The Generalized Multiset Ordering is NP-Complete

René Thiemann Lukas Schmidinger

March 24, 2023

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

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 $\bigwedge i.\ i < length\ ys1 \Longrightarrow (xs1\ !\ i,\ ys1\ !\ i) \in NS$
  and $xs2 \neq []$
  and $\bigwedge y.\ y \in set\ ys2 \Longrightarrow \exists\, a \in set\ xs2.\ (a,\ y) \in S$
shows $(xs,\ ys) \in s\text{-}mul\text{-}ext\ NS\ S$
  $\langle proof \rangle$

lemma *ns-mul-ext-intro*:
  assumes $xs = mset\ xs1 + mset\ xs2$
  and $ys = mset\ ys1 + mset\ ys2$
  and $length\ xs1 = length\ ys1$
  and $\bigwedge i.\ i < length\ ys1 \Longrightarrow (xs1\ !\ i,\ ys1\ !\ i) \in NS$
  and $\bigwedge y.\ y \in set\ ys2 \Longrightarrow \exists\, x \in set\ xs2.\ (x,\ y) \in S$
shows $(xs,\ ys) \in ns\text{-}mul\text{-}ext\ NS\ S$
  $\langle proof \rangle$

lemma *ns-mul-ext-elim*: **assumes** $(xs,\ ys) \in ns\text{-}mul\text{-}ext\ NS\ S$
  shows $\exists\ xs1\ xs2\ ys1\ ys2.$
    $xs = mset\ xs1 + mset\ xs2$
  $\wedge\ ys = mset\ ys1 + mset\ ys2$
  $\wedge\ length\ xs1 = length\ ys1$
  $\wedge\ (\forall\ i.\ i < length\ ys1 \longrightarrow (xs1\ !\ i,\ ys1\ !\ i) \in NS)$
  $\wedge\ (\forall\ y \in set\ ys2.\ \exists\, x \in set\ xs2.\ (x,\ y) \in S)$
$\langle proof \rangle$

lemma *s-mul-ext-elim*: **assumes** $(xs,\ ys) \in s\text{-}mul\text{-}ext\ NS\ S$
  shows $\exists\ xs1\ xs2\ ys1\ ys2.$
    $xs = mset\ xs1 + mset\ xs2$
  $\wedge\ ys = mset\ ys1 + mset\ ys2$
  $\wedge\ length\ xs1 = length\ ys1$
  $\wedge\ xs2 \neq []$
  $\wedge\ (\forall\ i.\ i < length\ ys1 \longrightarrow (xs1\ !\ i,\ ys1\ !\ i) \in NS)$
  $\wedge\ (\forall\ y \in set\ ys2.\ \exists\, x \in set\ xs2.\ (x,\ y) \in S)$
$\langle proof \rangle$

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-mul-ext* $(NS \cup S')$ S = *ns-mul-ext* NS S
⟨*proof*⟩

lemma *ns-mul-ext-NS-union-S*: *ns-mul-ext* $(NS \cup S)$ S = *ns-mul-ext* NS S
⟨*proof*⟩

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)) = *mset* (*map* v t)
⟨*proof*⟩

lemma *mset-map-split*: **assumes** *mset* (*map* f *xs*) = *mset ys1* + *mset ys2*
shows $\exists$ *zs1 zs2*. *mset xs* = *mset zs1* + *mset zs2* $\wedge$ *ys1* = *map f zs1* $\wedge$ *ys2* = *map f zs2*
⟨*proof*⟩

lemma *deciding-mult*:
assumes *tr*: *trans* S **and** *ir*: *irrefl* S
shows $(N,M) \in$ *mult* S = ($M \neq N$ $\wedge$ ($\forall$ b $\in\#$ $N - M$. $\exists$ a $\in\#$ $M - N$. $(b,a) \in S$))
⟨*proof*⟩

lemma *s-mul-ext-map*: ($\bigwedge a$ b. $a \in$ *set as* $\Longrightarrow$ $b \in$ *set bs* $\Longrightarrow$ $(a, b) \in S$ $\Longrightarrow$ $(f\ a, f\ b) \in S'$) $\Longrightarrow$
($\bigwedge a$ b. $a \in$ *set as* $\Longrightarrow$ $b \in$ *set bs* $\Longrightarrow$ $(a, b) \in NS$ $\Longrightarrow$ $(f\ a, f\ b) \in NS'$) $\Longrightarrow$
(*as*, *bs*) $\in$ {(*as*, *bs*). (*mset as*, *mset bs*) $\in$ *s-mul-ext NS S*} $\Longrightarrow$
(*map f as*, *map f bs*) $\in$ {(*as*, *bs*). (*mset as*, *mset bs*) $\in$ *s-mul-ext* NS' S'}
⟨*proof*⟩

