Difference Bound Matrices

Simon Wimmer, Peter Lammich

August 26, 2024

Abstract

Difference Bound Matrices (DBMs) [2] are a data structure used to represent a type of convex polytopes, often called zones. DBMs find application such as in timed automata model checking and static program analysis. This entry formalizes DBMs and operations for inclusion checking, intersection, variable reset, upper-bound relaxation, and extrapolation (as used in timed automata model checking). With the help of the Imperative Refinement Framework, efficient imperative implementations of these operations are also provided. For each zone, there exists a canonical DBM. The characteristic properties of canonical forms are proved, including the fact that DBMs can be transformed in canonical form by the Floyd-Warshall algorithm. This entry is part of the work described in a paper by the authors of this entry [4] and a PhD thesis [3].

Contents

theory *DBM*
imports
Floyd-Warshall.Floyd-Warshall
HOL.Real
begin

type-synonym $('c, 't)$ *cval* $= 'c \Rightarrow 't$

1 Difference Bound Matrices

1.1 Definitions

1.1.1 Definition and Semantics of DBMs

Difference Bound Matrices (DBMs) constrain differences of clocks (or more precisely, the difference of values assigned to individual clocks by a valuation). The possible constraints are given by the following datatype:

datatype $'t$ *DBMEntry* $=$ *Le* $'t$ | *Lt* $'t$ | *INF* (∞)

This yields a simple definition of DBMs:

type-synonym $'t$ *DBM* $=$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ $'t$ *DBMEntry*

To relate clocks with rows and columns of a DBM, we use a clock numbering v of type $'c \Rightarrow$ *nat* to map clocks to indices. DBMs will regularly be accompanied by a natural number n, which designates the number of clocks constrained by the matrix. To be able to represent the full set of clock constraints with DBMs, we add an imaginary clock **0**, which shall be assigned to 0 in every valuation. In the following predicate we explicitly keep track of **0**.

class *time* $=$ *linordered-ab-group-add* $+$
assumes *dense*: $x < y \Longrightarrow \exists z.\ x < z \wedge z < y$
assumes *non-trivial*: $\exists\ x.\ x \neq 0$

begin

lemma *non-trivial-neg*: $\exists\ x.\ x < 0$
⟨*proof*⟩

end

instantiation *real* :: *time*
begin
instance ⟨*proof*⟩
end

inductive *dbm-entry-val* :: ('*c*, '*t*) *cval* ⇒ '*c option* ⇒ '*c option* ⇒ ('*t*::*time*) *DBMEntry* ⇒ *bool*
where
u r ≤ *d* ⟹ *dbm-entry-val u* (*Some r*) *None* (*Le d*) |
−*u c* ≤ *d* ⟹ *dbm-entry-val u None* (*Some c*) (*Le d*) |
u r < *d* ⟹ *dbm-entry-val u* (*Some r*) *None* (*Lt d*) |
−*u c* < *d* ⟹ *dbm-entry-val u None* (*Some c*) (*Lt d*) |
u r − *u c* ≤ *d* ⟹ *dbm-entry-val u* (*Some r*) (*Some c*) (*Le d*) |
u r − *u c* < *d* ⟹ *dbm-entry-val u* (*Some r*) (*Some c*) (*Lt d*) |
dbm-entry-val - - - ∞

declare *dbm-entry-val.intros*[*intro*]
inductive-cases[*elim*!]: *dbm-entry-val u None* (*Some c*) (*Le d*)
inductive-cases[*elim*!]: *dbm-entry-val u* (*Some c*) *None* (*Le d*)
inductive-cases[*elim*!]: *dbm-entry-val u None* (*Some c*) (*Lt d*)
inductive-cases[*elim*!]: *dbm-entry-val u* (*Some c*) *None* (*Lt d*)
inductive-cases[*elim*!]: *dbm-entry-val u* (*Some r*) (*Some c*) (*Le d*)
inductive-cases[*elim*!]: *dbm-entry-val u* (*Some r*) (*Some c*) (*Lt d*)

fun *dbm-entry-bound* :: ('*t*::*time*) *DBMEntry* ⇒ '*t*
where
dbm-entry-bound (*Le t*) = *t* |
dbm-entry-bound (*Lt t*) = *t* |
dbm-entry-bound ∞ = *0*

inductive *dbm-lt* :: ('*t*::*linorder*) *DBMEntry* ⇒ '*t DBMEntry* ⇒ *bool*
(- ≺ - [*51*, *51*] *50*)
where
dbm-lt (*Lt* -) ∞ |
dbm-lt (*Le* -) ∞ |
a < *b* ⟹ *dbm-lt* (*Le a*) (*Le b*) |
a < *b* ⟹ *dbm-lt* (*Le a*) (*Lt b*) |
a ≤ *b* ⟹ *dbm-lt* (*Lt a*) (*Le b*) |
a < *b* ⟹ *dbm-lt* (*Lt a*) (*Lt b*)

declare *dbm-lt.intros*[*intro*]

definition *dbm-le* :: ('*t*::*linorder*) *DBMEntry* ⇒ '*t DBMEntry* ⇒ *bool*
(- ≼ - [*51*, *51*] *50*)
where
dbm-le a b ≡ (*a* ≺ *b*) ∨ *a* = *b*

Now a valuation is contained in the zone represented by a DBM if it fulfills all individual constraints:

definition *DBM-val-bounded* :: $('c \Rightarrow nat) \Rightarrow ('c, 't)\ cval \Rightarrow ('t::time)\ DBM \Rightarrow nat \Rightarrow bool$
where
$DBM\text{-}val\text{-}bounded\ v\ u\ m\ n \equiv Le\ 0 \preceq m\ 0\ 0 \wedge$
$(\forall\ c.\ v\ c \leq n \longrightarrow (dbm\text{-}entry\text{-}val\ u\ None\ (Some\ c)\ (m\ 0\ (v\ c))$
$\wedge\ dbm\text{-}entry\text{-}val\ u\ (Some\ c)\ None\ (m\ (v\ c)\ 0)))$
$\wedge (\forall\ c1\ c2.\ v\ c1 \leq n \wedge v\ c2 \leq n \longrightarrow dbm\text{-}entry\text{-}val\ u\ (Some\ c1)\ (Some\ c2)\ (m\ (v\ c1)\ (v\ c2)))$

abbreviation *DBM-val-bounded-abbrev* ::
$('c, 't)\ cval \Rightarrow ('c \Rightarrow nat) \Rightarrow nat \Rightarrow ('t::time)\ DBM \Rightarrow bool$
$(\text{-} \vdash_{\text{-},\text{-}} \text{-}\ [48, 48, 48, 48]\ 48)$
where
$u \vdash_{v,n} M \equiv DBM\text{-}val\text{-}bounded\ v\ u\ M\ n$

1.1.2 Ordering DBM Entries

abbreviation
$dmin\ a\ b \equiv if\ a \prec b\ then\ a\ else\ b$

lemma *dbm-le-dbm-min*:
$a \preceq b \Longrightarrow a = dmin\ a\ b\ \langle proof \rangle$

lemma *dbm-lt-asym*:
assumes $e \prec f$
shows $^\sim f \prec e$
$\langle proof \rangle$

lemma *dbm-le-dbm-min2*:
$a \preceq b \Longrightarrow a = dmin\ b\ a$
$\langle proof \rangle$

lemma *dmb-le-dbm-entry-bound-inf*:
$a \preceq b \Longrightarrow a = \infty \Longrightarrow b = \infty$
$\langle proof \rangle$

lemma *dbm-not-lt-eq*: $\neg\ a \prec b \Longrightarrow \neg\ b \prec a \Longrightarrow a = b$
$\langle proof \rangle$

lemma *dbm-not-lt-impl*: $\neg\ a \prec b \Longrightarrow b \prec a \vee a = b\ \langle proof \rangle$

lemma $dmin\ a\ b = dmin\ b\ a$

⟨*proof*⟩

lemma *dbm-lt-trans*: $a \prec b \Longrightarrow b \prec c \Longrightarrow a \prec c$
⟨*proof*⟩

lemma *aux-3*: $\neg\ a \prec b \Longrightarrow \neg\ b \prec c \Longrightarrow a \prec c \Longrightarrow c = a$
⟨*proof*⟩

inductive-cases[*elim*!]: $\infty \prec x$

lemma *dbm-lt-asymmetric*[*simp*]: $x \prec y \Longrightarrow y \prec x \Longrightarrow False$
⟨*proof*⟩

lemma *le-dbm-le*: $Le\ a \preceq Le\ b \Longrightarrow a \leq b$ ⟨*proof*⟩

lemma *le-dbm-lt*: $Le\ a \preceq Lt\ b \Longrightarrow a < b$ ⟨*proof*⟩

lemma *lt-dbm-le*: $Lt\ a \preceq Le\ b \Longrightarrow a \leq b$ ⟨*proof*⟩

lemma *lt-dbm-lt*: $Lt\ a \preceq Lt\ b \Longrightarrow a \leq b$ ⟨*proof*⟩

lemma *not-dbm-le-le-impl*: $\neg\ Le\ a \prec Le\ b \Longrightarrow a \geq b$ ⟨*proof*⟩

lemma *not-dbm-lt-le-impl*: $\neg\ Lt\ a \prec Le\ b \Longrightarrow a > b$ ⟨*proof*⟩

lemma *not-dbm-lt-lt-impl*: $\neg\ Lt\ a \prec Lt\ b \Longrightarrow a \geq b$ ⟨*proof*⟩

lemma *not-dbm-le-lt-impl*: $\neg\ Le\ a \prec Lt\ b \Longrightarrow a \geq b$ ⟨*proof*⟩

1.1.3 Addition on DBM Entries

fun *dbm-add* :: ('*t*::*linordered-cancel-ab-semigroup-add*) *DBMEntry* ⇒ '*t* *DBMEntry* ⇒ '*t* *DBMEntry* (**infixl** ⊗ *70*)
where
  dbm-add ∞ - = ∞ |
  dbm-add - ∞ = ∞ |
  dbm-add (*Le a*) (*Le b*) = (*Le* (*a*+*b*)) |
  dbm-add (*Le a*) (*Lt b*) = (*Lt* (*a*+*b*)) |
  dbm-add (*Lt a*) (*Le b*) = (*Lt* (*a*+*b*)) |
  dbm-add (*Lt a*) (*Lt b*) = (*Lt* (*a*+*b*))

lemma *aux-4*: $x \prec y \Longrightarrow \neg$ *dbm-add x z* $\prec$ *dbm-add y z* $\Longrightarrow$ *dbm-add x z* = *dbm-add y z*
⟨*proof*⟩

lemma *aux-5*: $\neg\ x \prec y \Longrightarrow$ *dbm-add x z* $\prec$ *dbm-add y z* $\Longrightarrow$ *dbm-add y z* = *dbm-add x z*
⟨*proof*⟩

lemma *aux-42*: $x \prec y \Longrightarrow \neg$ *dbm-add z x* $\prec$ *dbm-add z y* $\Longrightarrow$ *dbm-add z x* = *dbm-add z y*
⟨*proof*⟩

lemma *aux-52*: $\neg\ x \prec y \Longrightarrow$ *dbm-add z x* $\prec$ *dbm-add z y* $\Longrightarrow$ *dbm-add z y* = *dbm-add z x*
⟨*proof*⟩

lemma *dbm-add-not-inf*:
$a \neq \infty \Longrightarrow b \neq \infty \Longrightarrow$ *dbm-add a b* $\neq \infty$
⟨*proof*⟩

lemma *dbm-le-not-inf*:
$a \preceq b \Longrightarrow b \neq \infty \Longrightarrow a \neq \infty$
⟨*proof*⟩

1.1.4 Negation of DBM Entries

fun *neg-dbm-entry* **where**
neg-dbm-entry (*Le a*) = *Lt* $(-a)$ |
neg-dbm-entry (*Lt a*) = *Le* $(-a)$ |
neg-dbm-entry $\infty = \infty$
— This case does not make sense but we make this definition for technical convenience.

lemma *neg-entry*:
$\{u.\ \neg$ *dbm-entry-val u a b e*$\} = \{u.$ *dbm-entry-val u b a* (*neg-dbm-entry e*)$\}$
if $e \neq (\infty ::$ - *DBMEntry*$)$ $a \neq$ *None* $\vee$ $b \neq$ *None*
⟨*proof*⟩

instantiation *DBMEntry* :: (*uminus*) *uminus*
begin
definition *uminus*: *uminus* = *neg-dbm-entry*
instance ⟨*proof*⟩
end

Note that it is not clear that this is the only sensible definition for negation of DBM entries. The following would also have been quite viable: *fun*

neg-dbm-entry where neg-dbm-entry (Le a) = Le (−a) | neg-dbm-entry (Lt a) = Lt (−a) | neg-dbm-entry ∞ = ∞
For most practical proofs using arithmetic on DBM entries we have found that this does not make much of a difference. Lemma ⟦*?e ≠ ∞*; *?a ≠ None ∨ ?b ≠ None*⟧ ⟹ {*u*. ¬ *dbm-entry-val u ?a ?b ?e*} = {*u*. *dbm-entry-val u ?b ?a* (*neg-dbm-entry ?e*)} would not hold any longer, however.

