eval-all v (formula15 cs cns n m)
  shows (mset (map s [0 ..< n]), mset (map t [0 ..< m]))  $\in$  ns-mul-ext NS S
    eval-all v (formula16 cs cns n m)  $\implies$  (mset (map s [0 ..< n]), mset (map t [0
    ..< m]))  $\in$  s-mul-ext NS S
proof -
  from eval15[unfolded formula15-def]
  have eval14: eval-all v (formula14 n m) by auto
  define property where property i = v (Epsilon i) for i
  define j-of-i :: nat  $\Rightarrow$  nat
    where j-of-i i = (THE j. j < m  $\wedge$  v (Gamma i j)) for i
  define i-of-j :: nat  $\Rightarrow$  nat
    where i-of-j j = (THE i. i < n  $\wedge$  v (Gamma i j)) for j
  define xs1 where xs1 = filter ( $\lambda$  i. property i) [0 ..< n]
  define xs2 where xs2 = filter ( $\lambda$  i.  $\neg$  property i) [0 ..< n]
  define ys1 where ys1 = map j-of-i xs1
  define ys2 where ys2 = filter ( $\lambda$  j. j  $\notin$  set ys1) [0 ..< m]
  let ?xs1 = map s xs1
  let ?xs2 = map s xs2
  let ?ys1 = map t ys1
  let ?ys2 = map t ys2
  {
    fix i
    assume *: i < n v (Epsilon i)
    let ?vars = map ( $\lambda$ j. (Gamma i j, AuxZeroIJ i j, AuxOneIJ i j)) [0..<m]
    obtain one conds where enc: encode-exactly-one ?vars = (one,conds) by force
  }

```

```

have dist: distinct (map fst ?vars) unfolding map-map o-def fst-conv
  unfolding distinct-map by (auto simp: inj-on-def)
have eval-all v (enc-impl (Epsilon i) one # conds)
using eval14[unfolded formula14-def Let-def eval-all-append, unfolded eval-all-maps,
THEN conjunct2] *(1) enc by force
with * have eval v one eval-all v conds by auto
from encode-exactly-one-sound[OF enc dist this]
have 1:  $\exists! x. x \in \text{set } (\text{map } (\lambda j. \text{Gamma } i j) [0..<m]) \wedge v x$ 
  by (simp add: image-comp)
have 2:  $(\exists! x. x \in \text{set } (\text{map } (\lambda j. \text{Gamma } i j) [0..<m]) \wedge v x) =$ 
   $(\exists! j. j < m \wedge v (\text{Gamma } i j))$  by fastforce
have 3:  $\exists! j. j < m \wedge v (\text{Gamma } i j)$  using 1 2 by auto
have j-of-i  $i < m \wedge v (\text{Gamma } i (j\text{-of-}i\ i))$ 
  using 3 unfolding j-of-i-def
  by (metis (no-types, lifting) the-equality)
note this 3
} note j-of-i = this
{
  fix j
  assume *:  $j < m$ 
  let ?vars = map ( $\lambda i. (\text{Gamma } i j, \text{AuxZeroJI } i j, \text{AuxOneJI } i j)$ ) [0..<n]
  have dist: distinct (map fst ?vars) unfolding map-map o-def fst-conv
    unfolding distinct-map by (auto simp: inj-on-def)
  obtain one conds where enc: encode-exactly-one ?vars = (one, conds) by force
  have eval-all v (one # conds)
  using eval14[unfolded formula14-def Let-def eval-all-append, unfolded eval-all-maps,
THEN conjunct1] *(1) enc by force
  hence eval v one eval-all v conds by auto
  from encode-exactly-one-sound[OF enc dist this]
  have 1:  $\exists! x. x \in \text{set } (\text{map } (\lambda i. \text{Gamma } i j) [0..<n]) \wedge v x$ 
    by (simp add: image-comp)
  have 2:  $(\exists! x. x \in \text{set } (\text{map } (\lambda i. \text{Gamma } i j) [0..<n]) \wedge v x) =$ 
     $(\exists! i. i < n \wedge v (\text{Gamma } i j))$  by fastforce
  have 3:  $\exists! i. i < n \wedge v (\text{Gamma } i j)$  using 1 2 by auto
  have i-of-j  $j < n \wedge v (\text{Gamma } (i\text{-of-}j\ j) j)$ 
    using 3 unfolding i-of-j-def
    by (metis (no-types, lifting) the-equality)
  note this 3
} note i-of-j = this

have len: length ?xs1 = length ?ys1
  unfolding ys1-def by simp
note goals = len
{
  fix k
  define i where  $i = \text{xs1 } ! k$ 
  assume  $k < \text{length } ?ys1$ 
  hence k:  $k < \text{length } \text{xs1}$  using len by auto
  hence  $i \in \text{set } \text{xs1}$  using i-def by simp

```

```

hence ir: i < n v (Epsilon i)
  unfolding xs1-def property-def by auto
from j-of-i this
have **: j-of-i i < m ∧ v (Gamma i (j-of-i i)) by auto
have ys1k: ?ys1 ! k = t (j-of-i i) unfolding i-def ys1-def using k by auto
have xs1k: ?xs1 ! k = s i unfolding i-def using k by auto
from eval15 have ∀i∈{0..<n}.
  ∀j∈{0..<m}. v (Gamma i j) → (v (Epsilon i) → cns i j)
  unfolding formula15-def Let-def eval-all-append eval-all-maps
  by (auto split: if-splits)
hence cns i (j-of-i i) using ** ir by auto
then have (?xs1 ! k, ?ys1 ! k) ∈ NS
  unfolding xs1k ys1k using cns[of i (j-of-i i)] by (auto split: if-splits)
} note step2 = this
note goals = goals this
have xexp : mset (map s [0..<n]) = mset ?xs1 + mset ?xs2
  unfolding xs1-def xs2-def
  using mset-map-filter
  by metis
note goals = goals this
{
  fix i
  assume i < n property i
  hence i-of-j (j-of-i i) = i
    using i-of-j j-of-i[of i] unfolding property-def by auto
} note i-of-j-of-i = this
have mset ys1 = mset (filter (λj. j ∈ set (map j-of-i xs1)) [0..<m])
  (is mset ?l = mset ?r)
proof -
  have dl: distinct ?l unfolding ys1-def xs1-def distinct-map
  proof
    show distinct (filter property [0..<n]) by auto
    show inj-on j-of-i (set (filter property [0..<n]))
      by (intro inj-on-inverseI[of - i-of-j], insert i-of-j-of-i, auto)
  qed
  have dr: distinct ?r by simp
  have id: set ?l = set ?r unfolding ys1-def xs1-def using j-of-i i-of-j
    by (auto simp: property-def)
  from dl dr id show ?thesis using set-eq-iff-mset-eq-distinct by blast
qed
hence ys1: mset (map t ys1) = mset (map t ?r) by simp
have yeyp: mset (map t [0..<m]) = mset ?ys1 + mset ?ys2
  unfolding ys1 ys2-def unfolding ys1-def mset-map-filter ..
note goals = goals this
{
  fix y
  assume y ∈ set ?ys2
  then obtain j where j: j ∈ set ys2 and y: y = t j by auto
  from j[unfolded ys2-def ys1-def]

```

```

have  $j: j < m$  and  $nmem: j \notin \text{set}(\text{map } j\text{-of-}i \text{ } xs1)$  by auto
let  $?i = i\text{-of-}j \ j$ 
from  $i\text{-of-}j[OF \ j]$  have  $i: ?i < n$  and  $gamm: v(\text{Gamma } ?i \ j)$  by auto
from  $\text{eval15}[\text{unfolded formula15-def Let-def eval-all-append eval-all-maps}] \ i \ j$ 
gamm
have  $\neg v(\text{Epsilon } ?i) \implies cs \ ?i \ j$  by (force split: if-splits)
moreover have  $\text{not}: \neg v(\text{Epsilon } ?i)$  using  $nmem \ i \ j \ i\text{-of-}j \ j\text{-of-}i$ 
unfolding  $xs1\text{-def property-def}$ 
by (metis atLeast0LessThan filter-set imageI lessThan-iff list.set-map member-filter set-upt)
ultimately have  $cs \ ?i \ j$  by simp
hence  $sy: (s \ ?i, y) \in S$  unfolding  $y$  using  $cs[\text{of } ?i \ j]$  by (auto split: if-splits)
from  $\text{not } i$  have  $?i \in \text{set } xs2$  unfolding  $xs2\text{-def property-def}$  by auto
hence  $s \ ?i \in \text{set } ?xs2$  by simp
hence  $\exists x \in \text{set } ?xs2. (x, y) \in S$  using  $sy$  by auto
}
note  $\text{goals} = \text{goals this}$ 

```

```

show  $(\text{mset}(\text{map } s \ [0 \ ..< \ n]), \text{mset}(\text{map } t \ [0 \ ..< \ m])) \in ns\text{-mul-ext } NS \ S$ 
by (rule ns-mul-ext-intro[OF goals(3,4,1,2,5)])

```

```

assume  $\text{eval-all } v(\text{formula16 } cs \ cns \ n \ m)$ 
from  $\text{this}[\text{unfolded formula16-def Let-def}]$ 
obtain  $i$  where  $i: i < n$  and  $v: \neg v(\text{Epsilon } i)$  by auto
hence  $i \in \text{set } xs2$  unfolding  $xs2\text{-def property-def}$  by auto
hence  $?xs2 \neq []$  by auto
note  $\text{goals} = \text{goals this}$ 

```

```

show  $(\text{mset}(\text{map } s \ [0 \ ..< \ n]), \text{mset}(\text{map } t \ [0 \ ..< \ m])) \in s\text{-mul-ext } NS \ S$ 
by (rule s-mul-ext-intro[OF goals(3,4,1,2,6,5)])

```

qed

```

lemma bex1-cong:  $X = Y \implies (\bigwedge x. x \in Y \implies P \ x = Q \ x) \implies (\exists !x. x \in X \wedge P \ x) = (\exists !x. x \in Y \wedge Q \ x)$ 
by auto

```

lemma *encoding-complete*:

```

assumes  $(\text{mset}(\text{map } s \ [0 \ ..< \ n]), \text{mset}(\text{map } t \ [0 \ ..< \ m])) \in ns\text{-mul-ext } NS \ S$ 
shows  $(\exists v. \text{eval-all } v(\text{formula15 } cs \ cns \ n \ m) \wedge$ 

```

```

 $((\text{mset}(\text{map } s \ [0 \ ..< \ n]), \text{mset}(\text{map } t \ [0 \ ..< \ m])) \in s\text{-mul-ext } NS \ S \implies \text{eval-all } v(\text{formula16 } cs \ cns \ n \ m))$ 

```

proof –

```

let  $?S = (\text{mset}(\text{map } s \ [0 \ ..< \ n]), \text{mset}(\text{map } t \ [0 \ ..< \ m])) \in s\text{-mul-ext } NS \ S$ 

```