lemma *fst-mul-ext-imp-fst*: **assumes** *fst* (*mul-ext f xs ys*)
and *length xs* $\leq$ *length ys*
shows $\exists$ x y. $x \in$ *set xs* $\wedge$ $y \in$ *set ys* $\wedge$ *fst* (f x y)
⟨*proof*⟩

lemma *ns-mul-ext-point*: **assumes** (*as*,*bs*) $\in$ *ns-mul-ext NS S*
and $b \in\#$ *bs*
shows $\exists$ $a \in\#$ *as*. $(a,b) \in NS \cup S$
⟨*proof*⟩

lemma *s-mul-ext-point*: **assumes** (*as*,*bs*) $\in$ *s-mul-ext NS S*
and $b \in\#$ *bs*
shows $\exists$ $a \in\#$ *as*. $(a,b) \in NS \cup S$
⟨*proof*⟩

end

3 Propositional Formulas and CNFs

We provide a straight-forward definition of propositional formulas, defined as arbitray 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* ⇒ *bool*) ⇒ *'a formula* ⇒ *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* :: $('a \Rightarrow bool) \Rightarrow 'a\ cnf \Rightarrow bool$ **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*))
⟨*proof*⟩

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* :: $'a\ cnf \Rightarrow nat$ **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*]:

$eval\ \alpha\ (enc\text{-}False) = False$
$eval\ \alpha\ (enc\text{-}True) = True$
$eval\ \alpha\ (enc\text{-}pos\ x) = \alpha\ x$
$eval\ \alpha\ (enc\text{-}neg\ x) = (\neg\ \alpha\ x)$
$eval\ \alpha\ (enc\text{-}different\ x\ y) = (\alpha\ x \neq \alpha\ y)$
$eval\ \alpha\ (enc\text{-}equiv\text{-}and\text{-}not\ x\ y\ z) = (\alpha\ x \longleftrightarrow \alpha\ y \wedge \neg\ \alpha\ z)$
$eval\ \alpha\ (enc\text{-}equiv\text{-}ite\ x\ y\ z\ u) = (\alpha\ x \longleftrightarrow (if\ \alpha\ y\ then\ \alpha\ z\ else\ \alpha\ u))$
$eval\ \alpha\ (enc\text{-}ite\ x\ y\ z) = (if\ \alpha\ x\ then\ \alpha\ y\ else\ \alpha\ z)$
$eval\ \alpha\ (enc\text{-}impl\ x\ f) = (\alpha\ x \longrightarrow eval\ \alpha\ f)$
$eval\ \alpha\ (enc\text{-}var\text{-}impl\ x\ y) = (\alpha\ x \longrightarrow \alpha\ y)$
$eval\ \alpha\ (enc\text{-}not\text{-}and\ x\ y) = (\neg\ (\alpha\ x \wedge \alpha\ y))$
$eval\ \alpha\ (enc\text{-}not\text{-}all\ xs) = (\neg\ (Ball\ (set\ xs)\ \alpha))$
$eval\ \alpha\ (enc\text{-}conj\ fs) = (Ball\ (set\ fs)\ (eval\ \alpha))$

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* :: $('a \times 'a \times 'a)$ *list* $\Rightarrow$ $'f$ *list* $\times\ 'a \times 'a$ **where**
encode-sum-0-1-main $[(x,\ zero,\ one)] = ([enc\text{-}different\ zero\ x],\ zero,\ x)$
| *encode-sum-0-1-main* $((x,\ zero,\ one)\ \#\ rest) = ($*case encode-sum-0-1-main rest of*
$(conds,\ fzero,\ fone) \Rightarrow$ *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* :: $('a \times 'a \times 'a)$ *list* $\Rightarrow$ $'f \times 'f$ *list* **where**
encode-exactly-one vars $=$ (*case vars of* $[] \Rightarrow (enc\text{-}False,\ [])$
| $[(x,\text{-},\text{-})] \Rightarrow (enc\text{-}pos\ x,\ [])$
| $((x,\text{-},\text{-})\ \#\ vars) \Rightarrow$ (*case encode-sum-0-1-main vars of* $(conds,\ zero,\ one)$
$\Rightarrow (enc\text{-}ite\ x\ zero\ one,\ conds)))$

fun *encodeGammaCond* :: $'a \Rightarrow 'a \Rightarrow bool \Rightarrow bool \Rightarrow 'f$ **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*, *cands*) ⇒ *one* # *cands*;
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* (λ *i*. *case inner-right i of* (*one*, *cands*) ⇒ *enc-impl* (*Epsilon i*) *one* # *cands*) [*0* ..< *n*]
in left @ *right*)