1.2 DBM Entries Form a Linearly Ordered Abelian Monoid

instantiation *DBMEntry* :: (*linorder*) *linorder*
begin
  definition *less-eq*: (≤) ≡ *dbm-le*
  definition *less*: (<) = *dbm-lt*
  instance
  ⟨*proof*⟩
end

class *linordered-cancel-ab-monoid-add* =
  linordered-cancel-ab-semigroup-add + *zero* +
    assumes *neutl*[*simp*]: *0* + *x* = *x*
    assumes *neutr*[*simp*]: *x* + *0* = *x*
begin

  subclass *linordered-ab-monoid-add*
    ⟨*proof*⟩

end

instantiation *DBMEntry* :: (*zero*) *zero*
begin
  definition *neutral*: *0* = *Le 0*
  instance ⟨*proof*⟩
end

instantiation *DBMEntry* :: (*linordered-cancel-ab-monoid-add*) *linordered-ab-monoid-add*
begin

  definition *add*: (+) = *dbm-add*

  instance ⟨*proof*⟩

end

interpretation *linordered-monoid*:
linordered-ab-monoid-add dbm-add Le (*0*::*'t*::*linordered-cancel-ab-monoid-add*) *dbm-le dbm-lt*
⟨*proof*⟩

instance *time* ⊆ *linordered-cancel-ab-monoid-add* ⟨*proof*⟩

lemma *dbm-add-strict-right-mono-neutral*: $a < Le\ (d :: 't :: time) \Longrightarrow a + Le\ (-d) < Le\ 0$
⟨*proof*⟩

lemma *dbm-lt-not-inf-less*[*intro*]: $A \neq \infty \Longrightarrow A \prec \infty$ ⟨*proof*⟩

lemma *add-inf*[*simp*]:
$a + \infty = \infty\ \infty + a = \infty$
⟨*proof*⟩

lemma *inf-lt*[*simp*,*dest*!]:
$\infty < x \Longrightarrow False$
⟨*proof*⟩

lemma *inf-lt-impl-False*[*simp*]:
$\infty < x = False$
⟨*proof*⟩

lemma *Le-Le-dbm-lt-D*[*dest*]: $Le\ a \prec Lt\ b \Longrightarrow a < b$ ⟨*proof*⟩
lemma *Le-Lt-dbm-lt-D*[*dest*]: $Le\ a \prec Le\ b \Longrightarrow a < b$ ⟨*proof*⟩
lemma *Lt-Le-dbm-lt-D*[*dest*]: $Lt\ a \prec Le\ b \Longrightarrow a \leq b$ ⟨*proof*⟩
lemma *Lt-Lt-dbm-lt-D*[*dest*]: $Lt\ a \prec Lt\ b \Longrightarrow a < b$ ⟨*proof*⟩

lemma *Le-le-LeI*[*intro*]: $a \leq b \Longrightarrow Le\ a \leq Le\ b$ ⟨*proof*⟩
lemma *Lt-le-LeI*[*intro*]: $a \leq b \Longrightarrow Lt\ a \leq Le\ b$ ⟨*proof*⟩
lemma *Lt-le-LtI*[*intro*]: $a \leq b \Longrightarrow Lt\ a \leq Lt\ b$ ⟨*proof*⟩
lemma *Le-le-LtI*[*intro*]: $a < b \Longrightarrow Le\ a \leq Lt\ b$ ⟨*proof*⟩
lemma *Lt-lt-LeI*: $x \leq y \Longrightarrow Lt\ x < Le\ y$ ⟨*proof*⟩

lemma *Le-le-LeD*[*dest*]: $Le\ a \leq Le\ b \Longrightarrow a \leq b$ ⟨*proof*⟩
lemma *Le-le-LtD*[*dest*]: $Le\ a \leq Lt\ b \Longrightarrow a < b$ ⟨*proof*⟩
lemma *Lt-le-LeD*[*dest*]: $Lt\ a \leq Le\ b \Longrightarrow a \leq b$ ⟨*proof*⟩
lemma *Lt-le-LtD*[*dest*]: $Lt\ a \leq Lt\ b \Longrightarrow a \leq b$ ⟨*proof*⟩

lemma *inf-not-le-Le*[*simp*]: $\infty \leq Le\ x = False$ ⟨*proof*⟩
lemma *inf-not-le-Lt*[*simp*]: $\infty \leq Lt\ x = False$ ⟨*proof*⟩
lemma *inf-not-lt*[*simp*]: $\infty \prec x = False$ ⟨*proof*⟩

lemma *any-le-inf*: $x \leq (\infty :: \text{-}\ DBMEntry)$ ⟨*proof*⟩

lemma *dbm-lt-code-simps*[*code*]:
dbm-lt (*Lt a*) ∞ = *True*
dbm-lt (*Le a*) ∞ = *True*
dbm-lt (*Le a*) (*Le b*) = (*a* < *b*)
dbm-lt (*Le a*) (*Lt b*) = (*a* < *b*)
dbm-lt (*Lt a*) (*Le b*) = (*a* ≤ *b*)
dbm-lt (*Lt a*) (*Lt b*) = (*a* < *b*)
dbm-lt ∞ *x* = *False*
⟨*proof*⟩

1.3 Basic Properties of DBMs

1.3.1 DBMs and Length of Paths

lemma *dbm-entry-val-add-1*: *dbm-entry-val u* (*Some c*) (*Some d*) *a* ⟹ *dbm-entry-val u* (*Some d*) *None b*
⟹ *dbm-entry-val u* (*Some c*) *None* (*dbm-add a b*)
⟨*proof*⟩

lemma *dbm-entry-val-add-2*: *dbm-entry-val u None* (*Some c*) *a* ⟹ *dbm-entry-val u* (*Some c*) (*Some d*) *b*
⟹ *dbm-entry-val u None* (*Some d*) (*dbm-add a b*)
⟨*proof*⟩

lemma *dbm-entry-val-add-3*:
dbm-entry-val u (*Some c*) (*Some d*) *a* ⟹ *dbm-entry-val u* (*Some d*) (*Some e*) *b*
⟹ *dbm-entry-val u* (*Some c*) (*Some e*) (*dbm-add a b*)
⟨*proof*⟩

lemma *dbm-entry-val-add-4*:
dbm-entry-val u (*Some c*) *None a* ⟹ *dbm-entry-val u None* (*Some d*) *b*
⟹ *dbm-entry-val u* (*Some c*) (*Some d*) (*dbm-add a b*)
⟨*proof*⟩

no-notation *dbm-add* (**infixl** ⊗ *70*)

lemma *DBM-val-bounded-len-1'-aux*:
assumes *DBM-val-bounded v u m n v c* ≤ *n* ∀ *k* ∈ *set vs*. *k* > *0* ∧ *k* ≤ *n* ∧ (∃ *c*. *v c* = *k*)
shows *dbm-entry-val u* (*Some c*) *None* (*len m* (*v c*) *0 vs*) ⟨*proof*⟩

lemma *DBM-val-bounded-len-3′-aux*:
DBM-val-bounded v u m n $\Longrightarrow$ *v c* $\leq$ *n* $\Longrightarrow$ *v d* $\leq$ *n* $\Longrightarrow$ $\forall$ *k* $\in$ *set vs. k* $>$ *0* $\wedge$ *k* $\leq$ *n* $\wedge$ ($\exists$ *c. v c* $=$ *k*)
$\Longrightarrow$ *dbm-entry-val u* (*Some c*) (*Some d*) (*len m* (*v c*) (*v d*) *vs*)
⟨*proof*⟩

lemma *DBM-val-bounded-len-2′-aux*:
DBM-val-bounded v u m n $\Longrightarrow$ *v c* $\leq$ *n* $\Longrightarrow$ $\forall$ *k* $\in$ *set vs. k* $>$ *0* $\wedge$ *k* $\leq$ *n* $\wedge$ ($\exists$ *c. v c* $=$ *k*)
$\Longrightarrow$ *dbm-entry-val u None* (*Some c*) (*len m 0* (*v c*) *vs*)
⟨*proof*⟩

lemma *cnt-0-D*:
cnt x xs $=$ *0* $\Longrightarrow$ *x* $\notin$ *set xs*
⟨*proof*⟩

lemma *cnt-at-most-1-D*:
cnt x (*xs* @ *x* # *ys*) $\leq$ *1* $\Longrightarrow$ *x* $\notin$ *set xs* $\wedge$ *x* $\notin$ *set ys*
⟨*proof*⟩

lemma *nat-list-0* [*intro*]:
x $\in$ *set xs* $\Longrightarrow$ *0* $\notin$ *set* (*xs* :: *nat list*) $\Longrightarrow$ *x* $>$ *0*
⟨*proof*⟩

lemma *DBM-val-bounded-len′1*:
fixes *v*
assumes *DBM-val-bounded v u m n 0* $\notin$ *set vs v c* $\leq$ *n*
$\forall$ *k* $\in$ *set vs. k* $>$ *0* $\longrightarrow$ *k* $\leq$ *n* $\wedge$ ($\exists$ *c. v c* $=$ *k*)
shows *dbm-entry-val u* (*Some c*) *None* (*len m* (*v c*) *0 vs*)
⟨*proof*⟩

lemma *DBM-val-bounded-len′2*:
fixes *v*
assumes *DBM-val-bounded v u m n 0* $\notin$ *set vs v c* $\leq$ *n*
$\forall$ *k* $\in$ *set vs. k* $>$ *0* $\longrightarrow$ *k* $\leq$ *n* $\wedge$ ($\exists$ *c. v c* $=$ *k*)
shows *dbm-entry-val u None* (*Some c*) (*len m 0* (*v c*) *vs*)
⟨*proof*⟩

lemma *DBM-val-bounded-len′3*:
fixes *v*
assumes *DBM-val-bounded v u m n cnt 0 vs* $\leq$ *1 v c1* $\leq$ *n v c2* $\leq$ *n*
$\forall$ *k* $\in$ *set vs. k* $>$ *0* $\longrightarrow$ *k* $\leq$ *n* $\wedge$ ($\exists$ *c. v c* $=$ *k*)
shows *dbm-entry-val u* (*Some c1*) (*Some c2*) (*len m* (*v c1*) (*v c2*) *vs*)