```

from  $ns\text{-mul-ext-elim}[OF \ \text{assms}] \ s\text{-mul-ext-elim}[\text{of } \text{mset}(\text{map } s \ [0 \ ..< \ n]) \ \text{mset}(\text{map } t \ [0 \ ..< \ m])] \ NS \ S$ 

```

```

obtain  $Xs1 \ Xs2 \ Ys1 \ Ys2$  where

```

```

 $eq1: \text{mset}(\text{map } s \ [0 \ ..< \ n]) = \text{mset } Xs1 + \text{mset } Xs2$  and

```

```

 $eq2: \text{mset}(\text{map } t \ [0 \ ..< \ m]) = \text{mset } Ys1 + \text{mset } Ys2$  and

```

```

 $len: \text{length } Xs1 = \text{length } Ys1$  and

```

```

 $ne: ?S \implies Xs2 \neq []$  and

```

```

NS:  $\bigwedge i. i < \text{length } Ys1 \implies (Xs1 ! i, Ys1 ! i) \in NS$  and
S:  $\bigwedge y. y \in \text{set } Ys2 \implies \exists x \in \text{set } Xs2. (x, y) \in S$ 
by blast
from mset-map-split[OF eq1] obtain xs1 xs2 where
  xs: mset [0..<n] = mset xs1 + mset xs2
and xs1: Xs1 = map s xs1
and xs2: Xs2 = map s xs2 by auto
from mset-map-split[OF eq2] obtain ys1 ys2 where
  ys: mset [0..<m] = mset ys1 + mset ys2
and ys1: Ys1 = map t ys1
and ys2: Ys2 = map t ys2 by auto
from xs have dist-xs: distinct (xs1 @ xs2)
  by (metis distinct-upt mset-append mset-eq-imp-distinct-iff)
from xs have un-xs: set xs1  $\cup$  set xs2 = {..\cup set ys2 = {..\wedge xs ! p = i) for i
and xs :: nat list
from dist-xs dist-ys have distinct xs1 distinct ys1 by auto
{
  fix xs :: nat list and x
  assume dist: distinct xs and x: x  $\in$  set xs
  hence one:  $\exists ! i. i < \text{length } xs \wedge xs ! i = x$  by (rule distinct-Ex1)
  from theI'[OF this, folded pos-of-def]
  have pos-of xs x < length xs xs ! pos-of xs x = x by auto
  note this one
} note pos = this
note p-xs = pos[OF <distinct xs1>]
note p-ys = pos[OF <distinct ys1>]
define i-of-j2 where i-of-j2 j = (SOME i. i  $\in$  set xs2  $\wedge$  cs i j) for j
define v' :: PropVar  $\Rightarrow$  bool where
  v' x = (case x of
    Epsilon i  $\Rightarrow$  i  $\in$  set xs1
  | Gamma i j  $\Rightarrow$  (i  $\in$  set xs1  $\wedge$  j  $\in$  set ys1  $\wedge$  i = xs1 ! pos-of ys1 j
     $\vee$  i  $\in$  set xs2  $\wedge$  j  $\in$  set ys2  $\wedge$  i = i-of-j2 j)) for x
define v :: PropVar  $\Rightarrow$  bool where
  v x = (case x of
    AuxZeroJI i j  $\Rightarrow$  ( $\neg$  Bex (set (drop i (map ( $\lambda$ i. (Gamma i j)) [0..<n]))) v')
  | AuxOneJI i j  $\Rightarrow$  ( $\exists !$ y. y  $\in$  set (drop i (map ( $\lambda$ i. (Gamma i j)) [0..<n])))  $\wedge$  v'
y)
  | AuxZeroIJ i j  $\Rightarrow$  ( $\neg$  Bex (set (drop j (map ( $\lambda$ j. (Gamma i j)) [0..<m]))) v')
  | AuxOneIJ i j  $\Rightarrow$  ( $\exists !$ y. y  $\in$  set (drop j (map ( $\lambda$ j. (Gamma i j)) [0..<m])))  $\wedge$ 
v' y)
  | -  $\Rightarrow$  v' x) for x
note v-defs = v-def v'-def
{

```

```

fix j
assume j2: j ∈ set ys2
from j2 have t j ∈ set Ys2 unfolding ys2 by auto
from S[OF this, unfolded xs2] have ∃ i. i ∈ set xs2 ∧ cs i j
  by (auto simp: cs)
from someI-ex[OF this, folded i-of-j2-def]
have *: i-of-j2 j ∈ set xs2 cs (i-of-j2 j) j by auto
hence v (Gamma (i-of-j2 j) j) unfolding v-defs using j2 by auto
note * this
} note j-ys2 = this
{
fix j
assume j1: j ∈ set ys1
let ?pj = pos-of ys1 j
from p-ys[OF j1] have pj: ?pj < length Ys1 and yj: ys1 ! ?pj = j
  unfolding ys1 by auto
have pj': ?pj < length Xs1 using len pj by auto
from NS[OF pj] have (Xs1 ! ?pj, Ys1 ! ?pj) ∈ NS .
also have Ys1 ! ?pj = t j using pj unfolding ys1 using yj by auto
also have Xs1 ! ?pj = s (xs1 ! ?pj) using pj' unfolding xs1 by auto
finally have cns: cns (xs1 ! ?pj) j unfolding cns .
have mem: xs1 ! ?pj ∈ set xs1 using pj' unfolding xs1 by auto
have v: v (Gamma (xs1 ! ?pj) j)
  unfolding v-defs using j1 mem by auto
note mem cns v
} note j-ys1 = this
have 14: eval-all v (formula14 n m)
  unfolding formula14-def Let-def eval-all-append eval-all-maps
proof (intro conjI ballI, goal-cases)
case (1 j f)
then obtain one cands where j: j < m and f: f ∈ set (one # cands)
  and enc: encode-exactly-one (map (λi. (Gamma i j, AuxZeroJI i j, AuxOneJI
i j)) [0..<n]) = (one, cands) (is ?e = -)
  by (cases ?e, auto)
have eval-all v cands ∧
  eval v one = (∃!x. x ∈ fst ' set (map (λi. (Gamma i j, AuxZeroJI i j,
AuxOneJI i j)) [0..<n]) ∧ v x)
  apply (rule encode-exactly-one-complete[OF enc])
  subgoal for i y ze on
  proof (goal-cases)
  case 1
  hence ze: ze = AuxZeroJI i j and on: on = AuxOneJI i j by auto
  have id: fst ' set (drop i (map (λi. (Gamma i j, AuxZeroJI i j, AuxOneJI i
j)) [0..<n]))
  = set (drop i (map (λi. (Gamma i j)) [0..<n]))
  unfolding set-map[symmetric] drop-map by simp
  show ?thesis unfolding ze on id unfolding v-def drop-map
  by (intro conjI, force, simp, intro bex1-cong refl, auto)
qed

```

```

    subgoal by (auto simp: distinct-map intro: inj-onI)
  done
  also have fst ' set (map (λi. (Gamma i j, AuxZeroIJ i j, AuxOneIJ i j)) [0..<n])
    = (λi. Gamma i j) ' set [0..<n] unfolding set-map image-comp o-def by
auto
  also have (∃!x. x ∈ ... ∧ v x) = True unfolding eq-True
  proof -
    from j un-ys have j ∈ set ys1 ∨ j ∈ set ys2 by auto
    thus ∃!x. x ∈ (λi. Gamma i j) ' set [0..<n] ∧ v x
    proof
      assume j: j ∈ set ys2
      from j-ys2[OF j] un-xs have i-of-j2 j ∈ {0..<n} by auto
      from this j-ys2[OF j] dist-ys j
      show ?thesis
        by (intro ex1I[of - (Gamma (i-of-j2 j) j)], force, auto simp: v-defs)
    next
      assume j: j ∈ set ys1
      from j-ys1[OF j] un-xs have xs1 ! pos-of-ys1 j ∈ {0..<n} by auto
      from this j-ys1[OF j] dist-ys j
      show ?thesis
        by (intro ex1I[of - (Gamma (xs1 ! pos-of-ys1 j) j)], force, auto simp: v-defs)
    qed
  qed
  finally show ?case using 1 f by auto
next
  case (2 i f)
  then obtain one cands where i: i < n and f: f ∈ set (enc-impl (Epsilon i)
one # cands)
  and enc: encode-exactly-one (map (λj. (Gamma i j, AuxZeroIJ i j, AuxOneIJ
i j)) [0..<m]) = (one, cands) (is ?e = -)
  by (cases ?e, auto)
  have eval-all v cands ∧
    eval v one = (∃!x. x ∈ fst ' set (map (λj. (Gamma i j, AuxZeroIJ i j,
AuxOneIJ i j)) [0..<m]) ∧ v x)
  apply (rule encode-exactly-one-complete[OF enc])
  subgoal for j y ze on
  proof (goal-cases)
    case 1
    hence ze: ze = AuxZeroIJ i j and on: on = AuxOneIJ i j by auto
    have id: fst ' set (drop j (map (λj. (Gamma i j, AuxZeroIJ i j, AuxOneIJ i
j)) [0..<m]))
    = set (drop j (map (λj. (Gamma i j)) [0..<m]))
    unfolding set-map[symmetric] drop-map by simp
    show ?thesis unfolding ze on id unfolding v-def drop-map
    by (intro conjI, force, simp, intro bex1-cong refl, auto)
  qed
  subgoal by (auto simp: distinct-map intro: inj-onI)
  done
  also have fst ' set (map (λj. (Gamma i j, AuxZeroIJ i j, AuxOneIJ i j))

```

```

[0..<m])
  = (λj. Gamma i j) ‘ set [0..<m] unfolding set-map image-comp o-def by
auto
finally have cands: eval-all v cands
  and eval v one = (∃!x. x ∈ Gamma i ‘ set [0..<m] ∧ v x) by auto
note this(2)
also have v (Epsilon i) ⇒ ... = True unfolding eq-True
proof –
  assume v: v (Epsilon i)
  hence i-xs: i ∈ set xs1 i ∉ set xs2 unfolding v-defs using dist-xs by auto
  from this[unfolded set-conv-nth] obtain p where p1: p < length xs1
    and xpi: xs1 ! p = i by auto
  define j where j = ys1 ! p
  from p1 len have p2: p < length ys1 unfolding xs1 ys1 by auto
  hence j: j ∈ set ys1 unfolding j-def by auto
  from p-ys[OF j] p2 have pp: pos-of ys1 j = p by (auto simp: j-def)
  from j un-ys have jm: j < m by auto
  have v: v (Gamma i j) unfolding v-defs using j pp xpi i-xs by simp
  {
    fix k
    assume vk: v (Gamma i k)
    from vk[unfolded v-defs] i-xs
    have k: k ∈ set ys1 and ik: i = xs1 ! pos-of ys1 k by auto
    from p-ys[OF k] ik xpi have id: pos-of ys1 k = p
      by (metis ‹distinct xs1› len length-map nth-eq-iff-index-eq p1 xs1 ys1)
    have k = ys1 ! pos-of ys1 k using p-ys[OF k] by auto
    also have ... = j unfolding id j-def ..
    finally have k = j .
  } note unique = this
  show ∃!j. j ∈ Gamma i ‘ set [0..<m] ∧ v j
    by (intro ex1I[of - Gamma i j], use jm v in force, use unique in auto)
qed
finally show ?case using 2 f cands enc by auto
qed
{
  fix i j
  assume i: i < n and j: j < m
  assume v: v (Gamma i j)
  have strict: ¬ v (Epsilon i) ⇒ cs i j using i j v j-ys2[of j] unfolding v-defs
by auto
  {
    assume v (Epsilon i)
    hence i': i ∈ set xs1 i ∉ set xs2 unfolding v-defs using dist-xs by auto
    with v have j': j ∈ set ys1 unfolding v-defs using dist-ys by auto
    from v[unfolded v-defs] i' have ii: i = xs1 ! pos-of ys1 j by auto
    from j-ys1[OF j', folded ii] have cns i j by auto
  }
  note strict this
} note compare = this

```