definition *formula15* :: (*nat* ⇒ *nat* ⇒ *bool*) ⇒ (*nat* ⇒ *nat* ⇒ *bool*) ⇒ *nat* ⇒ *nat* ⇒ *'f list* **where**
formula15 cs cns n m = (*let*
conjs = *List.maps* (λ *i*. *List.maps* (λ *j*. *let s* = *cs i j*; *ns* = *cns i j in*
if s ∧ *ns then* [] *else* [*encodeGammaCond* (*Gamma i j*) (*Epsilon i*) *s ns*]) [*0* ..< *m*]) [*0* ..< *n*]
in conjs @ *formula14 n m*)

definition *formula16* :: (*nat* ⇒ *nat* ⇒ *bool*) ⇒ (*nat* ⇒ *nat* ⇒ *bool*) ⇒ *nat* ⇒ *nat* ⇒ *'f list* **where**
formula16 cs cns n m = (*enc-not-all* (*map Epsilon* [*0* ..< *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* ⇒ *'a* ⇒ *bool* × *bool*) ⇒ *'a list* ⇒ *'a list* ⇒ *'f* × *'f* **where**
encode-mul-ext s-ns xs ys = (*let*
n = *length xs*;
m = *length ys*;
cs = (λ *i j*. *fst* (*s-ns* (*xs* ! *i*) (*ys* ! *j*)));
cns = (λ *i j*. *snd* (*s-ns* (*xs* ! *i*) (*ys* ! *j*)));
f15 = *formula15 cs cns n m*;
f16 = *enc-not-all* (*map Epsilon* [*0* ..< *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* **where**
eval-all α *fs* $\equiv$ (*Ball* (*set fs*) (*eval* α))

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*) $\Longrightarrow$
(α *ze* $\longleftrightarrow$ $\neg$ ($\exists$ *y* $\in$ *insert x* (*fst* ' *set re*). α *y*))
$\wedge$ (α *on* $\longleftrightarrow$ ($\exists!$ *y* $\in$ *insert x* (*fst* ' *set re*). α *y*))
and $\neg$ *prop* $\Longrightarrow$ *eval-all* α *conds*
and *distinct* (*map fst vars*)
and *vars* $\neq$ []
shows *eval-all* α *conds*
$\wedge$ (α *zero* $\longleftrightarrow$ $\neg$ ($\exists$ *x* $\in$ *fst* ' *set vars*. α *x*))
$\wedge$ (α *one* $\longleftrightarrow$ ($\exists!$ *x* $\in$ *fst* ' *set vars*. α *x*))
$\langle$*proof*$\rangle$

lemma *encode-exactly-one-complete*: **assumes** *encode-exactly-one vars* = (*one*, *conds*)
and $\bigwedge$ *i x ze on*. *i* < *length vars* $\Longrightarrow$
vars ! *i* = (*x*,*ze*,*on*) $\Longrightarrow$
(α *ze* $\longleftrightarrow$ $\neg$ ($\exists$ *y* $\in$ *fst* ' *set* (*drop i vars*). α *y*))
$\wedge$ (α *on* $\longleftrightarrow$ ($\exists!$ *y* $\in$ *fst* ' *set* (*drop i vars*). α *y*))
and *distinct* (*map fst vars*)
shows *eval-all* α *conds* $\wedge$ (*eval* α *one* $\longleftrightarrow$ ($\exists!$ *x* $\in$ *fst* ' *set vars*. α *x*))
$\langle$*proof*$\rangle$

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* $\in$ *fst* ' *set vars*. α *x*
$\langle$*proof*$\rangle$

lemma *encodeGammaCond*[*simp*]: *eval* α (*encodeGammaCond gam eps s ns*) =
(α *gam* $\longrightarrow$ (α *eps* $\longrightarrow$ *ns*) $\wedge$ ($\neg$ α *eps* $\longrightarrow$ *s*))
$\langle$*proof*$\rangle$