⟨*proof*⟩

Now unused **lemma** *DBM-val-bounded-len'*:
fixes *v*
defines *vo* ≡ λ *k*. *if k* = *0 then None else Some* (*SOME c*. *v c* = *k*)
assumes *DBM-val-bounded v u m n cnt 0* (*i* # *j* # *vs*) ≤ *1*
∀ *k* ∈ *set* (*i* # *j* # *vs*). *k* > *0* ⟶ *k* ≤ *n* ∧ (∃ *c*. *v c* = *k*)
shows *dbm-entry-val u* (*vo i*) (*vo j*) (*len m i j vs*)
⟨*proof*⟩

lemma *DBM-val-bounded-len-1*: *DBM-val-bounded v u m n* ⟹ *v c* ≤ *n*
⟹ ∀ *c* ∈ *set cs*. *v c* ≤ *n*
⟹ *dbm-entry-val u* (*Some c*) *None* (*len m* (*v c*) *0* (*map v cs*))
⟨*proof*⟩

lemma *DBM-val-bounded-len-3*: *DBM-val-bounded v u m n* ⟹ *v c* ≤ *n*
⟹ *v d* ≤ *n* ⟹ ∀ *c* ∈ *set cs*. *v c* ≤ *n*
⟹ *dbm-entry-val u* (*Some c*) (*Some d*) (*len m* (*v c*) (*v d*) (*map v cs*))
⟨*proof*⟩

lemma *DBM-val-bounded-len-2*: *DBM-val-bounded v u m n* ⟹ *v c* ≤ *n*
⟹ ∀ *c* ∈ *set cs*. *v c* ≤ *n*
⟹ *dbm-entry-val u None* (*Some c*) (*len m 0* (*v c*) (*map v cs*))
⟨*proof*⟩

lemmas *DBM-arith-defs* = *add neutral uminus*

end
theory *Paths-Cycles*
imports *Floyd-Warshall.Floyd-Warshall*
begin

2 Library for Paths, Arcs and Lengths

lemma *length-eq-distinct*:
assumes *set xs* = *set ys distinct xs length xs* = *length ys*
shows *distinct ys*
⟨*proof*⟩

2.1 Arcs

fun *arcs* :: *nat* ⇒ *nat* ⇒ *nat list* ⇒ (*nat* ∗ *nat*) *list* **where**
arcs a b [] = [(*a*,*b*)] |

arcs a b (x # xs) = (a, x) # arcs x b xs

definition *arcs' :: nat list ⇒ (nat ∗ nat) set* **where**
arcs' xs = set (arcs (hd xs) (last xs) (butlast (tl xs)))

lemma *arcs'-decomp*:
length xs > 1 ⟹ (i, j) ∈ arcs' xs ⟹ ∃ zs ys. xs = zs @ i # j # ys
⟨*proof*⟩

lemma *arcs-decomp-tail*:
arcs j l (ys @ [i]) = arcs j i ys @ [(i, l)]
⟨*proof*⟩

lemma *arcs-decomp*: *xs = ys @ y # zs ⟹ arcs x z xs = arcs x y ys @ arcs y z zs*
⟨*proof*⟩

lemma *distinct-arcs-ex*:
distinct xs ⟹ i ∉ set xs ⟹ xs ≠ [] ⟹ ∃ a b. a ≠ x ∧ (a,b) ∈ set (arcs i j xs)
⟨*proof*⟩

lemma *cycle-rotate-2-aux*:
(i, j) ∈ set (arcs a b (xs @ [c])) ⟹ (i,j) ≠ (c,b) ⟹ (i, j) ∈ set (arcs a c xs)
⟨*proof*⟩

lemma *arcs-set-elem1*:
assumes *j ≠ k k ∈ set (i # xs)*
shows *∃ l. (k, l) ∈ set (arcs i j xs)* ⟨*proof*⟩

lemma *arcs-set-elem2*:
assumes *i ≠ k k ∈ set (j # xs)*
shows *∃ l. (l, k) ∈ set (arcs i j xs)* ⟨*proof*⟩

2.2 Length of Paths

lemmas (**in** *linordered-ab-monoid-add*) *comm = add.commute*

lemma *len-add*:
fixes *M :: ('a :: linordered-ab-monoid-add) mat*
shows *len M i j xs + len M i j xs = len (λi j. M i j + M i j) i j xs*
⟨*proof*⟩

2.3 Cycle Rotation

lemma *cycle-rotate*:
fixes $M :: ('a :: linordered\text{-}ab\text{-}monoid\text{-}add)\ mat$
assumes $length\ xs > 1\ (i, j) \in arcs'\ xs$
shows $\exists\ ys\ zs.\ len\ M\ a\ a\ xs = len\ M\ i\ i\ (j \# ys\ @\ a \# zs) \wedge xs = zs\ @\ i \# j \# ys$ ⟨*proof*⟩

lemma *cycle-rotate-2*:
fixes $M :: ('a :: linordered\text{-}ab\text{-}monoid\text{-}add)\ mat$
assumes $xs \neq []\ (i, j) \in set\ (arcs\ a\ a\ xs)$
shows $\exists\ ys.\ len\ M\ a\ a\ xs = len\ M\ i\ i\ (j \# ys) \wedge set\ ys \subseteq set\ (a \# xs) \wedge length\ ys < length\ xs$
⟨*proof*⟩

lemma *cycle-rotate-len-arcs*:
fixes $M :: ('a :: linordered\text{-}ab\text{-}monoid\text{-}add)\ mat$
assumes $length\ xs > 1\ (i, j) \in arcs'\ xs$
shows $\exists\ ys\ zs.\ len\ M\ a\ a\ xs = len\ M\ i\ i\ (j \# ys\ @\ a \# zs)$
$\wedge\ set\ (arcs\ a\ a\ xs) = set\ (arcs\ i\ i\ (j \# ys\ @\ a \# zs)) \wedge xs = zs\ @\ i \# j \# ys$
⟨*proof*⟩

lemma *cycle-rotate-2'*:
fixes $M :: ('a :: linordered\text{-}ab\text{-}monoid\text{-}add)\ mat$
assumes $xs \neq []\ (i, j) \in set\ (arcs\ a\ a\ xs)$
shows $\exists\ ys.\ len\ M\ a\ a\ xs = len\ M\ i\ i\ (j \# ys) \wedge set\ (i \# j \# ys) = set\ (a \# xs)$
$\wedge\ 1 + length\ ys = length\ xs \wedge set\ (arcs\ a\ a\ xs) = set\ (arcs\ i\ i\ (j \# ys))$
⟨*proof*⟩

2.4 More on Cycle-Freeness

lemma *cyc-free-diag-dest*:
assumes $cyc\text{-}free\ M\ n\ i \leq n\ set\ xs \subseteq \{0..n\}$
shows $len\ M\ i\ i\ xs \geq 0$
⟨*proof*⟩

lemma *cycle-free-0-0*:
fixes $M :: ('a :: linordered\text{-}ab\text{-}monoid\text{-}add)\ mat$
assumes $cycle\text{-}free\ M\ n$
shows $M\ 0\ 0 \geq 0$
⟨*proof*⟩

2.5 Helper Lemmas for Bouyer's Theorem on Approximation

lemma *aux1*: $i \leq n \Longrightarrow j \leq n \Longrightarrow set\ xs \subseteq \{0..n\} \Longrightarrow (a,b) \in set\ (arcs\ i\ j\ xs) \Longrightarrow a \leq n \wedge b \leq n$
⟨*proof*⟩

lemma *arcs-distinct1*:
$i \notin set\ xs \Longrightarrow j \notin set\ xs \Longrightarrow distinct\ xs \Longrightarrow xs \neq [] \Longrightarrow (a,b) \in set\ (arcs\ i\ j\ xs) \Longrightarrow a \neq b$
⟨*proof*⟩

lemma *arcs-distinct2*:
$i \notin set\ xs \Longrightarrow j \notin set\ xs \Longrightarrow distinct\ xs \Longrightarrow i \neq j \Longrightarrow (a,b) \in set\ (arcs\ i\ j\ xs) \Longrightarrow a \neq b$
⟨*proof*⟩

lemma *arcs-distinct3*: $distinct\ (a\ \#\ b\ \#\ c\ \#\ xs) \Longrightarrow (i,j) \in set\ (arcs\ a\ b\ xs) \Longrightarrow i \neq c \wedge j \neq c$
⟨*proof*⟩

lemma *arcs-elem*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)$ **shows** $a \in set\ (i\ \#\ xs)\ b \in set\ (j\ \#\ xs)$
⟨*proof*⟩

lemma *arcs-distinct-dest1*:
$distinct\ (i\ \#\ a\ \#\ zs) \Longrightarrow (b,c) \in set\ (arcs\ a\ j\ zs) \Longrightarrow b \neq i$
⟨*proof*⟩

lemma *arcs-distinct-fix*:
$distinct\ (a\ \#\ x\ \#\ xs\ @\ [b]) \Longrightarrow (a,c) \in set\ (arcs\ a\ b\ (x\ \#\ xs)) \Longrightarrow c = x$
⟨*proof*⟩

lemma *disjE3*: $A \vee B \vee C \Longrightarrow (A \Longrightarrow G) \Longrightarrow (B \Longrightarrow G) \Longrightarrow (C \Longrightarrow G) \Longrightarrow G$
⟨*proof*⟩

lemma *arcs-predecessor*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)\ a \neq i$
shows $\exists\ c.\ (c,\ a) \in set\ (arcs\ i\ j\ xs)$ ⟨*proof*⟩

lemma *arcs-successor*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)\ b \neq j$

shows $\exists\ c.\ (b,c) \in set\ (arcs\ i\ j\ xs)$ ⟨*proof*⟩

lemma *arcs-predecessor′*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ (x\ \#\ xs))\ (a,b) \neq (i,\ x)$
shows $\exists\ c.\ (c,\ a) \in set\ (arcs\ i\ j\ (x\ \#\ xs))$ ⟨*proof*⟩

lemma *arcs-cases*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)\ xs \neq []$
shows $(\exists\ ys.\ xs = b\ \#\ ys \wedge a = i) \vee (\exists\ ys.\ xs = ys\ @\ [a] \wedge b = j)$
$\vee\ (\exists\ c\ d\ ys.\ (a,b) \in set\ (arcs\ c\ d\ ys) \wedge xs = c\ \#\ ys\ @\ [d])$
⟨*proof*⟩

lemma *arcs-cases′*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)\ xs \neq []$
shows $(\exists\ ys.\ xs = b\ \#\ ys \wedge a = i) \vee (\exists\ ys.\ xs = ys\ @\ [a] \wedge b = j) \vee$
$(\exists\ ys\ zs.\ xs = ys\ @\ a\ \#\ b\ \#\ zs)$
⟨*proof*⟩

lemma *arcs-successor′*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)\ b \neq j$
shows $\exists\ c.\ xs = [b] \wedge a = i \vee (\exists\ ys.\ xs = b\ \#\ c\ \#\ ys \wedge a = i) \vee (\exists\ ys.$
$xs = ys\ @\ [a,b] \wedge c = j)$
$\vee\ (\exists\ ys\ zs.\ xs = ys\ @\ a\ \#\ b\ \#\ c\ \#\ zs)$
⟨*proof*⟩

lemma *list-last*:
$xs = [] \vee (\exists\ y\ ys.\ xs = ys\ @\ [y])$
⟨*proof*⟩

lemma *arcs-predecessor″*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)\ a \neq i$
shows $\exists\ c.\ xs = [a] \vee (\exists\ ys.\ xs = a\ \#\ b\ \#\ ys) \vee (\exists\ ys.\ xs = ys\ @\ [c,a]$
$\wedge\ b = j)$
$\vee\ (\exists\ ys\ zs.\ xs = ys\ @\ c\ \#\ a\ \#\ b\ \#\ zs)$
⟨*proof*⟩

lemma *arcs-ex-middle*:
$\exists\ b.\ (a,\ b) \in set\ (arcs\ i\ j\ (ys\ @\ a\ \#\ xs))$
⟨*proof*⟩

lemma *arcs-ex-head*:
$\exists\ b.\ (i,\ b) \in set\ (arcs\ i\ j\ xs)$
⟨*proof*⟩

2.5.1 Successive

fun *successive* **where**
successive - [] = *True* |
successive P [-] = *True* |
successive P (*x* # *y* # *xs*) ⟷ ¬ *P y* ∧ *successive P xs* ∨ ¬ *P x* ∧ *successive P* (*y* # *xs*)