```

have 15: eval-all v (formula15 cs cns n m)
unfolding formula15-def Let-def eval-all-maps eval-all-append using 14 compare
by auto
{
  assume ?S
  have 16:  $\exists x \in \{0..<n\}, \neg v (Epsilon\ x)$ 
  by (rule beXI[of - hd xs2]; insert ne[OF ‹?S›] xs2 un-xs dist-xs; cases xs2, auto
simp: v-defs)
  have eval-all v (formula16 cs cns n m)
  unfolding formula16-def Let-def using 15 16 by simp
}
with 15 show ?thesis by blast
qed

```

```

lemma formula15: (mset (map s [0 ..<n]), mset (map t [0 ..<m]))  $\in$  ns-mul-ext
NS S
 $\longleftrightarrow$  ( $\exists v$ . eval-all v (formula15 cs cns n m))
using encoding-sound encoding-complete by blast

```

```

lemma formula16: (mset (map s [0 ..<n]), mset (map t [0 ..<m]))  $\in$  s-mul-ext
NS S
 $\longleftrightarrow$  ( $\exists v$ . eval-all v (formula16 cs cns n m))
using encoding-sound encoding-complete s-ns-mul-ext[of - - NS S]
unfolding formula16-def Let-def eval-all-Cons by blast
end

```

```

lemma encode-mul-ext: assumes encode-mul-ext f xs ys = ( $\varphi_S, \varphi_{NS}$ )
shows mul-ext f xs ys = (( $\exists v$ . eval v  $\varphi_S$ ), ( $\exists v$ . eval v  $\varphi_{NS}$ ))
proof -
  have xs: mset xs = mset (map ( $\lambda i. xs ! i$ ) [0 ..<length xs]) by (simp add:
map-nth)
  have ys: mset ys = mset (map ( $\lambda i. ys ! i$ ) [0 ..<length ys]) by (simp add:
map-nth)
  from assms[unfolded encode-mul-ext-def Let-def, simplified]
  have phis:  $\varphi_{NS} = enc-conj$  (formula15 ( $\lambda i j. fst (f (xs ! i) (ys ! j))$ )) ( $\lambda i j. snd$ 
( $f (xs ! i) (ys ! j)$ )) (length xs) (length ys)
   $\varphi_S = enc-conj$  (formula16 ( $\lambda i j. fst (f (xs ! i) (ys ! j))$ )) ( $\lambda i j. snd$ 
( $f (xs ! i) (ys ! j)$ )) (length xs) (length ys)
  by (auto simp: formula16-def)
  show ?thesis unfolding mul-ext-def Let-def unfolding xs ys prod.inject phis
enc-sound
  by (intro conjI; rule formula15 formula16, auto)
qed
end

```

4.4 Encoding into Propositional Formulas

```

global-interpretation pf-encoder: ms-encoder
Disj []

```

```

Conj []
λ x. Prop x
λ x. Neg (Prop x)
λ x y. Equiv (Prop x) (Neg (Prop y))
λ x y z. Equiv (Prop x) (Conj [Prop y, Neg (Prop z)])
λ x y z u. Equiv (Prop x) (Disj [Conj [Prop y, Prop z], Conj [Neg (Prop y), Prop
u]])
λ x y z. Disj [Conj [Prop x, Prop y], Conj [Neg (Prop x), Prop z]]
λ x f. Impl (Prop x) f
λ x y. Impl (Prop x) (Prop y)
λ x y. Neg (Conj [Prop x, Prop y])
λ xs. Neg (Conj (map Prop xs))
Conj
eval
defines
  pf-encode-sum-0-1-main = pf-encoder.encode-sum-0-1-main and
  pf-encode-exactly-one = pf-encoder.encode-exactly-one and
  pf-encodeGammaCond = pf-encoder.encodeGammaCond and
  pf-formula14 = pf-encoder.formula14 and
  pf-formula15 = pf-encoder.formula15 and
  pf-formula16 = pf-encoder.formula16 and
  pf-encode-mul-ext = pf-encoder.encode-mul-ext
by (unfold-locales, auto)

```

The soundness theorem of the propositional formula encoder

```
thm pf-encoder.encode-mul-ext
```

4.5 Size of Propositional Formula Encoding is Quadratic

lemma *size-pf-encode-sum-0-1-main*: **assumes** *pf-encode-sum-0-1-main vars = (conds, one, zero)*

and *vars ≠ []*

shows *sum-list (map size-pf conds) = 16 * length vars - 12*

using *assms*

proof (*induct vars arbitrary: conds one zero rule: pf-encoder.encode-sum-0-1-main.induct*)

case (*1 x zero' one' conds zero one*)

hence *conds = [Equiv (Prop zero) (Neg (Prop x))]* **by** *auto*

thus *?case* **by** *simp*

next

case *Cons: (2 x zero one r rr conds' zero' one')*

let *?triple = (x, zero, one)*

let *?rest = r # rr*

obtain *conds fzero fone* **where** *res: pf-encode-sum-0-1-main ?rest = (conds, fzero, fone)*

by (*cases pf-encode-sum-0-1-main ?rest, auto*)

from *Cons(2)[unfolded pf-encoder.encode-sum-0-1-main.simps res split Let-def]*

have *conds': conds' = Equiv (Prop zero) (Conj [Prop fzero, Neg (Prop x)]) #*

Equiv (Prop one) (Disj [Conj [Prop x, Prop fzero], Conj [Neg (Prop x), Prop fone])] **#** *conds*

by *auto*

```

have sum-list (map size-pf conds') = 16 + sum-list (map size-pf conds)
  unfolding conds' by simp
with Cons(1)[OF res]
show ?case by simp
qed auto

lemma size-pf-encode-exactly-one: assumes pf-encode-exactly-one vars = (one,
conds)
  shows size-pf one + sum-list (map size-pf conds) = 1 + (16 * length vars - 21)
proof (cases vars = [])
  case True
  with assms have size-pf one = 1 conds = []
    by (auto simp add: pf-encoder.encode-exactly-one-def)
  thus ?thesis unfolding True by simp
next
  case False
  then obtain x ze' on' vs where vars: vars = (x,ze',on') # vs by (cases vars;
auto)
  show ?thesis
  proof (cases vs)
    case Nil
    have size-pf one = 1 conds = [] using assms unfolding vars Nil
      by (auto simp add: pf-encoder.encode-exactly-one-def)
    thus ?thesis unfolding vars Nil by simp
  next
  case (Cons v vs')
  obtain on zero where res: pf-encode-sum-0-1-main vs = (conds, zero, on)
    and one: one = Disj [Conj [Prop x, Prop zero], Conj [Neg (Prop x), Prop
on]]
    using assms(1) False Cons unfolding pf-encoder.encode-exactly-one-def vars
      by (cases pf-encode-sum-0-1-main vs, auto)
    from size-pf-encode-sum-0-1-main[OF res]
    have sum: sum-list (map size-pf conds) = (16 * length vars - 28) using Cons
vars by auto
    have one: size-pf one = 8 unfolding one by simp
    show ?thesis unfolding one sum vars Cons by simp
  qed
qed

```

```

lemma sum-list-concat: sum-list (concat xs) = sum-list (map sum-list xs)
  by (induct xs, auto)

```

```

lemma sum-list-triv-cong: assumes length xs = n
  and  $\bigwedge x. x \in \text{set } xs \implies f x = c$ 
shows sum-list (map f xs) = n * c
  by (subst map-cong[OF refl, of - - λ - . c], insert assms, auto simp: sum-list-triv)

```