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

$\langle$*proof*$\rangle$

lemma *eval-all-Cons*[*simp*]: *eval-all* α (*f* # *gs*) = (*eval* α *f* $\wedge$ *eval-all* α *gs*)
⟨*proof*⟩

lemma *eval-all-concat*[*simp*]: *eval-all* α (*concat fs*) = ($\forall$ *f* $\in$ *set fs*. *eval-all* α *f*)
⟨*proof*⟩

lemma *eval-all-maps*[*simp*]: *eval-all* α (*List.maps f fs*) = ($\forall$ *g* $\in$ *set fs*. *eval-all* α (*f g*))
⟨*proof*⟩
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*) $\Longrightarrow$ (*mset* (*map s* [*0* ..< *n*]), *mset* (*map t* [*0* ..< *m*])) $\in$ *s-mul-ext NS S*
⟨*proof*⟩

lemma *bex1-cong*: $X = Y \Longrightarrow (\bigwedge x.\ x \in Y \Longrightarrow P\ x = Q\ x) \Longrightarrow (\exists!x.\ x \in X \wedge P\ x) = (\exists!x.\ x \in Y \wedge Q\ x)$
⟨*proof*⟩

lemma *encoding-complete*:
assumes (*mset* (*map s* [*0* ..< *n*]), *mset* (*map t* [*0* ..< *m*])) $\in$ *ns-mul-ext NS S*
shows ($\exists$ *v*. *eval-all v* (*formula15 cs cns n m*) $\wedge$
((*mset* (*map s* [*0* ..< *n*]), *mset* (*map t* [*0* ..< *m*])) $\in$ *s-mul-ext NS S* $\longrightarrow$ *eval-all v* (*formula16 cs cns n m*)))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩
end

lemma *encode-mul-ext*: **assumes** *encode-mul-ext f xs ys* = (φ_S, φ_{NS})
shows *mul-ext f xs ys* = (($\exists$ *v*. *eval v* φ_S), ($\exists$ *v*. *eval v* φ_{NS}))
⟨*proof*⟩
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*
⟨*proof*⟩

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* $\neq$ []
shows *sum-list* (*map size-pf conds*) = *16* $*$ *length vars* $-$ *12*
⟨*proof*⟩

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*⟩

lemma *sum-list-concat*: *sum-list* (*concat xs*) = *sum-list* (*map sum-list xs*)
⟨*proof*⟩

lemma *sum-list-triv-cong*: **assumes** *length xs* = *n*
and $\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = c$
shows *sum-list* (*map f xs*) = *n* $*$ *c*
⟨*proof*⟩

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*⟩

lemma *size-pf-encodeGammaCond*: *size-pf* (*pf-encodeGammaCond gam eps ns s*) ≤ *4*
⟨*proof*⟩

lemma *size-pf-formula15*: *sum-list* (*map size-pf* (*pf-formula15 cs cns n m*)) ≤ *m* + *3* $*$ *n* + *m* $*$ (*n* $*$ *16* − *21*) + *n* $*$ (*m* $*$ *16* − *21*) + *4* $*$ *m* $*$ *n*
⟨*proof*⟩

lemma *size-pf-formula16*: *sum-list* (*map size-pf* (*pf-formula16 cs cns n m*)) ≤ *2* + *m* + *4* $*$ *n* + *m* $*$ (*n* $*$ *16* − *21*) + *n* $*$ (*m* $*$ *16* − *21*) + *4* $*$ *m* $*$ *n*
⟨*proof*⟩

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*: $n \neq 0$
shows *size-pf* $\varphi_S \leq 36 * n^2$
size-pf $\varphi_{NS} \leq 36 * n^2$
⟨*proof*⟩

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*
λ *x y z*. [[(*x*,*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*
⟨*proof*⟩

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* $\neq$ []
shows *sum-list* (*map size-cnf conds*) = *26* $*$ *length vars* − *20*
⟨*proof*⟩

lemma *size-cnf-encode-exactly-one*: **assumes** *cnf-encode-exactly-one vars* = (*one*, *conds*)
shows *size-cnf one* + *sum-list* (*map size-cnf conds*) $\leq$ *2* + (*26* $*$ *length vars* − *42*) $\wedge$ *length one* $\leq$ *2*
⟨*proof*⟩

lemma *sum-list-mono-const*: **assumes** $\bigwedge$ *x*. *x* $\in$ *set xs* $\Longrightarrow$ *f x* $\leq$ *c*
and *n* = *length xs*
shows *sum-list* (*map f xs*) $\leq$ *n* $*$ *c*
⟨*proof*⟩

lemma *size-cnf-formula14*: *sum-list* (*map size-cnf* (*cnf-formula14 n m*)) $\leq$ *2* $*$ *m* + *4* $*$ *n* + *m* $*$ (*26* $*$ *n* − *42*) + *n* $*$ (*26* $*$ *m* − *42*)
⟨*proof*⟩

lemma *size-cnf-encodeGammaCond*: *size-cnf* (*cnf-encodeGammaCond gam eps ns s*) $\leq$ *3*
⟨*proof*⟩

lemma *size-cnf-formula15*: *sum-list* (*map size-cnf* (*cnf-formula15 cs cns n m*)) $\leq$ *2* $*$ *m* + *4* $*$ *n* + *m* $*$ (*26* $*$ *n* − *42*) + *n* $*$ (*26* $*$ *m* − *42*) + *3* $*$ *n* $*$ *m*
⟨*proof*⟩