lemma ¬ *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *Suc 0*] ⟨*proof*⟩
lemma *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*] ⟨*proof*⟩
lemma *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *0*, *Suc 0*] ⟨*proof*⟩
lemma ¬ *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *0*, *Suc 0*, *Suc 0*] ⟨*proof*⟩
lemma ¬ *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *0*, *0*, *Suc 0*, *Suc 0*] ⟨*proof*⟩
lemma *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *0*, *Suc 0*, *0*, *Suc 0*] ⟨*proof*⟩
lemma ¬ *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *Suc 0*, *0*, *Suc 0*] ⟨*proof*⟩
lemma *successive* (λ *x*. *x* > (*0* :: *nat*)) [*0*, *0*, *Suc 0*, *0*] ⟨*proof*⟩

lemma *successive-step*: *successive P* (*x* # *xs*) ⟹ ¬ *P x* ⟹ *successive P xs*
⟨*proof*⟩

lemma *successive-step-2*: *successive P* (*x* # *y* # *xs*) ⟹ ¬ *P y* ⟹ *successive P xs*
⟨*proof*⟩

lemma *successive-stepI*:
successive P xs ⟹ ¬ *P x* ⟹ *successive P* (*x* # *xs*)
⟨*proof*⟩

lemmas *list-two-induct*[*case-names Nil Single Cons*] = *induct-list012*

lemma *successive-end-1*:
successive P xs ⟹ ¬ *P x* ⟹ *successive P* (*xs* @ [*x*])
⟨*proof*⟩

lemma *successive-ends-1*:
successive P xs ⟹ ¬ *P x* ⟹ *successive P ys* ⟹ *successive P* (*xs* @ *x* # *ys*)
⟨*proof*⟩

lemma *successive-ends-1'*:
successive P xs ⟹ ¬ *P x* ⟹ *P y* ⟹ ¬ *P z* ⟹ *successive P ys* ⟹ *successive P* (*xs* @ *x* # *y* # *z* # *ys*)

⟨*proof*⟩

lemma *successive-end-2*:
successive P xs ⟹ ¬ *P x* ⟹ *successive P* (*xs* @ [*x*,*y*])
⟨*proof*⟩

lemma *successive-end-2′*:
successive P (*xs* @ [*x*]) ⟹ ¬ *P x* ⟹ *successive P* (*xs* @ [*x*,*y*])
⟨*proof*⟩

lemma *successive-end-3*:
successive P (*xs* @ [*x*]) ⟹ ¬ *P x* ⟹ *P y* ⟹ ¬ *P z* ⟹ *successive P* (*xs* @ [*x*,*y*,*z*])
⟨*proof*⟩

lemma *successive-step-rev*:
successive P (*xs* @ [*x*]) ⟹ ¬ *P x* ⟹ *successive P xs*
⟨*proof*⟩

lemma *successive-glue*:
successive P (*zs* @ [*z*]) ⟹ *successive P* (*x* # *xs*) ⟹ ¬ *P z* ∨ ¬ *P x* ⟹ *successive P* (*zs* @ [*z*] @ *x* # *xs*)
⟨*proof*⟩

lemma *successive-glue′*:
successive P (*zs* @ [*y*]) ∧ ¬ *P z* ∨ *successive P zs* ∧ ¬ *P y*
⟹ *successive P* (*x* # *xs*) ∧ ¬ *P w* ∨ *successive P xs* ∧ ¬ *P x*
⟹ ¬ *P z* ∨ ¬ *P w* ⟹ *successive P* (*zs* @ *y* # *z* # *w* # *x* # *xs*)
⟨*proof*⟩

lemma *successive-dest-head*:
xs = *w* # *x* # *ys* ⟹ *successive P xs* ⟹ *successive P* (*x* # *xs*) ∧ ¬ *P w* ∨ *successive P xs* ∧ ¬ *P x*
⟨*proof*⟩

lemma *successive-dest-tail*:
xs = *zs* @ [*y*,*z*] ⟹ *successive P xs*
⟹ *successive P* (*zs* @ [*y*]) ∧ ¬ *P z* ∨ *successive P zs* ∧ ¬ *P y*
⟨*proof*⟩

lemma *successive-split*:
xs = *ys* @ *zs* ⟹ *successive P xs* ⟹ *successive P ys* ∧ *successive P zs*
⟨*proof*⟩

lemma *successive-decomp*:
$xs = x \# ys \mathbin{@} zs \mathbin{@} [z] \Longrightarrow$ *successive* P $xs \Longrightarrow \neg\ P\ x \vee \neg\ P\ z \Longrightarrow$ *successive* P $(zs \mathbin{@} [z] \mathbin{@} (x \# ys))$
⟨*proof*⟩

lemma *successive-predecessor*:
assumes $(a, b) \in$ *set* $(arcs\ i\ j\ xs)$ $a \neq i$ *successive* P $(arcs\ i\ j\ xs)$ P (a,b) $xs \neq []$
shows $\exists\ c.\ (xs = [a] \wedge c = i \vee (\exists\ ys.\ xs = a \# b \# ys \wedge c = i) \vee (\exists\ ys.\ xs = ys \mathbin{@} [c,a] \wedge b = j)$
$\vee\ (\exists\ ys\ zs.\ xs = ys \mathbin{@} c \# a \# b \# zs)) \wedge \neg\ P\ (c,a)$
⟨*proof*⟩

lemma *successive-successor*:
assumes $(a, b) \in$ *set* $(arcs\ i\ j\ xs)$ $b \neq j$ *successive* P $(arcs\ i\ j\ xs)$ P (a,b) $xs \neq []$
shows $\exists\ c.\ (xs = [b] \wedge c = j \vee (\exists\ ys.\ xs = b \# c \# ys) \vee (\exists\ ys.\ xs = ys \mathbin{@} [a,b] \wedge c = j)$
$\vee\ (\exists\ ys\ zs.\ xs = ys \mathbin{@} a \# b \# c \# zs)) \wedge \neg\ P\ (b,c)$
⟨*proof*⟩

lemmas *add-mono-right* = *add-mono*[*OF order-refl*]
lemmas *add-mono-left* = *add-mono*[*OF* - *order-refl*]

Obtaining successive and distinct paths **lemma** *canonical-successive*:
fixes A B
defines $M \equiv \lambda\ i\ j.\ min\ (A\ i\ j)\ (B\ i\ j)$
assumes *canonical* A n
assumes *set* $xs \subseteq \{0..n\}$
assumes $i \leq n$ $j \leq n$
shows $\exists\ ys.\ len\ M\ i\ j\ ys \leq len\ M\ i\ j\ xs \wedge set\ ys \subseteq \{0..n\}$
$\wedge$ *successive* $(\lambda\ (a, b).\ M\ a\ b = A\ a\ b)$ $(arcs\ i\ j\ ys)$
⟨*proof*⟩

lemma *canonical-successive-distinct*:
fixes A B
defines $M \equiv \lambda\ i\ j.\ min\ (A\ i\ j)\ (B\ i\ j)$
assumes *canonical* A n
assumes *set* $xs \subseteq \{0..n\}$
assumes $i \leq n$ $j \leq n$
assumes *distinct* xs $i \notin set\ xs$ $j \notin set\ xs$
shows $\exists\ ys.\ len\ M\ i\ j\ ys \leq len\ M\ i\ j\ xs \wedge set\ ys \subseteq set\ xs$
$\wedge$ *successive* $(\lambda\ (a, b).\ M\ a\ b = A\ a\ b)$ $(arcs\ i\ j\ ys)$

$\wedge$ *distinct ys* $\wedge$ $i \notin$ *set ys* $\wedge$ $j \notin$ *set ys*
⟨*proof*⟩

lemma *successive-snd-last*: *successive P* (*xs* @ [*x*, *y*]) $\Longrightarrow$ *P y* $\Longrightarrow$ $\neg$ *P x*
⟨*proof*⟩

lemma *canonical-shorten-rotate-neg-cycle*:
fixes *A B*
defines $M \equiv \lambda\ i\ j.$ *min* (*A i j*) (*B i j*)
assumes *canonical A n*
assumes *set xs* $\subseteq \{0..n\}$
assumes $i \leq n$
assumes *len M i i xs* < 0
shows $\exists$ *j ys. len M j j ys* $< 0 \wedge$ *set* (*j* # *ys*) $\subseteq$ *set* (*i* # *xs*)
$\wedge$ *successive* (λ (*a*, *b*). *M a b* = *A a b*) (*arcs j j ys*)
$\wedge$ *distinct ys* $\wedge$ $j \notin$ *set ys* $\wedge$
(*ys* $\neq$ [] $\longrightarrow$ *M j* (*hd ys*) $\neq$ *A j* (*hd ys*) $\vee$ *M* (*last ys*) *j* $\neq$ *A* (*last ys*) *j*)
⟨*proof*⟩

lemma *successive-arcs-extend-last*:
successive P (*arcs i j xs*) $\Longrightarrow$ $\neg$ *P* (*i*, *hd xs*) $\vee$ $\neg$ *P* (*last xs*, *j*) $\Longrightarrow$ *xs* $\neq$ []
$\Longrightarrow$ *successive P* (*arcs i j xs* @ [(*i*, *hd xs*)])
⟨*proof*⟩

lemma *cycle-rotate-arcs*:
fixes *M* :: ('*a* :: *linordered-ab-monoid-add*) *mat*
assumes *length xs* $>$ *1* (*i*, *j*) $\in$ *arcs*′ *xs*
shows $\exists$ *ys zs. set* (*arcs a a xs*) = *set* (*arcs i i* (*j* # *ys* @ *a* # *zs*)) $\wedge$ *xs* = *zs* @ *i* # *j* # *ys* ⟨*proof*⟩

lemma *cycle-rotate-len-arcs-successive*:
fixes *M* :: ('*a* :: *linordered-ab-monoid-add*) *mat*
assumes *length xs* $>$ *1* (*i*, *j*) $\in$ *arcs*′ *xs successive P* (*arcs a a xs*) $\neg$ *P* (*a*, *hd xs*) $\vee$ $\neg$ *P* (*last xs*, *a*)
shows $\exists$ *ys zs. len M a a xs* = *len M i i* (*j* # *ys* @ *a* # *zs*)
$\wedge$ *set* (*arcs a a xs*) = *set* (*arcs i i* (*j* # *ys* @ *a* # *zs*)) $\wedge$ *xs* = *zs* @ *i* # *j* # *ys*
$\wedge$ *successive P* (*arcs i i* (*j* # *ys* @ *a* # *zs*))
⟨*proof*⟩

lemma *successive-successors*:
xs = *ys* @ *a* # *b* # *c* # *zs* $\Longrightarrow$ *successive P* (*arcs i j xs*) $\Longrightarrow$ $\neg$ *P* (*a*,*b*)