```

lemma size-pf-formula14: sum-list (map size-pf (pf-formula14 n m)) = m + 3 *

```

$n + m * (n * 16 - 21) + n * (m * 16 - 21)$
proof –
have $sum-list (map size-pf (pf-formula14 n m)) = m * (1 + (16 * n - 21)) + n * (3 + (16 * m - 21))$
unfolding $pf-encoder.formula14-def Let-def sum-list-append map-append map-concat List.maps-def sum-list-concat map-map o-def$
proof ($intro arg-cong2[of - - - (+)]$, $goal-cases$)
case 1
show $?case$
apply ($rule sum-list-triv-cong$, $force$)
subgoal for j
by ($cases pf-encode-exactly-one (map (\lambda i. (Gamma i j, AuxZeroII i j, AuxOneII i j))) [0..<n]$),
 $auto simp: size-pf-encode-exactly-one$)
done
next
case 2
show $?case$
apply ($rule sum-list-triv-cong$, $force$)
subgoal for i
by ($cases pf-encode-exactly-one (map (\lambda j. (Gamma i j, AuxZeroIJ i j, AuxOneIJ i j))) [0..<m]$),
 $auto simp: size-pf-encode-exactly-one$)
done
qed
also have $\dots = m + 3 * n + m * (n * 16 - 21) + n * (m * 16 - 21)$
by ($simp add: algebra-simps$)
finally show $?thesis$.
qed

lemma $size-pf-encodeGammaCond: size-pf (pf-encodeGammaCond gam eps ns s) \leq 4$
by ($cases ns; cases s, auto$)

lemma $size-pf-formula15: sum-list (map size-pf (pf-formula15 cs cns n m)) \leq m + 3 * n + m * (n * 16 - 21) + n * (m * 16 - 21) + 4 * m * n$
proof –
have $sum-list (map size-pf (pf-formula15 cs cns n m)) \leq sum-list (map size-pf (pf-formula14 n m)) + 4 * m * n$
unfolding $pf-encoder.formula15-def Let-def$
apply ($simp add: size-list-conv-sum-list List.maps-def map-concat o-def length-concat sum-list-triv sum-list-concat algebra-simps$)
apply ($rule le-trans, rule sum-list-mono, rule sum-list-mono[of - - \lambda - 4]$)
by ($auto simp: size-pf-encodeGammaCond sum-list-triv$)
also have $\dots = m + 3 * n + m * (n * 16 - 21) + n * (m * 16 - 21) + 4 * m * n$
unfolding $size-pf-formula14$ **by** $auto$
finally show $?thesis$.

qed

lemma *size-pf-formula16*: $\text{sum-list } (\text{map size-pf } (\text{pf-formula16 cs cns n m})) \leq 2 + m + 4 * n + m * (n * 16 - 21) + n * (m * 16 - 21) + 4 * m * n$

proof –

have $\text{sum-list } (\text{map size-pf } (\text{pf-formula16 cs cns n m})) = \text{sum-list } (\text{map size-pf } (\text{pf-formula15 cs cns n m})) + (n + 2)$

unfolding *pf-encoder.formula16-def Let-def* **by** (*simp add: o-def size-list-conv-sum-list sum-list-triv*)

also have $\dots \leq (m + 3 * n + m * (n * 16 - 21) + n * (m * 16 - 21) + 4 * m * n) + (n + 2)$

by (*rule add-right-mono[OF size-pf-formula15]*)

also have $\dots = 2 + m + 4 * n + m * (n * 16 - 21) + n * (m * 16 - 21) + 4 * m * n$ **by** *simp*

finally show *?thesis* .

qed

lemma *size-pf-encode-mul-ext*: **assumes** *pf-encode-mul-ext f xs ys = (φ_S, φ_{NS})*

and *n: n = max (length xs) (length ys)*

and *n0: n ≠ 0*

shows *size-pf φ_S ≤ 36 * n²*

*size-pf φ_{NS} ≤ 36 * n²*

proof –

from *assms[unfolded pf-encoder.encode-mul-ext-def Let-def, simplified]*

have *phis: φ_{NS} = Conj (pf-formula15 (λi j. fst (f (xs ! i) (ys ! j))) (λi j. snd (f (xs ! i) (ys ! j)))) (length xs) (length ys))*

φ_S = Conj (pf-formula16 (λi j. fst (f (xs ! i) (ys ! j))) (λi j. snd (f (xs ! i) (ys ! j)))) (length xs) (length ys))

by (*auto simp: pf-encoder.formula16-def*)

have *size-pf φ_S ≤ 1 + (2 + n + 4 * n + n * (n * 16 - 21) + n * (n * 16 - 21) + 4 * n * n)*

unfolding *phis* **unfolding** *n size-pf.simps*

by (*rule add-left-mono, rule le-trans[OF size-pf-formula16], intro add-mono mult-mono le-refl, auto*)

also have $\dots \leq 36 * n^2 - 24 * n$ **using** *n0* **by** (*cases n; auto simp: power2-eq-square algebra-simps*)

finally show *size-pf φ_S ≤ 36 * n²* **by** *simp*

have *size-pf φ_{NS} ≤ 1 + (n + 4 * n + n * (n * 16 - 21) + n * (n * 16 - 21) + 4 * n * n)*

unfolding *phis* **unfolding** *n size-pf.simps*

apply (*rule add-left-mono*)

apply (*rule le-trans[OF size-pf-formula15]*)

by (*intro max.mono add-mono mult-mono le-refl, auto*)

also have $\dots \leq 36 * n^2 - 25 * n$ **using** *n0* **by** (*cases n; auto simp: power2-eq-square algebra-simps*)

finally show *size-pf φ_{NS} ≤ 36 * n²* **by** *simp*

qed

4.6 Encoding into Conjunctive Normal Form

global-interpretation *cnf-encoder: ms-encoder*

```

[]
[]
λ x. [(x, True)]
λ x. [(x, False)]
λ x y. [(x, True), (y, True)], [(x, False), (y, False)]
λ x y z. [(x, False), (y, True)], [(x, False), (z, False)], [(x, True), (y, False), (z, True)]
λ x y z u. [(x, True), (y, True), (u, False)], [(x, True), (y, False), (z, False)], [(x, False), (y, False), (z, True)], [(x, False), (y, True), (z, True)], [(x, False), (y, True)]
λ x xs. map (λ c. (x, False) # c) xs
λ x y. [(x, False), (y, True)]
λ x y. [(x, False), (y, False)]
λ xs. [map (λ x. (x, False)) xs]

```

concat

eval-cnf

defines

cnf-encode-sum-0-1-main = *cnf-encoder.encode-sum-0-1-main* **and**

cnf-encode-exactly-one = *cnf-encoder.encode-exactly-one* **and**

cnf-encodeGammaCond = *cnf-encoder.encodeGammaCond* **and**

cnf-formula14 = *cnf-encoder.formula14* **and**

cnf-formula15 = *cnf-encoder.formula15* **and**

cnf-formula16 = *cnf-encoder.formula16* **and**

cnf-encode-mul-ext = *cnf-encoder.encode-mul-ext*

by *unfold-locales (force simp: eval-cnf-alt-def)+*

The soundness theorem of the CNF-encoder

thm *cnf-encoder.encode-mul-ext*

4.7 Size of CNF-Encoding is Quadratic

lemma *size-cnf-encode-sum-0-1-main*: **assumes** *cnf-encode-sum-0-1-main vars = (conds, one, zero)*

and *vars ≠ []*

shows *sum-list (map size-cnf conds) = 26 * length vars - 20*

using *assms*

proof (*induct vars arbitrary: conds one zero rule: cnf-encoder.encode-sum-0-1-main.induct*)

case (*1 x zero' one' conds zero one*)

hence *conds = [[[(zero, True), (one, True)], [(zero, False), (one, False)]]]* **by**

auto

hence *sum-list (map size-cnf conds) = 6* **by** (*simp add: size-cnf-def*)

thus *?case* **by** *simp*

next

case *Cons: (2 x zero one r rr conds' zero' one')*

let *?triple = (x, zero, one)*

let *?rest = r # rr*

obtain *conds fzero fone* **where** *res: cnf-encode-sum-0-1-main ?rest = (conds, fzero, fone)*

by (*cases cnf-encode-sum-0-1-main ?rest, auto*)

```

from Cons(2)[unfolded cnf-encoder.encode-sum-0-1-main.simps res split Let-def]
have conds': conds' = [[(zero, False), (fzero, True)], [(zero, False), (x, False)],
[(zero, True), (fzero, False), (x, True)]] #
  [[(one, True), (x, True), (fone, False)], [(one, True), (x, False), (fzero, False)],
[(one, False), (x, False), (fzero, True)],
 [(one, False), (x, True), (fone, True)]] #
  conds
by auto
have sum-list (map size-cnf conds') = 26 + sum-list (map size-cnf conds)
  unfolding conds' by (simp add: size-cnf-def)
with Cons(1)[OF res]
show ?thesis by simp
qed auto

```

```

lemma size-cnf-encode-exactly-one: assumes cnf-encode-exactly-one vars = (one,
conds)
shows size-cnf one + sum-list (map size-cnf conds) ≤ 2 + (26 * length vars -
42) ∧ length one ≤ 2
proof (cases vars = [])
  case True
    with assms have size-cnf one = 1 length one = 1 conds = []
      by (auto simp add: cnf-encoder.encode-exactly-one-def size-cnf-def)
    thus ?thesis unfolding True by simp
  next
    case False
      then obtain x ze' on' vs where vars: vars = (x,ze',on') # vs by (cases vars;
auto)
      show ?thesis
      proof (cases vs)
        case Nil
          have size-cnf one = 2 length one = 1 conds = [] using assms unfolding vars
Nil
          by (auto simp add: cnf-encoder.encode-exactly-one-def size-cnf-def)
          thus ?thesis unfolding vars Nil by simp
        next
          case (Cons v vs')
            obtain on zero where res: cnf-encode-sum-0-1-main vs = (conds, zero, on)
              and one: one = [[(x, True), (on, True)], [(x, False), (zero, True)]]
              using assms(1) False Cons unfolding cnf-encoder.encode-exactly-one-def vars
by (cases cnf-encode-sum-0-1-main vs, auto)
            from size-cnf-encode-sum-0-1-main[OF res]
            have sum: sum-list (map size-cnf conds) = 26 * length vars - 46 using Cons
vars by auto
            have one: size-cnf one = 6 length one = 2 unfolding one by (auto simp add:
size-cnf-def)
            show ?thesis unfolding one sum vars Cons by simp
          qed
        qed

```

lemma *sum-list-mono-const*: **assumes** $\bigwedge x. x \in \text{set } xs \implies f x \leq c$
and $n = \text{length } xs$
shows $\text{sum-list } (\text{map } f \text{ } xs) \leq n * c$
unfolding *assms(2)* **using** *assms(1)*
by (*induct xs; force*)

lemma *size-cnf-formula14*: $\text{sum-list } (\text{map } \text{size-cnf } (\text{cnf-formula14 } n \text{ } m)) \leq 2 * m + 4 * n + m * (26 * n - 42) + n * (26 * m - 42)$
proof –
have $\text{sum-list } (\text{map } \text{size-cnf } (\text{cnf-formula14 } n \text{ } m)) \leq m * (2 + (26 * n - 42)) + n * (4 + (26 * m - 42))$
unfolding *cnf-encoder.formula14-def Let-def sum-list-append map-append map-concat List.maps-def sum-list-concat map-map o-def*
proof ((*intro add-mono; intro sum-list-mono-const*), *goal-cases*)
case (1 j)
obtain *one conds* **where** *cnf: cnf-encode-exactly-one* ($\text{map } (\lambda i. (\text{Gamma } i \text{ } j, \text{AuxZeroII } i \text{ } j, \text{AuxOneII } i \text{ } j)) [0..<n]) = (\text{one}, \text{conds})$ (**is** $?e = -$)
by (*cases ?e, auto*)
show $?case$ **unfolding** *cnf split* **using** *size-cnf-encode-exactly-one[OF cnf]* **by** *auto*
next
case (3 i)
obtain *one conds* **where** *cnf: cnf-encode-exactly-one* ($\text{map } (\lambda j. (\text{Gamma } i \text{ } j, \text{AuxZeroIJ } i \text{ } j, \text{AuxOneIJ } i \text{ } j)) [0..<m]) = (\text{one}, \text{conds})$ (**is** $?e = -$)
by (*cases ?e, auto*)
have *id: size-cnf* ($\text{map } ((\#) (\text{Epsilon } i, \text{False})) \text{one}$) = *size-cnf one* + *length one* **unfolding** *size-cnf-def* **by** (*induct one, auto simp: o-def*)
show $?case$ **unfolding** *cnf split* **using** *size-cnf-encode-exactly-one[OF cnf]* **by** (*simp add: id*)
qed *auto*
also have $\dots = 2 * m + 4 * n + m * (26 * n - 42) + n * (26 * m - 42)$
by (*simp add: algebra-simps*)
finally show $?thesis$.
qed

lemma *size-cnf-encodeGammaCond*: $\text{size-cnf } (\text{cnf-encodeGammaCond } \text{gam } \text{eps } \text{ns } s) \leq 3$
by (*cases ns; cases s, auto simp: size-cnf-def*)

lemma *size-cnf-formula15*: $\text{sum-list } (\text{map } \text{size-cnf } (\text{cnf-formula15 } \text{cs } \text{cns } n \text{ } m)) \leq 2 * m + 4 * n + m * (26 * n - 42) + n * (26 * m - 42) + 3 * n * m$
proof –
have $\text{sum-list } (\text{map } \text{size-cnf } (\text{cnf-formula15 } \text{cs } \text{cns } n \text{ } m)) \leq \text{sum-list } (\text{map } \text{size-cnf } (\text{cnf-formula14 } n \text{ } m)) + 3 * n * m$
unfolding *cnf-encoder.formula15-def Let-def*
apply (*simp add: size-list-conv-sum-list List.maps-def map-concat o-def length-concat sum-list-triv sum-list-concat algebra-simps*)
apply (*rule le-trans, rule sum-list-mono-const[OF - refl], rule sum-list-mono-const[OF*

- refl, of - - 3])
 by (auto simp: size-cnf-encodeGammaCond)
 also have ... $\leq (2 * m + 4 * n + m * (26 * n - 42) + n * (26 * m - 42)) + 3 * n * m$
 by (rule add-right-mono[OF size-cnf-formula14])
 finally show ?thesis .
 qed

lemma size-cnf-formula16: $sum-list (map size-cnf (cnf-formula16 cs cns n m)) \leq 1 + 2 * m + 5 * n + m * (26 * n - 42) + n * (26 * m - 42) + 3 * n * m$

proof -
 have $sum-list (map size-cnf (cnf-formula16 cs cns n m)) = sum-list (map size-cnf (cnf-formula15 cs cns n m)) + (n + 1)$
 unfolding cnf-encoder.formula16-def Let-def by (simp add: o-def size-list-conv-sum-list sum-list-triv size-cnf-def)
 also have ... $\leq (2 * m + 4 * n + m * (26 * n - 42) + n * (26 * m - 42) + 3 * n * m) + (n + 1)$
 by (rule add-right-mono[OF size-cnf-formula15])
 also have ... $= 1 + 2 * m + 5 * n + m * (26 * n - 42) + n * (26 * m - 42) + 3 * n * m$ by simp
 finally show ?thesis .
 qed

lemma size-cnf-concat: $size-cnf (concat xs) = sum-list (map size-cnf xs)$ **unfolding** size-cnf-def
 by (induct xs, auto)

lemma size-cnf-encode-mul-ext: **assumes** $cnf-encode-mul-ext f xs ys = (\varphi_S, \varphi_{NS})$

and $n: n = max (length xs) (length ys)$
and $n0: n \neq 0$
shows $size-cnf \varphi_S \leq 55 * n^2$
 $size-cnf \varphi_{NS} \leq 55 * n^2$
proof -
 let ?fns = $cnf-formula15 (\lambda i j. fst (f (xs ! i) (ys ! j))) (\lambda i j. snd (f (xs ! i) (ys ! j))) (length xs) (length ys)$
 let ?fs = $cnf-formula16 (\lambda i j. fst (f (xs ! i) (ys ! j))) (\lambda i j. snd (f (xs ! i) (ys ! j))) (length xs) (length ys)$
 from **assms**[unfolded cnf-encoder.encode-mul-ext-def Let-def, simplified]
 have $phis: \varphi_{NS} = concat ?fns \varphi_S = concat ?fs$
 by (auto simp: cnf-encoder.formula16-def)
 have $le-s: sum-list (map size-cnf ?fs) \leq 1 + 2 * n + 5 * n + n * (26 * n - 42) + n * (26 * n - 42) + 3 * n * n$
 by (rule le-trans[OF size-cnf-formula16], intro add-mono mult-mono le-refl, insert n, auto)
 have $le-ns: sum-list (map size-cnf ?fns) \leq 2 * n + 4 * n + n * (26 * n - 42) + n * (26 * n - 42) + 3 * n * n$
 by (rule le-trans[OF size-cnf-formula15], intro add-mono mult-mono le-refl, insert n, auto)