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*⟩

lemma *size-cnf-concat*: *size-cnf* (*concat xs*) = *sum-list* (*map size-cnf xs*) ⟨*proof*⟩

lemma *size-cnf-encode-mul-ext*: **assumes** *cnf-encode-mul-ext f xs ys* = (φ_S, φ_{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*⟩

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* (λ *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* (λ *i j*. (*True*, *False*)) [*0..<n*] [*0..<n*])), $51 * n^2$)

Example encoding

value (*code*) *fst* (*pf-encode-mul-ext* (λ *i j*. ($i > j$, $i \geq j$)) [*0..<3*] [*0..<5*])
value (*code*) *fst* (*cnf-encode-mul-ext* (λ *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*)
⟷ ((*mset left*, *mset right*) ∈ *ns-mul-ext* (*set NSS*) (*set S*))
cnf ≠ [] ⟹ (∃ β. *eval-cnf* β *cnf*)
⟷ ((*mset left*, *mset right*) ∈ *s-mul-ext* (*set NSS*) (*set S*))
⟨*proof*⟩

lemma *multiset-problem-of-cnf-mul-ext*:

assumes *multiset-problem-of-cnf cnf* = (*xs*, *ys*, *S*, *NS*)
and *non-trivial*: *cnf* $\neq$ []
shows ($\exists$ β. *eval-cnf* β *cnf*)
$\longleftrightarrow$ *mul-ext* (λ *a b*. ((*a*,*b*) $\in$ *set S*, (*a*,*b*) $\in$ *set NS*)) *xs ys* = (*True*,*True*)
⟨*proof*⟩

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* $\leq$ *2* $*$ *s length ys* $\leq$ *2* $*$ *s length S* $\leq$ *s length NS* $\leq$ *3* $*$ *s*
⟨*proof*⟩

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*) $\Rightarrow$ (''*SAT*: '', *mul-ext* (λ *x y*. ((*x*,*y*) $\in$ *set S*, (*x*,*y*) $\in$ *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* $\Rightarrow$ '*a ms-elem* $\Rightarrow$ (*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-cnf* :: *'a cnf* ⇒ (*FSyms*, *'a* + *nat*)*term list* × (*FSyms*, *'a* + *nat*)*term list* **where**
  term-lists-of-cnf cnf = (*case multiset-problem-of-cnf cnf 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* ⇒ (-,-)*term* × (-,-)*term* **where**
  rpo-constraint-of-cnf cnf = (*case term-lists-of-cnf cnf of*
    (*as*, *bs*) ⇒ (*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*
  ⟨*proof*⟩

6.2 Soundness of the Encoding

fun *term-to-ms-elem* :: (*FSyms*, *'a* + *nat*)*term* ⇒ *'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*
  ⟨*proof*⟩

lemma (**in** *rpo-with-assms*) *rpo-vars-term*: *rpo-s s t* ∨ *rpo-ns s t* ⟹ *vars-term s* ⊇ *vars-term t*
⟨*proof*⟩

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*) $\in$ *s-mul-ext* (*trivial-rpo.RPO-NS*) (*trivial-rpo.RPO-S*)
length (*vars-of-cnf cnf*) $\geq$ *2* $\Longrightarrow$
($\exists$ β. *eval-cnf* β *cnf*) $\longleftrightarrow$ (*Fun G as*, *Fun H bs*) $\in$ *trivial-rpo.RPO-S*
$\langle$*proof*$\rangle$

lemma *rpo-constraint-of-cnf*: **assumes** *non-triv*: *length* (*vars-of-cnf cnf*) $\geq$ *2*
shows ($\exists$ β. *eval-cnf* β *cnf*) $\longleftrightarrow$ *rpo-constraint-of-cnf cnf* $\in$ *trivial-rpo.RPO-S*
$\langle$*proof*$\rangle$

6.3 Size of Encoding is Quadratic

fun *term-size* :: ('*f*,'*v*)*term* $\Rightarrow$ *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 cnf* = (*s*,*t*)
and *size-cnf cnf* = *n*
shows *term-size s* + *term-size t* $\leq$ $4 * n^2 + 12 * n + 2$
$\langle$*proof*$\rangle$

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*) $\Rightarrow$ (''*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.