$\vee \neg P\ (b,\ c)$
⟨*proof*⟩

lemma *successive-successors′*:
$xs = ys\ @\ a\ \#\ b\ \#\ zs \Longrightarrow successive\ P\ xs \Longrightarrow \neg P\ a \vee \neg P\ b$
⟨*proof*⟩

lemma *cycle-rotate-len-arcs-successive′*:
fixes $M :: ('a :: linordered\text{-}ab\text{-}monoid\text{-}add)\ mat$
assumes $length\ xs > 1\ (i,\ j) \in arcs'\ xs\ successive\ P\ (arcs\ a\ a\ xs)$
$\neg P\ (a,\ hd\ xs) \vee \neg P\ (last\ xs,\ a)$
shows $\exists\ ys\ zs.\ len\ M\ a\ a\ xs = len\ M\ i\ i\ (j\ \#\ ys\ @\ a\ \#\ zs)$
$\wedge\ set\ (arcs\ a\ a\ xs) = set\ (arcs\ i\ i\ (j\ \#\ ys\ @\ a\ \#\ zs)) \wedge xs = zs\ @\ i\ \#\ j\ \#\ ys$
$\wedge\ successive\ P\ (arcs\ i\ i\ (j\ \#\ ys\ @\ a\ \#\ zs)\ @\ [(i,j)])$
⟨*proof*⟩

lemma *cycle-rotate-3*:
fixes $M :: ('a :: linordered\text{-}ab\text{-}monoid\text{-}add)\ mat$
assumes $xs \neq []\ (i,\ j) \in set\ (arcs\ a\ a\ xs)\ successive\ P\ (arcs\ a\ a\ xs) \neg P\ (a,\ hd\ xs) \vee \neg P\ (last\ xs,\ a)$
shows $\exists\ ys.\ len\ M\ a\ a\ xs = len\ M\ i\ i\ (j\ \#\ ys) \wedge set\ (i\ \#\ j\ \#\ ys) = set\ (a\ \#\ xs) \wedge 1 + length\ ys = length\ xs$
$\wedge\ set\ (arcs\ a\ a\ xs) = set\ (arcs\ i\ i\ (j\ \#\ ys))$
$\wedge\ successive\ P\ (arcs\ i\ i\ (j\ \#\ ys))$
⟨*proof*⟩

lemma *cycle-rotate-3′*:
fixes $M :: ('a :: linordered\text{-}ab\text{-}monoid\text{-}add)\ mat$
assumes $xs \neq []\ (i,\ j) \in set\ (arcs\ a\ a\ xs)\ successive\ P\ (arcs\ a\ a\ xs) \neg P\ (a,\ hd\ xs) \vee \neg P\ (last\ xs,\ a)$
shows $\exists\ ys.\ len\ M\ a\ a\ xs = len\ M\ i\ i\ (j\ \#\ ys) \wedge set\ (i\ \#\ j\ \#\ ys) = set\ (a\ \#\ xs) \wedge 1 + length\ ys = length\ xs$
$\wedge\ set\ (arcs\ a\ a\ xs) = set\ (arcs\ i\ i\ (j\ \#\ ys))$
$\wedge\ successive\ P\ (arcs\ i\ i\ (j\ \#\ ys)\ @\ [(i,\ j)])$
⟨*proof*⟩

end

2.5.2 Zones and DBMs

theory *Zones*
imports *DBM*
begin

type-synonym $('c, 't)\ zone = ('c, 't)\ cval\ set$

type-synonym $('c, 't)\ cval = 'c \Rightarrow 't$

definition *cval-add* :: $('c,'t)\ cval \Rightarrow 't::plus \Rightarrow ('c,'t)\ cval$ (**infixr** $\oplus$ *64*)
where
$u \oplus d = (\lambda\ x.\ u\ x + d)$

definition *zone-delay* :: $('c, ('t::time))\ zone \Rightarrow ('c, 't)\ zone$
($-^{\uparrow}$ [*71*] *71*)
where
$Z^{\uparrow} = \{u \oplus d | u\ d.\ u \in Z \wedge d \geq (0::'t)\}$

fun *clock-set* :: $'c\ list \Rightarrow 't::time \Rightarrow ('c,'t)\ cval \Rightarrow ('c,'t)\ cval$
where
clock-set [] - $u = u$ |
clock-set $(c\#cs)\ t\ u =$ (*clock-set cs t u*)($c:=t$)

abbreviation *clock-set-abbrv* :: $'c\ list \Rightarrow 't::time \Rightarrow ('c,'t)\ cval \Rightarrow ('c,'t)$ *cval*
([-→-]- [*65*,*65*,*65*] *65*)
where
$[r \rightarrow t]u \equiv$ *clock-set r t u*

definition *zone-set* :: $('c, 't::time)\ zone \Rightarrow 'c\ list \Rightarrow ('c, 't)\ zone$
($-_{-} \rightarrow_0$ [*71*] *71*)
where
zone-set $Z\ r = \{[r \rightarrow (0::'t)]u \mid u\ .\ u \in Z\}$

lemma *clock-set-set*[*simp*]:
$([r \rightarrow d]u)\ c = d$ **if** $c \in set\ r$
⟨*proof*⟩

lemma *clock-set-id*[*simp*]:
$([r \rightarrow d]u)\ c = u\ c$ **if** $c \notin set\ r$
⟨*proof*⟩

definition *DBM-zone-repr* :: $('t::time)\ DBM \Rightarrow ('c \Rightarrow nat) \Rightarrow nat \Rightarrow ('c,$ $'t :: time)\ zone$
([-]-,- [*72*,*72*,*72*] *72*)
where
$[M]_{v,n} = \{u\ .\ DBM\text{-}val\text{-}bounded\ v\ u\ M\ n\}$

lemma *dbm-entry-val-mono1*:
dbm-entry-val u (*Some c*) (*Some c'*) *b* $\Longrightarrow$ *b* $\preceq$ *b'* $\Longrightarrow$ *dbm-entry-val u* (*Some c*) (*Some c'*) *b'*
⟨*proof*⟩

lemma *dbm-entry-val-mono2*:
dbm-entry-val u None (*Some c*) *b* $\Longrightarrow$ *b* $\preceq$ *b'* $\Longrightarrow$ *dbm-entry-val u None* (*Some c*) *b'*
⟨*proof*⟩

lemma *dbm-entry-val-mono3*:
dbm-entry-val u (*Some c*) *None b* $\Longrightarrow$ *b* $\preceq$ *b'* $\Longrightarrow$ *dbm-entry-val u* (*Some c*) *None b'*
⟨*proof*⟩

lemmas *dbm-entry-val-mono* = *dbm-entry-val-mono1 dbm-entry-val-mono2 dbm-entry-val-mono3*

lemma *DBM-le-subset*:
$\forall$ *i j*. *i* $\leq$ *n* $\longrightarrow$ *j* $\leq$ *n* $\longrightarrow$ *M i j* $\preceq$ *M' i j* $\Longrightarrow$ *u* $\in$ $[M]_{v,n}$ $\Longrightarrow$ *u* $\in$ $[M']_{v,n}$
⟨*proof*⟩

end
theory *DBM-Basics*
imports
DBM
Paths-Cycles
Zones
begin

2.5.3 Useful definitions

fun *get-const* **where**
get-const (*Le c*) = *c* |
get-const (*Lt c*) = *c* |
get-const (∞ :: - *DBMEntry*) = *undefined*

2.5.4 Updating DBMs

abbreviation *DBM-update* :: ('*t*::*time*) *DBM* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ ('*t DBMEntry*) $\Rightarrow$ ('*t*::*time*) *DBM*
where
DBM-update M m n v $\equiv$ (λ *x y*. *if m* = *x* $\wedge$ *n* = *y then v else M x y*)

fun *DBM-upd* :: ('*t*::*time*) *DBM* ⇒ (*nat* ⇒ *nat* ⇒ '*t DBMEntry*) ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ '*t DBM*
where
DBM-upd M f 0 0 - = DBM-update M 0 0 (f 0 0) |
DBM-upd M f (Suc i) 0 n = DBM-update (DBM-upd M f i n n) (Suc i) 0 (f (Suc i) 0) |
DBM-upd M f i (Suc j) n = DBM-update (DBM-upd M f i j n) i (Suc j) (f i (Suc j))

lemma *upd-1*:
assumes $j \leq n$
shows *DBM-upd M1 f (Suc m) n N (Suc m) j = DBM-upd M1 f (Suc m) j N (Suc m) j*
⟨*proof*⟩

lemma *upd-2*:
assumes $i \leq m$
shows *DBM-upd M1 f (Suc m) n N i j = DBM-upd M1 f (Suc m) 0 N i j*
⟨*proof*⟩

lemma *upd-3*:
assumes $m \leq N\ n \leq N\ j \leq n\ i \leq m$
shows *(DBM-upd M1 f m n N) i j = (DBM-upd M1 f i j N) i j*
⟨*proof*⟩

lemma *upd-id*:
assumes $m \leq N\ n \leq N\ i \leq m\ j \leq n$
shows *(DBM-upd M1 f m n N) i j = f i j*
⟨*proof*⟩

2.5.5 DBMs Without Negative Cycles are Non-Empty

We need all of these assumptions for the proof that matrices without negative cycles represent non-negative zones:

- Abelian (linearly ordered) monoid
- Time is non-trivial
- Time is dense

lemmas (**in** *linordered-ab-monoid-add*) *comm = add.commute*

lemma *sum-gt-neutral-dest'*:

$(a :: (('a :: time)\ DBMEntry)) \geq 0 \Longrightarrow a + b > 0 \Longrightarrow \exists\ d.\ Le\ d \leq a \wedge Le\ (-d) \leq b \wedge d \geq 0$
⟨*proof*⟩

lemma *sum-gt-neutral-dest*:
$(a :: (('a :: time)\ DBMEntry)) + b > 0 \Longrightarrow \exists\ d.\ Le\ d \leq a \wedge Le\ (-d) \leq b$
⟨*proof*⟩

2.5.6 Negative Cycles in DBMs

lemma *DBM-val-bounded-neg-cycle1*:
fixes *i xs* **assumes**
bounded: *DBM-val-bounded v u M n* **and** *A*:$i \leq n$ *set xs* $\subseteq \{0..n\}$ *len M i i xs* < 0 **and**
surj-on: $\forall\ k \leq n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k)$ **and** *at-most*: $i \neq 0$ *cnt 0 xs* ≤ 1
shows *False*
⟨*proof*⟩

lemma *cnt-0-I*:
$x \notin set\ xs \Longrightarrow cnt\ x\ xs = 0$
⟨*proof*⟩

lemma *distinct-cnt*: *distinct xs* $\Longrightarrow$ *cnt x xs* ≤ 1
⟨*proof*⟩

lemma *DBM-val-bounded-neg-cycle*:
fixes *i xs* **assumes**
bounded: *DBM-val-bounded v u M n* **and** *A*:$i \leq n$ *set xs* $\subseteq \{0..n\}$ *len M i i xs* < 0 **and**
surj-on: $\forall\ k \leq n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k)$
shows *False*
⟨*proof*⟩

Nicer Path Boundedness Theorems **lemma** *DBM-val-bounded-len-1*:
fixes *v*
assumes *DBM-val-bounded v u M n v c* $\leq n$ *set vs* $\subseteq \{0..n\}\ \forall\ k \leq n.\ (\exists\ c.\ v\ c = k)$
shows *dbm-entry-val u* (*Some c*) *None* (*len M* (*v c*) *0 vs*) ⟨*proof*⟩

lemma *DBM-val-bounded-len-2*:
fixes *v*
assumes *DBM-val-bounded v u M n v c* $\leq n$ *set vs* $\subseteq \{0..n\}\ \forall\ k \leq n.\ (\exists$

c. v c = k)
shows *dbm-entry-val u None* (*Some c*) (*len M 0* (*v c*) *vs*) ⟨*proof*⟩

lemma *DBM-val-bounded-len-3*:
fixes *v*
assumes *DBM-val-bounded v u M n v c1* ≤ *n v c2* ≤ *n set vs* ⊆ {*0..n*}
∀ *k* ≤ *n*. (∃ *c. v c = k*)
shows *dbm-entry-val u* (*Some c1*) (*Some c2*) (*len M* (*v c1*) (*v c2*) *vs*)
⟨*proof*⟩