```

{
  fix  $\varphi$ 
  assume  $\varphi \in \{\varphi_{NS}, \varphi_S\}$ 
  then obtain  $f$  where  $f \in \{?fs, ?fns\}$  and  $\text{phi}: \varphi = \text{concat } f$  unfolding  $\text{phis}$ 
by auto
  hence size-cnf  $\varphi \leq 1 + 2 * n + 5 * n + n * (26 * n - 42) + n * (26 * n - 42) + 3 * n * n$ 
  unfolding  $\text{phi}$  size-cnf-concat
  using le-s le-ns by auto
  also have ... =  $1 + n * 7 + n * n * 3 + (n * n * 52 - n * 84)$  by (simp
add: algebra-simps)
  also have ...  $\leq n * n * 55$  using n0 by (cases n; auto)
  also have ... =  $55 * n^2$  by (auto simp: power2-eq-square)
  finally have size-cnf  $\varphi \leq 55 * n^2$  .
}
thus size-cnf  $\varphi_{NS} \leq 55 * n^2$  size-cnf  $\varphi_S \leq 55 * n^2$  by auto
qed

```

4.8 Check Executability

The constant 36 in the size-estimation for the PF-encoder is not that bad in comparison to the actual size, since using 34 in the size-estimation would be wrong:

```

value (code) let  $n = 20$  in ( $36 * n^2$ , size-pf (fst (pf-encode-mul-ext ( $\lambda i j. (True, False)$ ) [ $0..<n$ ] [ $0..<n$ ]))),  $34 * n^2$ )

```

Similarly, the constant 55 in the size-estimation for the CNF-encoder is not that bad in comparison to the actual size, since using 51 in the size-estimation would be wrong:

```

value (code) let  $n = 20$  in ( $55 * n^2$ , size-cnf (fst (cnf-encode-mul-ext ( $\lambda i j. (True, False)$ ) [ $0..<n$ ] [ $0..<n$ ]))),  $51 * n^2$ )

```

Example encoding

```

value (code) fst (pf-encode-mul-ext ( $\lambda i j. (i > j, i \geq j)$ ) [ $0..<3$ ] [ $0..<5$ ])
value (code) fst (cnf-encode-mul-ext ( $\lambda i j. (i > j, i \geq j)$ ) [ $0..<3$ ] [ $0..<5$ ])

```

end

5 Deciding the Generalized Multiset Ordering is NP-hard

We prove that satisfiability of conjunctive normal forms (a NP-hard problem) can be encoded into a multiset-comparison problem of linear size. Therefore multiset-set comparisons are NP-hard as well.

theory

Multiset-Ordering-NP-Hard

```

imports
  Multiset-Ordering-More
  Propositional-Formula
  Weighted-Path-Order.Multiset-Extension2-Impl
begin

```

5.1 Definition of the Encoding

The multiset-elements are either annotated variables or indices (of clauses). We basically follow the proof in [4] where these elements are encoded as terms (and the relation is some fixed recursive path order).

```

datatype Annotation = Unsigned | Positive | Negative

```

```

type-synonym 'a ms-elem = ('a × Annotation) + nat

```

```

fun ms-elem-of-lit :: 'a × bool ⇒ 'a ms-elem where
  ms-elem-of-lit (x,True) = Inl (x,Positive)
| ms-elem-of-lit (x,False) = Inl (x,Negative)

```

```

definition vars-of-cnf :: 'a cnf ⇒ 'a list where
  vars-of-cnf = (remdups o concat o map (map fst))

```

We encode a CNF into a multiset-problem, i.e., a quadruple (xs, ys, S, NS) where xs and ys are the lists to compare, and S and NS are underlying relations of the generalized multiset ordering. In the encoding, we add the strict relation S to the non-strict relation NS as this is a somewhat more natural order. In particular, the relations S and NS are precisely those that are obtained when using the mentioned recursive path order of [4].

```

definition multiset-problem-of-cnf :: 'a cnf ⇒
  ('a ms-elem list ×
   'a ms-elem list ×
   ('a ms-elem × 'a ms-elem)list ×
   ('a ms-elem × 'a ms-elem)list) where
  multiset-problem-of-cnf cnf = (let
    xs = vars-of-cnf cnf;
    cs = [0 ..< length cnf];
    S = List.maps (λ i. map (λ l. (ms-elem-of-lit l, Inr i)) (cnf ! i)) cs;
    NS = List.maps (λ x. [(Inl (x,Positive), Inl (x,Unsigned)), (Inl (x,Negative),
Inl (x,Unsigned))]) xs
    in (List.maps (λ x. [Inl (x,Positive), Inl (x,Negative)]) xs,
       map (λ x. Inl (x,Unsigned)) xs @ map Inr cs,
       S, NS @ S))

```

5.2 Soundness of the Encoding

```

lemma multiset-problem-of-cnf:
  assumes multiset-problem-of-cnf cnf = (left, right, S, NSS)
  shows (∃ β. eval-cnf β cnf)

```

$\longleftrightarrow ((mset\ left, mset\ right) \in ns\text{-}mul\text{-}ext\ (set\ NSS)\ (set\ S))$
 $cnf \neq [] \implies (\exists \beta. eval\text{-}cnf\ \beta\ cnf)$
 $\longleftrightarrow ((mset\ left, mset\ right) \in s\text{-}mul\text{-}ext\ (set\ NSS)\ (set\ S))$

proof –

```

define xs where xs = vars-of-cnf cnf
define cs where cs = [0 ..< length cnf]
define NS :: ('a ms-elem × 'a ms-elem)list where NS = concat (map (λ x. [(Inl
(x,Positive), Inl (x,Unsigned)), (Inl (x,Negative), Inl (x,Unsigned))]) xs)
note res = assms[unfolded multiset-problem-of-cnf-def Let-def List.maps-def,
folded xs-def cs-def]
have S: S = concat (map (λ i. map (λ l. (ms-elem-of-lit l, Inr i)) (cnf ! i)) cs)
using res by auto
have NSS: NSS = NS @ S unfolding S NS-def using res by auto
have left: left = concat (map (λ x. [Inl (x,Positive), Inl (x,Negative)])) xs
using res by auto
let ?nsright = map (λ x. Inl (x,Unsigned)) xs
let ?sright = map Inr cs :: 'a ms-elem list
have right: right = ?nsright @ ?sright
using res by auto

```

We first consider completeness: if the formula is sat, then the lists are decreasing w.r.t. the multiset-order.

```

{
assume (∃ β. eval-cnf β cnf)
then obtain β where sat: eval β (formula-of-cnf cnf) unfolding eval-cnf-def
by auto
define f :: 'a ms-elem ⇒ bool where
f = (λ c. case c of (Inl (x,sign)) ⇒ (β x ⇔ sign = Negative))
let ?nsleft = filter f left
let ?sleft = filter (Not o f) left
have id-left: mset left = mset ?nsleft + mset ?sleft by simp
have id-right: mset right = mset ?nsright + mset ?sright unfolding right by
auto
have nsleft: ?nsleft = map (λ x. ms-elem-of-lit (x, ¬ (β x))) xs
unfolding left f-def by (induct xs, auto)
have sleft: ?sleft = map (λ x. ms-elem-of-lit (x, β x)) xs
unfolding left f-def by (induct xs, auto)
have len: length ?nsleft = length ?nsright unfolding nsleft by simp
{
fix i
assume i: i < length ?nsright
define x where x = xs ! i
have x: x ∈ set xs unfolding x-def using i by auto
have (?nsleft ! i, ?nsright ! i) = (ms-elem-of-lit (x, ¬ β x), Inl (x,Unsigned))
unfolding nsleft x-def using i by auto
also have ... ∈ set NS unfolding NS-def using x by (cases β x, auto)
finally have (?nsleft ! i, ?nsright ! i) ∈ set NSS unfolding NSS by auto
} note non-strict = this
{

```

```

fix t
assume t ∈ set ?sright
then obtain i where i: i ∈ set cs and t: t = Inr i by auto
define c where c = cnf ! i
from i have ii: i < length cnf unfolding cs-def by auto
have c: c ∈ set cnf using i unfolding c-def cs-def by auto
from sat[unfolded formula-of-cnf-def] c
have eval β (Disj (map formula-of-lit c)) unfolding o-def by auto
then obtain l where l: l ∈ set c and eval: eval β (formula-of-lit l)
  by auto
obtain x b where l = (x, b) by (cases l, auto)
with eval have lx: l = (x, β x) by (cases b, auto)
from l c lx have x: x ∈ set xs unfolding xs-def vars-of-cnf-def by force
have mem: (ms-elem-of-lit l) ∈ set ?sleft unfolding sleft lx using x by auto
have ∃ s ∈ set ?sleft. (s,t) ∈ set S
proof (intro bexI[OF - mem])
  show (ms-elem-of-lit l, t) ∈ set S
  unfolding t S cs-def using ii l c-def
  by (auto intro!: bexI[of - i])
qed
} note strict = this

have NS: ((mset left, mset right) ∈ ns-mul-ext (set NSS) (set S))
by (intro ns-mul-ext-intro[OF id-left id-right len non-strict strict])
{
  assume ne: cnf ≠ []
  then obtain c where c: c ∈ set cnf by (cases cnf, auto)
  with sat[unfolded formula-of-cnf-def]
  have eval β (Disj (map formula-of-lit c)) by auto
  then obtain x where x: x ∈ set xs
  using c unfolding vars-of-cnf-def xs-def by (cases c; cases snd (hd c);
force)
  have S: ((mset left, mset right) ∈ s-mul-ext (set NSS) (set S))
  proof (intro s-mul-ext-intro[OF id-left id-right len non-strict - strict])
    show ?sleft ≠ [] unfolding sleft using x by auto
  qed
} note S = this
note NS S
} note one-direction = this

```

We next consider soundness: if the lists are decreasing w.r.t. the multiset-order, then the cnf is sat.

```

{
  assume ((mset left, mset right) ∈ ns-mul-ext (set NSS) (set S))
  ∨ ((mset left, mset right) ∈ s-mul-ext (set NSS) (set S))
  hence ((mset left, mset right) ∈ ns-mul-ext (set NSS) (set S))
  using s-ns-mul-ext by auto
  also have ns-mul-ext (set NSS) (set S) = ns-mul-ext (set NS) (set S)
  unfolding NSS set-append by (rule ns-mul-ext-NS-union-S)

```

finally have $(mset\ left, mset\ right) \in ns\text{-}mul\text{-}ext\ (set\ NS)\ (set\ S)$.
from $ns\text{-}mul\text{-}ext\text{-}elim[OF\ this]$
obtain $ns\text{-}left\ s\text{-}left\ ns\text{-}right\ s\text{-}right$
where $id\text{-}left: mset\ left = mset\ ns\text{-}left + mset\ s\text{-}left$
and $id\text{-}right: mset\ right = mset\ ns\text{-}right + mset\ s\text{-}right$
and $len: length\ ns\text{-}left = length\ ns\text{-}right$
and $ns: \bigwedge i. i < length\ ns\text{-}right \implies (ns\text{-}left\ !\ i, ns\text{-}right\ !\ i) \in set\ NS$
and $s: \bigwedge t. t \in set\ s\text{-}right \implies \exists s \in set\ s\text{-}left. (s, t) \in set\ S$ **by** $blast$

This is the satisfying assignment

define β **where** $\beta\ x = (ms\text{-}elem\text{-}of\text{-}lit\ (x, True) \in set\ s\text{-}left)$ **for** x
{
fix c
assume $cnf: c \in set\ cnf$
then obtain i **where** $i: i \in set\ cs$
and $c\text{-}def: c = cnf\ !\ i$
and $ii: i < length\ cnf$
unfolding $cs\text{-}def\ set\text{-}conv\text{-}nth$ **by** $force$

from i **have** $Inr\ i \in \# mset\ right$ **unfolding** $right$ **by** $auto$
from $this[unfolding\ id\text{-}right]$ **have** $Inr\ i \in set\ ns\text{-}right \vee Inr\ i \in set\ s\text{-}right$ **by**
 $auto$
hence $Inr\ i \in set\ s\text{-}right$ **using** $ns[unfolding\ NSS\ NS\text{-}def]$
unfolding $set\text{-}conv\text{-}nth[of\ ns\text{-}right]$ **by** $force$
from $s[OF\ this]$ **obtain** s **where** $sleft: s \in set\ s\text{-}left$ **and** $si: (s, Inr\ i) \in set$
 S **by** $auto$
from $si[unfolding\ S, simplified]$ **obtain** l **where**
 $lc: l \in set\ c$ **and** $sl: s = ms\text{-}elem\text{-}of\text{-}lit\ l$ **unfolding** $c\text{-}def\ cs\text{-}def$ **using** ii **by**
 $blast$
obtain $x\ b$ **where** $lxb: l = (x, b)$ **by** $force$
from $lc\ lxb\ cnf$ **have** $x: x \in set\ xs$ **unfolding** $xs\text{-}def\ vars\text{-}of\text{-}cnf\text{-}def$ **by** $force$
have $\exists l \in set\ c. eval\ \beta\ (formula\text{-}of\text{-}lit\ l)$
proof $(intro\ beXI[OF\ -\ lc])$
from $sleft[unfolding\ sl\ lxb]$
have $mem: ms\text{-}elem\text{-}of\text{-}lit\ (x, b) \in set\ s\text{-}left$ **by** $auto$
have $\beta\ x = b$
proof $(cases\ b)$
case $True$
thus $?thesis$ **unfolding** $\beta\text{-}def$ **using** mem **by** $auto$
next
case $False$
show $?thesis$
proof $(rule\ ccontr)$
assume $\beta\ x \neq b$
with $False$ **have** $\beta\ x$ **by** $auto$
with $False\ mem$
have $ms\text{-}elem\text{-}of\text{-}lit\ (x, True) \in set\ s\text{-}left$
 $ms\text{-}elem\text{-}of\text{-}lit\ (x, False) \in set\ s\text{-}left$
unfolding $\beta\text{-}def$ **by** $auto$

hence *mem*: *ms-elem-of-lit* $(x, b) \in \text{set } s\text{-left}$ **for** *b* **by** (*cases b, auto*)

have *dist*: *distinct left unfolding left*
by (*intro distinct-concat, auto simp: distinct-map xs-def vars-of-cnf-def cs-def intro!: inj-onI*)
from *id-left* **have** *mset left* = *mset (ns-left @ s-left)* **by** *auto*
from *mset-eq-imp-distinct-iff[OF this]* *dist* **have** *set ns-left* \cap *set s-left*
= $\{\}$ **by** *auto*
with *mem* **have** *nmem*: *ms-elem-of-lit* $(x, b) \notin \text{set } ns\text{-left}$ **for** *b* **by** *auto*
have *Inl (x, Unsigned) \in # mset right* **unfolding right using x** **by** *auto*
from *this[unfolded id-right]*
have *Inl (x, Unsigned) \in set ns-right \cup set s-right* **by** *auto*
with *s[unfolded S]* **have** *Inl (x, Unsigned) \in set ns-right* **by** *auto*
with *ns* **obtain** *s* **where** *pair*: $(s, \text{Inl } (x, \text{Unsigned})) \in \text{set } NS$ **and** *sns*:
s \in set ns-left
unfolding set-conv-nth[of ns-right] **using len** **by** *force*
from *pair[unfolded NSS]* **have** *pair*: $(s, \text{Inl } (x, \text{Unsigned})) \in \text{set } NS$ **by**
auto
from *pair[unfolded NS-def, simplified]* **have** *s* = *Inl (x, Positive) \vee s* =
Inl (x, Negative) **by** *auto*
from *sns this nmem[of True] nmem[of False]* **show** *False* **by** *auto*
qed
qed
thus *eval β (formula-of-lit l)* **unfolding lxb** **by** (*cases b, auto*)
qed
}
hence *eval β (formula-of-cnf cnf)* **unfolding formula-of-cnf-def o-def** **by** *auto*
hence $\exists \beta. \text{eval-cnf } \beta \text{ cnf}$ **unfolding eval-cnf-def** **by** *auto*
} **note** *other-direction = this*

from *one-direction other-direction* **show** $(\exists \beta. \text{eval-cnf } \beta \text{ cnf})$
 $\longleftrightarrow ((\text{mset left}, \text{mset right}) \in \text{ns-mul-ext } (\text{set } NSS) (\text{set } S))$
by *blast*
show *cnf $\neq []$* $\implies (\exists \beta. \text{eval-cnf } \beta \text{ cnf})$
 $\longleftrightarrow ((\text{mset left}, \text{mset right}) \in \text{s-mul-ext } (\text{set } NSS) (\text{set } S))$
using one-direction other-direction **by** *blast*
qed

lemma *multiset-problem-of-cnf-mul-ext*:
assumes *multiset-problem-of-cnf cnf = (xs, ys, S, NS)*
and *non-trivial: cnf $\neq []$*
shows $(\exists \beta. \text{eval-cnf } \beta \text{ cnf})$
 $\longleftrightarrow \text{mul-ext } (\lambda a b. ((a, b) \in \text{set } S, (a, b) \in \text{set } NS)) \text{ xs ys} = (\text{True}, \text{True})$
proof –
have $(\exists \beta. \text{eval-cnf } \beta \text{ cnf}) = ((\exists \beta. \text{eval-cnf } \beta \text{ cnf}) \wedge (\exists \beta. \text{eval-cnf } \beta \text{ cnf}))$
by *simp*
also **have** $\dots = (((\text{mset } xs, \text{mset } ys) \in \text{s-mul-ext } (\text{set } NS) (\text{set } S)) \wedge ((\text{mset } xs,$
*mset ys) \in ns-mul-ext (set NS) (set S)))
by (*subst multiset-problem-of-cnf(1)[symmetric, OF assms(1)]*, *subst multi-**

```

set-problem-of-cnf(2)[symmetric, OF assms], simp)
  also have ... = (mul-ext (λ a b. ((a,b) ∈ set S, (a,b) ∈ set NS)) xs ys =
  (True, True))
  unfolding mul-ext-def Let-def by auto
  finally show ?thesis .
qed

```

5.3 Size of Encoding is Linear