An equivalent way of handling **0**

fun *val-0* :: ('*c* ⇒ ('*a* :: *linordered-ab-group-add*)) ⇒ '*c option* ⇒ '*a* **where**
val-0 u None = 0 |
val-0 u (*Some c*) = *u c*

notation *val-0* (-$_{\mathbf{0}}$ - [*90*,*90*] *90*)

lemma *dbm-entry-val-None-None*[*dest*]:
dbm-entry-val u None None l ⟹ *l* = ∞
⟨*proof*⟩

lemma *dbm-entry-val-dbm-lt*:
assumes *dbm-entry-val u x y l*
shows *Lt* ($u_{\mathbf{0}}$ *x* − $u_{\mathbf{0}}$ *y*) ≺ *l*
⟨*proof*⟩

lemma *dbm-lt-dbm-entry-val-1*:
assumes *Lt* (*u x*) ≺ *l*
shows *dbm-entry-val u* (*Some x*) *None l*
⟨*proof*⟩

lemma *dbm-lt-dbm-entry-val-2*:
assumes *Lt* (− *u x*) ≺ *l*
shows *dbm-entry-val u None* (*Some x*) *l*
⟨*proof*⟩

lemma *dbm-lt-dbm-entry-val-3*:
assumes *Lt* (*u x* − *u y*) ≺ *l*
shows *dbm-entry-val u* (*Some x*) (*Some y*) *l*
⟨*proof*⟩

A more uniform theorem for boundedness by paths

lemma *DBM-val-bounded-len*:
fixes *v*

defines $v' \equiv \lambda\ x.\ \mathit{if}\ x = \mathit{None}\ \mathit{then}\ 0\ \mathit{else}\ v\ (\mathit{the}\ x)$
assumes *DBM-val-bounded* $v\ u\ M\ n\ v'\ x \leq n\ v'\ y \leq n\ \mathit{set}\ vs \subseteq \{0..n\}$
$\forall\ k \leq n.\ (\exists\ c.\ v\ c = k)\ x \neq \mathit{None} \vee y \neq \mathit{None}$
shows $Lt\ (u_{\mathbf{0}}\ x - u_{\mathbf{0}}\ y) \prec \mathit{len}\ M\ (v'\ x)\ (v'\ y)\ vs\ \langle \mathit{proof} \rangle$

2.5.7 Floyd-Warshall Algorithm Preservers Zones

lemma *D-dest*: $x = D\ m\ i\ j\ k \Longrightarrow$
$x \in \{\mathit{len}\ m\ i\ j\ xs\ |xs.\ \mathit{set}\ xs \subseteq \{0..k\} \wedge i \notin \mathit{set}\ xs \wedge j \notin \mathit{set}\ xs \wedge \mathit{distinct}\ xs\}$
$\langle \mathit{proof} \rangle$

lemma *FW-zone-equiv*:
$\forall\ k \leq n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k) \Longrightarrow [M]_{v,n} = [FW\ M\ n]_{v,n}$
$\langle \mathit{proof} \rangle$

lemma *new-negative-cycle-aux'*:
fixes $M :: ('a :: \mathit{time})\ \mathit{DBM}$
fixes $i\ j\ d$
defines $M' \equiv \lambda\ i'\ j'.\ \mathit{if}\ (i' = i \wedge j' = j)\ \mathit{then}\ \mathit{Le}\ d$
$\mathit{else\ if}\ (i' = j \wedge j' = i)\ \mathit{then}\ \mathit{Le}\ (-d)$
$\mathit{else}\ M\ i'\ j'$
assumes $i \leq n\ j \leq n\ \mathit{set}\ xs \subseteq \{0..n\}\ \mathit{cycle\text{-}free}\ M\ n\ \mathit{length}\ xs = m$
assumes $\mathit{len}\ M'\ i\ i\ (j \# xs) < 0 \vee \mathit{len}\ M'\ j\ j\ (i \# xs) < 0$
assumes $i \neq j$
shows $\exists xs.\ \mathit{set}\ xs \subseteq \{0..n\} \wedge j \notin \mathit{set}\ xs \wedge i \notin \mathit{set}\ xs$
$\wedge\ (\mathit{len}\ M'\ i\ i\ (j \# xs) < 0 \vee \mathit{len}\ M'\ j\ j\ (i \# xs) < 0)\ \langle \mathit{proof} \rangle$

lemma *new-negative-cycle-aux*:
fixes $M :: ('a :: \mathit{time})\ \mathit{DBM}$
fixes $i\ d$
defines $M' \equiv \lambda\ i'\ j'.\ \mathit{if}\ (i' = i \wedge j' = 0)\ \mathit{then}\ \mathit{Le}\ d$
$\mathit{else\ if}\ (i' = 0 \wedge j' = i)\ \mathit{then}\ \mathit{Le}\ (-d)$
$\mathit{else}\ M\ i'\ j'$
assumes $i \leq n\ \mathit{set}\ xs \subseteq \{0..n\}\ \mathit{cycle\text{-}free}\ M\ n\ \mathit{length}\ xs = m$
assumes $\mathit{len}\ M'\ 0\ 0\ (i \# xs) < 0 \vee \mathit{len}\ M'\ i\ i\ (0 \# xs) < 0$
assumes $i \neq 0$
shows $\exists xs.\ \mathit{set}\ xs \subseteq \{0..n\} \wedge 0 \notin \mathit{set}\ xs \wedge i \notin \mathit{set}\ xs$
$\wedge\ (\mathit{len}\ M'\ 0\ 0\ (i \# xs) < 0 \vee \mathit{len}\ M'\ i\ i\ (0 \# xs) < 0)\ \langle \mathit{proof} \rangle$

2.6 The Characteristic Property of Canonical DBMs

theorem *fix-index'*:
fixes $M :: (('a :: \mathit{time})\ \mathit{DBMEntry})\ \mathit{mat}$

assumes $Le\ r \leq M\ i\ j\ Le\ (-r) \leq M\ j\ i\ cycle\text{-}free\ M\ n\ canonical\ M\ n\ i \leq n\ j \leq n\ i \neq j$
defines $M' \equiv \lambda\ i'\ j'.\ if\ (i' = i \wedge j' = j)\ then\ Le\ r$
$else\ if\ (i' = j \wedge j' = i)\ then\ Le\ (-r)$
$else\ M\ i'\ j'$
shows $(\forall\ u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M'\ n \longrightarrow DBM\text{-}val\text{-}bounded\ v\ u\ M\ n) \wedge cycle\text{-}free\ M'\ n$
⟨*proof*⟩

lemma *fix-index*:
fixes $M :: (('a :: time)\ DBMEntry)\ mat$
assumes $M\ 0\ i + M\ i\ 0 > 0\ cycle\text{-}free\ M\ n\ canonical\ M\ n\ i \leq n\ i \neq 0$
shows
$\exists\ (M' :: ('a\ DBMEntry)\ mat).\ ((\exists\ u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M'\ n) \longrightarrow (\exists\ u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M\ n))$
$\wedge\ M'\ 0\ i + M'\ i\ 0 = 0 \wedge cycle\text{-}free\ M'\ n$
$\wedge\ (\forall\ j.\ i \neq j \wedge M\ 0\ j + M\ j\ 0 = 0 \longrightarrow M'\ 0\ j + M'\ j\ 0 = 0)$
$\wedge\ (\forall\ j.\ i \neq j \wedge M\ 0\ j + M\ j\ 0 > 0 \longrightarrow M'\ 0\ j + M'\ j\ 0 > 0)$
⟨*proof*⟩

Putting it together **lemma** *FW-not-empty*:
$DBM\text{-}val\text{-}bounded\ v\ u\ (FW\ M'\ n)\ n \Longrightarrow DBM\text{-}val\text{-}bounded\ v\ u\ M'\ n$
⟨*proof*⟩

lemma *fix-indices*:
fixes $M :: (('a :: time)\ DBMEntry)\ mat$
assumes $set\ xs \subseteq \{0..n\}\ distinct\ xs$
assumes $cyc\text{-}free\ M\ n\ canonical\ M\ n$
shows
$\exists\ (M' :: ('a\ DBMEntry)\ mat).\ ((\exists\ u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M'\ n) \longrightarrow (\exists\ u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M\ n))$
$\wedge\ (\forall\ i \in set\ xs.\ i \neq 0 \longrightarrow M'\ 0\ i + M'\ i\ 0 = 0) \wedge cyc\text{-}free\ M'\ n$
$\wedge\ (\forall\ i{\leq}n.\ i \notin set\ xs \wedge M\ 0\ i + M\ i\ 0 = 0 \longrightarrow M'\ 0\ i + M'\ i\ 0 = 0)$
⟨*proof*⟩

lemma *cyc-free-obtains-valuation*:
$cyc\text{-}free\ M\ n \Longrightarrow \forall\ c.\ v\ c \leq n \longrightarrow v\ c > 0 \Longrightarrow \exists\ u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M\ n$
⟨*proof*⟩

2.6.1 Floyd-Warshall and Empty DBMs

theorem *FW-detects-empty-zone*:
$\forall\, k{\leq}n.\ 0 < k \longrightarrow (\exists\, c.\ v\ c = k) \Longrightarrow \forall\ c.\ v\ c \leq n \longrightarrow v\ c > 0$

$\Longrightarrow [FW\ M\ n]_{v,n} = \{\} \longleftrightarrow (\exists\ i{\leq}n.\ (FW\ M\ n)\ i\ i < Le\ 0)$
⟨*proof*⟩

hide-const (**open**) *D*

2.6.2 Mixed Corollaries

lemma *cyc-free-not-empty*:
assumes *cyc-free M n* ∀ *c. v c* ≤ *n* ⟶ *0* < *v c*
shows $[(M :: ('a :: time)\ DBM)]_{v,n} \neq \{\}$
⟨*proof*⟩

lemma *empty-not-cyc-free*:
assumes ∀ *c. v c* ≤ *n* ⟶ *0* < *v c* $[(M :: ('a :: time)\ DBM)]_{v,n} = \{\}$
shows ¬ *cyc-free M n*
⟨*proof*⟩

lemma *not-empty-cyc-free*:
assumes ∀ *k*≤*n. 0* < *k* ⟶ (∃ *c. v c* = *k*) $[(M :: ('a :: time)\ DBM)]_{v,n}$
≠ {}
shows *cyc-free M n* ⟨*proof*⟩

lemma *neg-cycle-empty*:
assumes ∀ *k*≤*n. 0* < *k* ⟶ (∃ *c. v c* = *k*) *set xs* ⊆ {*0..n*} *i* ≤ *n len M i*
i xs < *0*
shows $[(M :: ('a :: time)\ DBM)]_{v,n} = \{\}$ ⟨*proof*⟩

abbreviation *clock-numbering′* :: (*′c* ⇒ *nat*) ⇒ *nat* ⇒ *bool*
where
clock-numbering′ v n ≡ ∀ *c. v c* > *0* ∧ (∀ *x.* ∀ *y. v x* ≤ *n* ∧ *v y* ≤ *n* ∧ *v*
x = *v y* ⟶ *x* = *y*)

lemma *non-empty-dbm-diag-set*:
clock-numbering′ v n ⟹ $[M]_{v,n} \neq \{\}$
⟹ $[M]_{v,n} = [(\lambda\ i\ j.\ if\ i = j\ then\ 0\ else\ M\ i\ j)]_{v,n}$
⟨*proof*⟩

lemma *non-empty-cycle-free*:
assumes $[M]_{v,n} \neq \{\}$
and ∀ *k*≤*n. 0* < *k* ⟶ (∃ *c. v c* = *k*)
shows *cycle-free M n*
⟨*proof*⟩

lemma *neg-diag-empty*:
assumes $\forall\, k{\leq}n.\ 0 < k \longrightarrow (\exists\, c.\ v\ c = k)\ i \leq n\ M\ i\ i < 0$
shows $[M]_{v,n} = \{\}$
⟨*proof*⟩

lemma *canonical-empty-zone*:
assumes $\forall\, k{\leq}n.\ 0 < k \longrightarrow (\exists\, c.\ v\ c = k)\ \forall\, c.\ v\ c \leq n \longrightarrow 0 < v\ c$
and *canonical M n*
shows $[M]_{v,n} = \{\} \longleftrightarrow (\exists\, i{\leq}n.\ M\ i\ i < 0)$
⟨*proof*⟩