```

lemma size-of-multiset-problem-of-cnf: assumes multiset-problem-of-cnf cnf =
(xs, ys, S, NS)
  and size-cnf cnf = s
shows length xs ≤ 2 * s length ys ≤ 2 * s length S ≤ s length NS ≤ 3 * s
proof -
  define vs where vs = vars-of-cnf cnf
  have lus: length vs ≤ s unfolding assms(2)[symmetric] vs-def vars-of-cnf-def
  o-def size-cnf-def
  by (rule order.trans[OF length-remdups-leq], induct cnf, auto)
  have lcnf: length cnf ≤ s using assms(2) unfolding size-cnf-def by auto
  note res = assms(1)[unfolded multiset-problem-of-cnf-def Let-def List.maps-def,
  folded vs-def, simplified]
  have xs: xs = concat (map (λx. [Inl (x, Positive), Inl (x, Negative)]) vs) using
  res by auto
  have length xs ≤ length vs + length vs unfolding xs by (induct vs, auto)
  also have ... ≤ 2 * s using lus by auto
  finally show length xs ≤ 2 * s .
  have length ys = length (map (λx. Inl (x, Unsigned)) vs @ map Inr [0..

```

5.4 Check Executability

```

value (code) case multiset-problem-of-cnf [

```



```

[("x", True), ("y", False)],           — clause 0
[("x", False)],                       — clause 1
[("y", True), ("z", True)],          — clause 2
[("x", True), ("y", True), ("z", False)]] — clause 3
  of (left, right, S, NS) ⇒ ("SAT: ", mul-ext (λ x y. ((x, y) ∈ set S, (x, y) ∈ set
NS)) left right = (True, True),
    "Encoding: ", left, ">mul ", right, "strict element order: ", S, "non-strict:
", NS)

```

end

6 Deciding RPO-constraints is NP-hard

We show that for a given an RPO it is NP-hard to decide whether two terms are in relation, following a proof in [4].

```

theory RPO-NP-Hard
imports
  Multiset-Ordering-NP-Hard
  Weighted-Path-Order.RPO
begin

```

6.1 Definition of the Encoding

```

datatype FSyms = A | F | G | H | U | P | N

```

We slightly deviate from the paper encoding, since we add the three constants U , P , N in order to be able to easily convert an encoded term back to the multiset-element.

```

fun ms-elem-to-term :: 'a cnf ⇒ 'a ms-elem ⇒ (FSyms, 'a + nat)term where

```

```

  ms-elem-to-term cnf (Inr i) = Var (Inr i)
|
  ms-elem-to-term cnf (Inl (x, Unsigned)) = Fun F (Var (Inl x) # Fun U [] #
  map (λ -. Fun A []) cnf)
|
  ms-elem-to-term cnf (Inl (x, Positive)) = Fun F (Var (Inl x) # Fun P [] #
  map (λ i. if (x, True) ∈ set (cnf ! i) then Var (Inr i) else Fun A []) [0 ..<
length cnf])
|
  ms-elem-to-term cnf (Inl (x, Negative)) = Fun F (Var (Inl x) # Fun N [] #
  map (λ i. if (x, False) ∈ set (cnf ! i) then Var (Inr i) else Fun A []) [0 ..<
length cnf])

```

```

definition term-lists-of-cnfnf :: 'a cnf ⇒ (FSyms, 'a + nat)term list × (FSyms, 'a
+ nat)term list where

```

```

  term-lists-of-cnfnf cnf = (case multiset-problem-of-cnfnf of
  (as, bs, S, NS) ⇒
  (map (ms-elem-to-term cnf) as, map (ms-elem-to-term cnf) bs))

```

definition *rpo-constraint-of-cnf* :: 'a cnf \Rightarrow (-,-)term \times (-,-)term **where**
rpo-constraint-of-cnf cnf = (case term-lists-of-cnf cnf of
 (as, bs) \Rightarrow (Fun G as, Fun H bs))

An RPO instance where all symbols are equivalent in precedence and all symbols have multiset-status.

interpretation *trivial-rpo*: *rpo-with-assms* λ f g. (False, True) λ f. True λ -. Mul
 0

by (unfold-locales, auto)

6.2 Soundness of the Encoding

fun *term-to-ms-elem* :: (FSyms, 'a + nat)term \Rightarrow 'a ms-elem **where**
term-to-ms-elem (Var (Inr i)) = Inr i
 | *term-to-ms-elem* (Fun F (Var (Inl x) # Fun U - # ts)) = Inl (x, Unsigned)
 | *term-to-ms-elem* (Fun F (Var (Inl x) # Fun P - # ts)) = Inl (x, Positive)
 | *term-to-ms-elem* (Fun F (Var (Inl x) # Fun N - # ts)) = Inl (x, Negative)
 | *term-to-ms-elem* - = undefined

lemma *term-to-ms-elem-ms-elem-to-term[simp]*: *term-to-ms-elem* (ms-elem-to-term
 cnf x) = x

apply (cases x)

subgoal for a by (cases a, cases snd a, auto)

by auto

lemma (in *rpo-with-assms*) *rpo-vars-term*: *rpo-s* s t \vee *rpo-ns* s t \Longrightarrow *vars-term* s
 \supseteq *vars-term* t

proof (induct s t rule: *rpo.induct*[of - prc prl c n], force, force)

case (\exists f ss y)

thus ?case

by (smt (verit, best) fst-conv *rpo.simps*(\exists) snd-conv subset-eq *term.set-intros*(4))

next

case (λ f ss g ts)

show ?case

proof (cases \exists s \in set ss. *rpo-ns* s (Fun g ts))

case True

from λ (1) True **show** ?thesis **by** auto

next

case False

obtain ps pns **where** prc: prc (f, length ss) (g, length ts) = (ps, pns) **by** force

from False **have** (if (\exists s \in set ss. *rpo-ns* s (Fun g ts)) then b else e) = e **for** b e

:: bool \times bool **by** simp

note res = λ (5)[unfolded *rpo.simps* this prc Let-def split]

from res **have** rel: \forall t \in set ts. *rpo-s* (Fun f ss) t **by** (auto split: if-splits)

note IH = λ (2)[OF False prc[symmetric] refl]

from rel IH **show** ?thesis **by** auto

qed

qed

lemma *term-lists-of-cnf*: **assumes** *term-lists-of-cnf* *cnf* = (*as*, *bs*)
and *non-triv*: *cnf* ≠ []
shows (∃ β. *eval-cnf* β *cnf*)
 \longleftrightarrow (*mset as*, *mset bs*) ∈ *s-mul-ext* (*trivial-rpo.RPO-NS*) (*trivial-rpo.RPO-S*)
length (*vars-of-cnf* *cnf*) ≥ 2 \implies
(∃ β. *eval-cnf* β *cnf*) \longleftrightarrow (*Fun G as*, *Fun H bs*) ∈ *trivial-rpo.RPO-S*

proof –
obtain *xs ys S NS* **where** *mp*: *mset-problem-of-cnf* *cnf* = (*xs,ys,S,NS*)
by (*cases mset-problem-of-cnf* *cnf*, *auto*)
from *assms*(1)[*unfolded term-lists-of-cnf-def mp split*]
have *abs*: *as* = *map* (*ms-elem-to-term* *cnf*) *xs* *bs* = *map* (*ms-elem-to-term* *cnf*)
ys **by** *auto*
from *mp*[*unfolded mset-problem-of-cnf-def Let-def List.maps-def, simplified*]
have *S*: *S* = *concat* (*map* (λ*i*. *map* (λ*l*. (*ms-elem-of-lit* *l*, *Inr i*)) (*cnf* ! *i*))
[0..*length* *cnf*])
and *NS*: *NS* = *concat* (*map* (λ*x*. [(*Inl* (*x*, *Positive*), *Inl* (*x*, *Unsigned*)), (*Inl* (*x*,
Negative), *Inl* (*x*, *Unsigned*))]) (*vars-of-cnf* *cnf*)) @ *S*
and *ys*: *ys* = *map* (λ*x*. *Inl* (*x*, *Unsigned*)) (*vars-of-cnf* *cnf*) @ *map* *Inr* [0..*length*
cnf]
and *xs*: *xs* = *concat* (*map* (λ*x*. [*Inl* (*x*, *Positive*), *Inl* (*x*, *Negative*))]) (*vars-of-cnf*
cnf)) **by** *auto*
show *one*: (∃ β. *eval-cnf* β *cnf*)
 \longleftrightarrow (*mset as*, *mset bs*) ∈ *s-mul-ext* (*trivial-rpo.RPO-NS*) (*trivial-rpo.RPO-S*)
unfolding *mset-problem-of-cnf*(2)[*OF mp non-triv*]

proof
assume (*mset xs*, *mset ys*) ∈ *s-mul-ext* (*set NS*) (*set S*)
hence *mem*: (*xs*, *ys*) ∈ {(*as*, *bs*). (*mset as*, *mset bs*) ∈ *s-mul-ext* (*set NS*) (*set*
S)} **by** *simp*
have (*as*, *bs*) ∈ {(*as*, *bs*). (*mset as*, *mset bs*) ∈ *s-mul-ext* *trivial-rpo.RPO-NS*
trivial-rpo.RPO-S}
unfolding *abs*
proof (*rule s-mul-ext-map*[*OF - - mem, of ms-elem-to-term* *cnf*])
{
fix *a b*
assume (*a,b*) ∈ *set S*
from *this*[*unfolded S, simplified*]
obtain *i x s* **where** *i*: *i* < *length* *cnf* **and** *a*: *a* = *ms-elem-of-lit* (*x,s*)
and *mem*: (*x,s*) ∈ *set* (*cnf* ! *i*) **and** *b*: *b* = *Inr i* **by** *auto*
from *mem i* **obtain** *t ts* **where** *a*: *ms-elem-to-term* *cnf* *a* = *Fun F* (*Var*
(*Inl x*) # *t* # *ts*) **and** *len*: *length* *ts* = *length* *cnf* **and** *tsi*: *ts* ! *i* = *Var* (*Inr i*)
unfolding *a* **by** (*cases s, auto*)
from *len i tsi* **have** *mem*: *Var* (*Inr i*) ∈ *set* *ts* **unfolding** *set-conv-nth* **by**
auto
show (*ms-elem-to-term* *cnf* *a*, *ms-elem-to-term* *cnf* *b*) ∈ *trivial-rpo.RPO-S*
unfolding *a b* **by** (*simp add: Let-def, intro disjI2 beXI*[*OF - mem*], *simp*)
} **note** *S* = *this*
fix *a b*

```

assume mem: (a,b) ∈ set NS
show (ms-elem-to-term cnf a, ms-elem-to-term cnf b) ∈ trivial-rpo.RPO-NS
proof (cases (a,b) ∈ set S)
  case True
    from S[OF this] show ?thesis using trivial-rpo.RPO-S-subset-RPO-NS by
fastforce
  next
    case False
      with mem[unfolded NS] obtain x s where x ∈ set (vars-of-cnf cnf) and
        a: a = Inl (x, s) and b: b = Inl (x, Unsigned) and s: s = Positive ∨ s =
Negative
      by auto
      show ?thesis unfolding a b using s
      apply (auto intro!: all-nstri-imp-mul-nstri)
      subgoal for i by (cases i; cases i - 1, auto intro!: all-nstri-imp-mul-nstri)
      subgoal for i by (cases i; cases i - 1, auto intro!: all-nstri-imp-mul-nstri)
      done
    qed
  qed
thus (mset as, mset bs) ∈ s-mul-ext trivial-rpo.RPO-NS trivial-rpo.RPO-S
unfolding abs by simp
next
  assume (mset as, mset bs) ∈ s-mul-ext trivial-rpo.RPO-NS trivial-rpo.RPO-S
  hence mem: (as, bs) ∈ {(as, bs). (mset as, mset bs) ∈ s-mul-ext trivial-rpo.RPO-NS
trivial-rpo.RPO-S} by simp
  have xsys: xs = map term-to-ms-elem as ys = map term-to-ms-elem bs un-
folding abs map-map o-def
  by (rule nth-equalityI, auto)
  have (xs, ys) ∈ {(as, bs). (mset as, mset bs) ∈ s-mul-ext (set NS) (set S)}
  unfolding xsys
  proof (rule s-mul-ext-map[OF - - mem])
  fix a b
  assume ab: a ∈ set as b ∈ set bs
  from ab(2)[unfolded abs] obtain y where y: y ∈ set ys and b: b =
ms-elem-to-term cnf y by auto
  from ab(1)[unfolded abs] obtain x where x: x ∈ set xs and a: a =
ms-elem-to-term cnf x by auto
  from y[unfolded ys] obtain v i where y: y = Inl (v, Unsigned) ∧ v ∈ set
(vars-of-cnf cnf)
    ∨ y = Inr i ∧ i < length cnf by auto
  from x[unfolded xs] obtain w s where s: s = Positive ∨ s = Negative and
w: w ∈ set (vars-of-cnf cnf)
  and x: x = Inl (w, s) by auto
  {
    assume y: y = Inl (v, Unsigned) and v: v ∈ set (vars-of-cnf cnf)
    {
      assume (a,b) ∈ trivial-rpo.RPO-NS
      from s this v have (term-to-ms-elem a, term-to-ms-elem b) ∈ set NS
unfolding a b x y

```

```

    by (cases s, auto split: if-splits simp: Let-def NS)
  } note case11 = this
  {
    assume (a,b) ∈ trivial-rpo.RPO-S
    hence trivial-rpo.rpo-s a b by simp
    from s this v have False unfolding a b x y
    by (cases, auto split: if-splits simp: Let-def, auto dest!: fst-mul-ext-imp-fst)
  } note case12 = this
  note case11 case12
} note case1 = this
{
  assume y: y = Inr i and i: i < length cnf
  assume (a,b) ∈ trivial-rpo.RPO-NS ∪ trivial-rpo.RPO-S
  hence (a,b) ∈ trivial-rpo.RPO-NS
  using trivial-rpo.RPO-S-subset-RPO-NS by blast
  from s this have (term-to-ms-elem a, term-to-ms-elem b) ∈ set S unfolding
a b x y
  by (cases, auto split: if-splits simp: Let-def NS S, force+)
} note case2 = this
from case1 case2 y
show (a, b) ∈ trivial-rpo.RPO-S ⇒ (term-to-ms-elem a, term-to-ms-elem b)
∈ set S by auto
from case1 case2 y
show (a, b) ∈ trivial-rpo.RPO-NS ⇒ (term-to-ms-elem a, term-to-ms-elem
b) ∈ set NS unfolding NS by auto
qed
thus (mset xs, mset ys) ∈ s-mul-ext (set NS) (set S) by simp
qed

```

Here the encoding for single RPO-terms is handled. We do this here and not in a separate lemma, since some of the properties of xs, ys, as, bs, etc. are required.

```

assume len2: length (vars-of-cnf cnf) ≥ 2
show (∃ β. eval-cnf β cnf) ↔ (Fun G as, Fun H bs) ∈ trivial-rpo.RPO-S
  unfolding one
proof
  assume mul: (mset as, mset bs) ∈ s-mul-ext trivial-rpo.RPO-NS trivial-rpo.RPO-S
  {
    fix b
    assume b ∈ set bs
    hence b ∈# mset bs by auto
    from s-mul-ext-point[OF mul this] have ∃ a ∈ set as. (a,b) ∈ trivial-rpo.RPO-NS
      using trivial-rpo.RPO-S-subset-RPO-NS by fastforce
    hence (Fun G as, b) ∈ trivial-rpo.RPO-S by (cases b, auto)
  }
  with mul show (Fun G as, Fun H bs) ∈ trivial-rpo.RPO-S
    by (auto simp: mul-ext-def)
next

```

```

assume rpo: (Fun G as, Fun H bs) ∈ trivial-rpo.RPO-S
have ¬ (∃ s ∈ set as. trivial-rpo.rpo-ns s (Fun H bs))
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain a where a: a ∈ set as and trivial-rpo.rpo-ns a (Fun H bs) by
  auto
  with trivial-rpo.rpo-vars-term[of a Fun H bs]
  have vars: vars-term (Fun H bs) ⊆ vars-term a by auto
  from a[unfolded abs xs, simplified] obtain x where vars-term a ∩ range Inl
  = {Inl x}
  by force
  with vars have sub: vars-term (Fun G bs) ∩ range Inl ⊆ {Inl x} by auto
  from len2 obtain y z vs where vars: vars-of-cnf cnf = y # z # vs
  by (cases vars-of-cnf cnf; cases tl (vars-of-cnf cnf), auto)
  have distinct (vars-of-cnf cnf) unfolding vars-of-cnf-def by auto
  with vars have yz: y ≠ z by auto
  have {Inl y, Inl z} ⊆ vars-term (Fun G bs)
  unfolding abs ys vars by auto
  with sub yz
  show False by auto
qed
with rpo have fst (mul-ext trivial-rpo.rpo-pr as bs) by (auto split: if-splits)
thus (mset as, mset bs) ∈ s-mul-ext trivial-rpo.RPO-NS trivial-rpo.RPO-S
by (auto simp: mul-ext-def Let-def)
qed
qed

```

```

lemma rpo-constraint-of-cnf: assumes non-triv: length (vars-of-cnf cnf) ≥ 2
shows (∃ β. eval-cnf β cnf) ⟷ rpo-constraint-of-cnf cnf ∈ trivial-rpo.RPO-S
proof –
  obtain as bs where res: term-lists-of-cnf cnf = (as,bs) by force
  from non-triv have cnf: cnf ≠ [] unfolding vars-of-cnf-def by auto
  show ?thesis unfolding rpo-constraint-of-cnf-def res split
  by (subst term-lists-of-cnf(2)[OF res cnf non-triv], auto)
qed

```

6.3 Size of Encoding is Quadratic

```

fun term-size :: ('f,'v)term ⇒ nat where
  term-size (Var x) = 1
| term-size (Fun f ts) = 1 + sum-list (map term-size ts)

```

```

lemma size-of-rpo-constraint-of-cnf:
assumes rpo-constraint-of-cnf = (s,t)
and size-cnf cnf = n
shows term-size s + term-size t ≤ 4 * n2 + 12 * n + 2
proof –
obtain as bs S NS where mp: multiset-problem-of-cnf cnf = (as, bs, S, NS)
by (cases multiset-problem-of-cnf cnf, auto)

```

```

from size-of-multiset-problem-of-cnf[OF mp assms(2)]
have las: length as ≤ 2 * n length bs ≤ 2 * n by auto
have tl: term-lists-of-cnf cnf = (map (ms-elem-to-term cnf) as, map (ms-elem-to-term
cnf) bs)
  unfolding term-lists-of-cnf-def mp split by simp
  from assms(1)[unfolded rpo-constraint-of-cnf-def tl split]
  have st: s = Fun G (map (ms-elem-to-term cnf) as) t = Fun H (map (ms-elem-to-term
cnf) bs) by auto
  have [simp]: term-size (if b then Var (Inr x) :: (FSyms, 'a + nat) Term.term
else Fun A []) = 1 for b x
    by (cases b, auto)
  have len-n: length cnf ≤ n using assms(2) unfolding size-cnf-def by auto
  have term-size (ms-elem-to-term cnf a) ≤ 3 + length cnf for a
    by (cases (cnf,a) rule: ms-elem-to-term.cases, auto simp: o-def sum-list-triv)
  also have ... ≤ 3 + n using len-n by auto
  finally have size-ms: term-size (ms-elem-to-term cnf a) ≤ 3 + n for a .
  {
    fix u
    assume u ∈ {s,t}
    then obtain G cs where cs ∈ {as, bs} and u: u = Fun G (map (ms-elem-to-term
cnf) cs)
      unfolding st by auto
      hence lcs: length cs ≤ 2 * n using las by auto
      have term-size u = 1 + (∑ x←cs. term-size (ms-elem-to-term cnf x)) unfolding
u by (simp add: o-def size-list-conv-sum-list)
      also have ... ≤ 1 + (∑ x←cs. 3 + n)
        by (intro add-mono lcs le-refl sum-list-mono size-ms)
      also have ... ≤ 1 + (2 * n) * (3 + n) unfolding sum-list-triv
        by (intro add-mono le-refl mult-mono, insert lcs, auto)
      also have ... = 2 * n2 + 6 * n + 1 by (simp add: field-simps power2-eq-square)
      finally have term-size u ≤ 2 * n2 + 6 * n + 1 .
  }
from this[of s] this[of t]
show term-size s + term-size t ≤ 4 * n2 + 12 * n + 2 by simp
qed

```

6.4 Check Executability

```

value (code) case rpo-constraint-of-cnf [
  [("x", True), ("y", False)],      — clause 0
  [("x", False)],                  — clause 1
  [("y", True), ("z", True)],      — clause 2
  [("x", True), ("y", True), ("z", False)]] — clause 3
  of (s,t) ⇒ ("SAT: ", trivial-rpo.rpo-s s t, "Encoding: ", s, " >RPO ", t)

```

```

hide-const (open) A F G H U P N

```

```

end

```

References

- [1] M. Codish, J. Giesl, P. Schneider-Kamp, and R. Thiemann. SAT solving for termination proofs with recursive path orders and dependency pairs. *J. Autom. Reason.*, 49(1):53–93, 2012.
- [2] N. Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1/2):69–116, 1987.
- [3] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *J. Satisf. Boolean Model. Comput.*, 2(1-4):1–26, 2006.
- [4] R. Thiemann, G. Allais, and J. Nagele. On the formalization of termination techniques based on multiset orderings. In A. Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA '12), RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, volume 15 of *LIPICs*, pages 339–354. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.