2.7 Orderings of DBMs

lemma *canonical-saturated-1*:
assumes $Le\ r \leq M\ (v\ c1)\ 0$
and $Le\ (-\ r) \leq M\ 0\ (v\ c1)$
and *cycle-free M n*
and *canonical M n*
and $v\ c1 \leq n$
and $v\ c1 > 0$
and $\forall\, c.\ v\ c \leq n \longrightarrow 0 < v\ c$
obtains u **where** $u \in [M]_{v,n}\ u\ c1 = r$
⟨*proof*⟩

lemma *canonical-saturated-2*:
assumes $Le\ r \leq M\ 0\ (v\ c2)$
and $Le\ (-\ r) \leq M\ (v\ c2)\ 0$
and *cycle-free M n*
and *canonical M n*
and $v\ c2 \leq n$
and $v\ c2 > 0$
and $\forall\, c.\ v\ c \leq n \longrightarrow 0 < v\ c$
obtains u **where** $u \in [M]_{v,n}\ u\ c2 = -\ r$
⟨*proof*⟩

lemma *canonical-saturated-3*:
assumes $Le\ r \leq M\ (v\ c1)\ (v\ c2)$
and $Le\ (-\ r) \leq M\ (v\ c2)\ (v\ c1)$
and *cycle-free M n*
and *canonical M n*
and $v\ c1 \leq n\ v\ c2 \leq n$
and $v\ c1 \neq v\ c2$
and $\forall\, c.\ v\ c \leq n \longrightarrow 0 < v\ c$
obtains u **where** $u \in [M]_{v,n}\ u\ c1 - u\ c2 = r$

⟨*proof*⟩

lemma *DBM-canonical-subset-le*:
notes *any-le-inf*[*intro*]
fixes M :: *real DBM*
assumes *canonical* $M\ n\ [M]_{v,n} \subseteq [M^{\uparrow}]_{v,n}\ [M]_{v,n} \neq \{\}\ i \leq n\ j \leq n\ i \neq j$
assumes *clock-numbering*: *clock-numbering'* $v\ n$
$\forall\, k{\leq}n.\ 0 < k \longrightarrow (\exists\, c.\ v\ c = k)$
shows $M\ i\ j \leq M'\ i\ j$
⟨*proof*⟩

end
theory *FW-More*
imports
DBM-Basics
Floyd-Warshall.FW-Code
begin

2.8 Partial Floyd-Warshall Preserves Zones

lemma *fwi-len-distinct*:
$\exists$ *ys. set ys* $\subseteq \{k\} \wedge$ *fwi* $m\ n\ k\ n\ n\ i\ j =$ *len* $m\ i\ j$ *ys* $\wedge\ i \notin$ *set ys* $\wedge\ j \notin$ *set ys* $\wedge$ *distinct ys*
if $i \leq n\ j \leq n\ k \leq n\ m\ k\ k \geq 0$
⟨*proof*⟩

lemma *FWI-mono*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow FWI\ M\ n\ k\ i\ j \leq M\ i\ j$
⟨*proof*⟩

lemma *FWI-zone-equiv*:
$[M]_{v,n} = [FWI\ M\ n\ k]_{v,n}$ **if** *surj-on*: $\forall\ k \leq n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k)$
and $k \leq n$
⟨*proof*⟩

end

3 DBM Operations

theory *DBM-Operations*
imports
DBM-Basics
begin

3.1 Auxiliary

lemmas [*trans*] = *finite-subset*

lemma *finite-vimageI2*: *finite* (*h* −‘ *F*) **if** *finite F inj-on h* {*x*. *h x* ∈ *F*}
⟨*proof*⟩

lemma *gt-swap*:
fixes *a b c* :: ′*t* :: *time*
assumes *c* < *a* + *b*
shows *c* < *b* + *a*
⟨*proof*⟩

lemma *le-swap*:
fixes *a b c* :: ′*t* :: *time*
assumes *c* ≤ *a* + *b*
shows *c* ≤ *b* + *a*
⟨*proof*⟩

abbreviation *clock-numbering* :: (′*c* ⇒ *nat*) ⇒ *bool*
where
clock-numbering v ≡ ∀ *c*. *v c* > *0*

lemma *DBM-triv*:
u ⊢*v*,*n* (λ*i j*. ∞)
⟨*proof*⟩

3.2 Relaxation

Relaxation of upper bound constraints on all variables. Used to compute time lapse in timed automata.

definition
up :: (′*t*::*linordered-cancel-ab-semigroup-add*) *DBM* ⇒ ′*t DBM*
where
up M ≡
λ *i j*. *if i* > *0 then if j* = *0 then* ∞ *else min* (*dbm-add* (*M i 0*) (*M 0 j*))
(*M i j*) *else M i j*

lemma *dbm-entry-dbm-lt*:
assumes *dbm-entry-val u* (*Some c1*) (*Some c2*) *a a* ≺ *b*
shows *dbm-entry-val u* (*Some c1*) (*Some c2*) *b*
⟨*proof*⟩

lemma *dbm-entry-dbm-min2*:

assumes *dbm-entry-val u None* (*Some c*) (*min a b*)
shows *dbm-entry-val u None* (*Some c*) *b*
⟨*proof*⟩

lemma *dbm-entry-dbm-min3*:
assumes *dbm-entry-val u* (*Some c*) *None* (*min a b*)
shows *dbm-entry-val u* (*Some c*) *None b*
⟨*proof*⟩

lemma *dbm-entry-dbm-min*:
assumes *dbm-entry-val u* (*Some c1*) (*Some c2*) (*min a b*)
shows *dbm-entry-val u* (*Some c1*) (*Some c2*) *b*
⟨*proof*⟩

lemma *dbm-entry-dbm-min3'*:
assumes *dbm-entry-val u* (*Some c*) *None* (*min a b*)
shows *dbm-entry-val u* (*Some c*) *None a*
⟨*proof*⟩

lemma *dbm-entry-dbm-min2'*:
assumes *dbm-entry-val u None* (*Some c*) (*min a b*)
shows *dbm-entry-val u None* (*Some c*) *a*
⟨*proof*⟩

lemma *dbm-entry-dbm-min'*:
assumes *dbm-entry-val u* (*Some c1*) (*Some c2*) (*min a b*)
shows *dbm-entry-val u* (*Some c1*) (*Some c2*) *a*
⟨*proof*⟩

lemma *DBM-up-complete'*: *clock-numbering v* $\Longrightarrow$ $u \in ([M]_{v,n})^{\uparrow} \Longrightarrow u \in [up\ M]_{v,n}$
⟨*proof*⟩

fun *theLe* :: (*'t*::*time*) *DBMEntry* $\Rightarrow$ *'t* **where**
theLe (*Le d*) = *d* |
theLe (*Lt d*) = *d* |
theLe ∞ = *0*

lemma *DBM-up-sound'*:
assumes *clock-numbering' v n u* $\in [up\ M]_{v,n}$
shows $u \in ([M]_{v,n})^{\uparrow}$
⟨*proof*⟩

3.3 Intersection

fun *And* :: ('*t* :: {*linordered-cancel-ab-monoid-add*}) *DBM* ⇒ '*t DBM* ⇒ '*t DBM* **where**
And M1 M2 = (λ *i j*. *min* (*M1 i j*) (*M2 i j*))

lemma *DBM-and-complete*:
assumes *DBM-val-bounded v u M1 n DBM-val-bounded v u M2 n*
shows *DBM-val-bounded v u* (*And M1 M2*) *n*
⟨*proof*⟩

lemma *DBM-and-sound1*:
assumes *DBM-val-bounded v u* (*And M1 M2*) *n*
shows *DBM-val-bounded v u M1 n*
⟨*proof*⟩

lemma *DBM-and-sound2*:
assumes *DBM-val-bounded v u* (*And M1 M2*) *n*
shows *DBM-val-bounded v u M2 n*
⟨*proof*⟩

lemma *And-correct*:
$[M1]_{v,n} \cap [M2]_{v,n} = [And\ M1\ M2]_{v,n}$
⟨*proof*⟩

3.4 Variable Reset

definition
DBM-reset :: ('*t* :: *time*) *DBM* ⇒ *nat* ⇒ *nat* ⇒ '*t* ⇒ '*t DBM* ⇒ *bool*
where
DBM-reset M n k d M' ≡
(∀ *j* ≤ *n*. 0 < *j* ∧ *k* ≠ *j* ⟶ *M' k j* = ∞ ∧ *M' j k* = ∞) ∧ *M' k 0* = *Le d* ∧ *M' 0 k* = *Le* (− *d*)
∧ *M' k k* = *M k k*
∧ (∀ *i* ≤ *n*. ∀ *j* ≤ *n*.
i ≠ *k* ∧ *j* ≠ *k* ⟶ *M' i j* = *min* (*dbm-add* (*M i k*) (*M k j*)) (*M i j*))

lemma *DBM-reset-mono*:
assumes *DBM-reset M n k d M' i* ≤ *n j* ≤ *n i* ≠ *k j* ≠ *k*
shows *M' i j* ≤ *M i j*
⟨*proof*⟩

lemma *DBM-reset-len-mono*:

assumes *DBM-reset M n k d M′ k* ∉ *set xs i* ≠ *k j* ≠ *k set* (*i* # *j* # *xs*) ⊆ {*0..n*}
shows *len M′ i j xs* ≤ *len M i j xs*
⟨*proof*⟩

lemma *DBM-reset-neg-cycle-preservation*:
assumes *DBM-reset M n k d M′ len M i i xs* < *Le 0 set* (*k* # *i* # *xs*) ⊆ {*0..n*}
shows ∃ *j*. ∃ *ys*. *set* (*j* # *ys*) ⊆ {*0..n*} ∧ *len M′ j j ys* < *Le 0*
⟨*proof*⟩

Implementation of DBM reset

definition
reset :: ('*t*::{*linordered-cancel-ab-semigroup-add*,*uminus*}) *DBM* ⇒ *nat* ⇒ *nat* ⇒ '*t* ⇒ '*t DBM*
where
reset M n k d =
(λ *i j*.
if i = *k* ∧ *j* = *0 then Le d*
else if i = *0* ∧ *j* = *k then Le* (−*d*)
else if i = *k* ∧ *j* ≠ *k then* ∞
else if i ≠ *k* ∧ *j* = *k then* ∞
else if i = *k* ∧ *j* = *k then M k k*
else min (*dbm-add* (*M i k*) (*M k j*)) (*M i j*)
)

fun
reset′ ::
('*t*::{*linordered-cancel-ab-semigroup-add*,*uminus*}) *DBM*
⇒ *nat* ⇒ '*c list* ⇒ ('*c* ⇒ *nat*) ⇒ '*t* ⇒ '*t DBM*
where
reset′ M n [] *v d* = *M* |
reset′ M n (*c* # *cs*) *v d* = *reset* (*reset′ M n cs v d*) *n* (*v c*) *d*

lemma *DBM-reset-reset*:
0 < *k* ⟹ *k* ≤ *n* ⟹ *DBM-reset M n k d* (*reset M n k d*)
⟨*proof*⟩

lemma *DBM-reset-complete*:
assumes *clock-numbering′ v n v c* ≤ *n DBM-reset M n* (*v c*) *d M′ DBM-val-bounded v u M n*
shows *DBM-val-bounded v* (*u*(*c* := *d*)) *M′ n*
⟨*proof*⟩

lemma *DBM-reset-sound-empty*:
assumes *clock-numbering′ v n v c* ≤ *n DBM-reset M n* (*v c*) *d M′*
∀ *u* . ¬ *DBM-val-bounded v u M′ n*
shows ¬ *DBM-val-bounded v u M n*
⟨*proof*⟩

lemma *DBM-reset-diag-preservation*:
∀ *k*≤*n*. *M′ k k* ≤ *0* **if** ∀ *k*≤*n*. *M k k* ≤ *0 DBM-reset M n i d M′*
⟨*proof*⟩

lemma *FW-diag-preservation*:
∀ *k*≤*n*. *M k k* ≤ *0* ⟹ ∀ *k*≤*n*. (*FW M n*) *k k* ≤ *0*
⟨*proof*⟩

lemma *DBM-reset-not-cyc-free-preservation*:
assumes ¬ *cyc-free M n DBM-reset M n k d M′ k* ≤ *n*
shows ¬ *cyc-free M′ n*
⟨*proof*⟩

lemma *DBM-reset-complete-empty′*:
assumes ∀ *k*≤*n*. *k* > *0* ⟶ (∃ *c*. *v c* = *k*) *clock-numbering v k* ≤ *n*
DBM-reset M n k d M′ ∀ *u* . ¬ *DBM-val-bounded v u M n*
shows ¬ *DBM-val-bounded v u M′ n*
⟨*proof*⟩

lemma *DBM-reset-complete-empty*:
assumes ∀ *k*≤*n*. *k* > *0* ⟶ (∃ *c*. *v c* = *k*) *clock-numbering v*
DBM-reset (*FW M n*) *n* (*v c*) *d M′* ∀ *u* . ¬ *DBM-val-bounded v u*
(*FW M n*) *n*
shows ¬ *DBM-val-bounded v u M′ n*
⟨*proof*⟩

lemma *DBM-reset-complete-empty1*:
assumes ∀ *k*≤*n*. *k* > *0* ⟶ (∃ *c*. *v c* = *k*) *clock-numbering v*
DBM-reset (*FW M n*) *n* (*v c*) *d M′* ∀ *u* . ¬ *DBM-val-bounded v u*
M n
shows ¬ *DBM-val-bounded v u M′ n*
⟨*proof*⟩

Lemma *FW-canonical-id* allows us to prove correspondences between reset and canonical, like for the two below. Can be left out for the rest because of the triviality of the correspondence.

lemma *DBM-reset-empty″*:
assumes ∀ *k*≤*n*. *k* > *0* ⟶ (∃ *c*. *v c* = *k*) *clock-numbering′ v n v c* ≤ *n*

DBM-reset M n (v c) d M′
shows $[M]_{v,n} = \{\} \longleftrightarrow [M']_{v,n} = \{\}$
⟨*proof*⟩

lemma *DBM-reset-empty*:
assumes $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering′ v n v c* $\leq n$
DBM-reset (FW M n) n (v c) d M′
shows $[FW\ M\ n]_{v,n} = \{\} \longleftrightarrow [M']_{v,n} = \{\}$
⟨*proof*⟩

lemma *DBM-reset-empty′*:
assumes *canonical M n* $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering′ v n v c* $\leq n$
DBM-reset (FW M n) n (v c) d M′
shows $[M]_{v,n} = \{\} \longleftrightarrow [M']_{v,n} = \{\}$
⟨*proof*⟩

lemma *DBM-reset-sound′*:
assumes *clock-numbering′ v n v c* $\leq n$ *DBM-reset M n (v c) d M′ DBM-val-bounded v u M′ n*
DBM-val-bounded v u″ M n
obtains d' **where** *DBM-val-bounded v (u(c := d′)) M n*
⟨*proof*⟩

lemma *DBM-reset-sound2*:
assumes *v c* $\leq n$ *DBM-reset M n (v c) d M′ DBM-val-bounded v u M′ n*
shows $u\ c = d$
⟨*proof*⟩

lemma *DBM-reset-sound″*:
fixes *M v c n d*
defines $M' \equiv$ *reset M n (v c) d*
assumes *clock-numbering′ v n v c* $\leq n$ *DBM-val-bounded v u M′ n*
DBM-val-bounded v u″ M n
obtains d' **where** *DBM-val-bounded v (u(c := d′)) M n*
⟨*proof*⟩

lemma *DBM-reset-sound*:
fixes *M v c n d*
defines $M' \equiv$ *reset M n (v c) d*
assumes $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering′ v n v c* $\leq n$
$u \in [M']_{v,n}$
obtains d' **where** $u(c := d') \in [M]_{v,n}$
⟨*proof*⟩

lemma *DBM-reset'-complete'*:
assumes *DBM-val-bounded v u M n clock-numbering' v n* ∀ *c* ∈ *set cs. v c* ≤ *n*
shows ∃ *u'. DBM-val-bounded v u' (reset' M n cs v d) n*
⟨*proof*⟩

lemma *DBM-reset'-complete*:
assumes *DBM-val-bounded v u M n clock-numbering' v n* ∀ *c* ∈ *set cs. v c* ≤ *n*
shows *DBM-val-bounded v* ([*cs* → *d*]*u*) (*reset' M n cs v d*) *n*
⟨*proof*⟩

lemma *DBM-reset'-sound-empty*:
assumes *clock-numbering' v n* ∀ *c* ∈ *set cs. v c* ≤ *n*
∀ *u* . ¬ *DBM-val-bounded v u* (*reset' M n cs v d*) *n*
shows ¬ *DBM-val-bounded v u M n*
⟨*proof*⟩

fun *set-clocks* :: *'c list* ⇒ *'t*::*time list*⇒ (*'c*,*'t*) *cval* ⇒ (*'c*,*'t*) *cval*
where
set-clocks [] - *u* = *u* |
set-clocks - [] *u* = *u* |
set-clocks (*c*#*cs*) (*t*#*ts*) *u* = (*set-clocks cs ts* (*u*(*c*:=*t*)))

lemma *DBM-reset'-sound'*:
fixes *M v c n d cs*
assumes *clock-numbering' v n* ∀ *c* ∈ *set cs. v c* ≤ *n*
DBM-val-bounded v u (*reset' M n cs v d*) *n DBM-val-bounded v u''* *M n*
shows ∃ *ts. DBM-val-bounded v* (*set-clocks cs ts u*) *M n*
⟨*proof*⟩

lemma *DBM-reset'-resets*:
fixes *M v c n d cs*
assumes ∀ *k*≤*n. k* > *0* ⟶ (∃ *c. v c* = *k*) *clock-numbering' v n* ∀ *c* ∈ *set cs. v c* ≤ *n*
DBM-val-bounded v u (*reset' M n cs v d*) *n*
shows ∀ *c* ∈ *set cs. u c* = *d*
⟨*proof*⟩

lemma *DBM-reset'-resets'*:
fixes *M* :: (*'t* :: *time*) *DBM* **and** *v c n d cs*
assumes *clock-numbering' v n* ∀ *c* ∈ *set cs. v c* ≤ *n DBM-val-bounded v*

u ($reset'$ M n cs v d) n
DBM-val-bounded v u'' M n
shows $\forall\, c \in set\ cs.\ u\ c = d$
⟨*proof*⟩

lemma *DBM-reset'-neg-diag-preservation'*:
fixes $M :: ('t :: time)\ DBM$
assumes $k{\leq}n\ M\ k\ k < 0$ *clock-numbering* $v \forall\ c \in set\ cs.\ v\ c \leq n$
shows $reset'\ M\ n\ cs\ v\ d\ k\ k < 0$ ⟨*proof*⟩

lemma *DBM-reset'-complete-empty'*:
assumes $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering'* $v\ n$
$\forall\ c \in set\ cs.\ v\ c \leq n \forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u\ M\ n$
shows $\forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u$ ($reset'\ M\ n\ cs\ v\ d$) n ⟨*proof*⟩

lemma *DBM-reset'-complete-empty*:
assumes $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering'* $v\ n$
$\forall\ c \in set\ cs.\ v\ c \leq n \forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u\ M\ n$
shows $\forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u$ ($reset'$ ($FW\ M\ n$) $n\ cs\ v\ d$) n ⟨*proof*⟩

lemma *DBM-reset'-empty'*:
assumes $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering'* $v\ n \forall\ c \in set\ cs.\ v\ c \leq n$
shows $[M]_{v,n} = \{\} \longleftrightarrow [reset'\ (FW\ M\ n)\ n\ cs\ v\ d]_{v,n} = \{\}$
⟨*proof*⟩

lemma *DBM-reset'-empty*:
assumes $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering'* $v\ n \forall\ c \in set\ cs.\ v\ c \leq n$
shows $[M]_{v,n} = \{\} \longleftrightarrow [reset'\ M\ n\ cs\ v\ d]_{v,n} = \{\}$
⟨*proof*⟩

lemma *DBM-reset'-sound*:
assumes $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering'* $v\ n$
and $\forall\, c{\in}set\ cs.\ v\ c \leq n$
and $u \in [reset'\ M\ n\ cs\ v\ d]_{v,n}$
shows $\exists\, ts.$ *set-clocks* $cs\ ts\ u \in [M]_{v,n}$
⟨*proof*⟩

3.5 Misc Preservation Lemmas

lemma *get-const-sum*[*simp*]:
$a \neq \infty \Longrightarrow b \neq \infty \Longrightarrow$ *get-const* $a \in \mathbb{Z} \Longrightarrow$ *get-const* $b \in \mathbb{Z} \Longrightarrow$ *get-const* $(a + b) \in \mathbb{Z}$

⟨*proof*⟩

lemma *sum-not-inf-dest*:
assumes $a + b \neq (\infty :: \text{-} DBMEntry)$
shows $a \neq (\infty :: \text{-} DBMEntry) \wedge b \neq (\infty :: \text{-} DBMEntry)$
⟨*proof*⟩

lemma *sum-not-inf-int*:
assumes $a + b \neq (\infty :: \text{-} DBMEntry)$ *get-const* $a \in \mathbb{Z}$ *get-const* $b \in \mathbb{Z}$
shows *get-const* $(a + b) \in \mathbb{Z}$
⟨*proof*⟩

lemma *int-fw-upd*:
$\forall\ i \leq n.\ \forall\ j \leq n.\ m\ i\ j \neq \infty \longrightarrow$ *get-const* $(m\ i\ j) \in \mathbb{Z} \Longrightarrow k \leq n \Longrightarrow i \leq n \Longrightarrow j \leq n$
$\Longrightarrow i' \leq n \Longrightarrow j' \leq n \Longrightarrow$ (*fw-upd* $m\ k\ i\ j\ i'\ j') \neq \infty$
$\Longrightarrow$ *get-const* (*fw-upd* $m\ k\ i\ j\ i'\ j') \in \mathbb{Z}$
⟨*proof*⟩

abbreviation *dbm-int* $M\ n \equiv \forall\ i{\leq}n.\ \forall\ j{\leq}n.\ M\ i\ j \neq \infty \longrightarrow$ *get-const* $(M\ i\ j) \in \mathbb{Z}$

abbreviation *dbm-int-all* $M \equiv \forall\ i.\ \forall\ j.\ M\ i\ j \neq \infty \longrightarrow$ *get-const* $(M\ i\ j) \in \mathbb{Z}$

lemma *dbm-intI*:
dbm-int-all $M \Longrightarrow$ *dbm-int* $M\ n$
⟨*proof*⟩

lemma *fwi-int-preservation*:
dbm-int (*fwi* $M\ n\ k\ i\ j$) n **if** *dbm-int* $M\ n\ k \leq n$
⟨*proof*⟩

lemma *fw-int-preservation*:
dbm-int (*fw* $M\ n\ k$) n **if** *dbm-int* $M\ n\ k \leq n$
⟨*proof*⟩

lemma *FW-int-preservation*:
assumes *dbm-int* $M\ n$
shows *dbm-int* (*FW* $M\ n$) n
⟨*proof*⟩

lemma *FW-int-all-preservation*:
assumes *dbm-int-all* M

shows *dbm-int-all* (*FW M n*)
⟨*proof*⟩

lemma *And-int-all-preservation*[*intro*]:
assumes *dbm-int-all M1 dbm-int-all M2*
shows *dbm-int-all* (*And M1 M2*)
⟨*proof*⟩

lemma *And-int-preservation*:
assumes *dbm-int M1 n dbm-int M2 n*
shows *dbm-int* (*And M1 M2*) *n*
⟨*proof*⟩

lemma *up-int-all-preservation*:
dbm-int-all (*M* :: (('*t* :: {*time*, *ring-1*}) *DBM*)) ⟹ *dbm-int-all* (*up M*)
⟨*proof*⟩

lemma *up-int-preservation*:
dbm-int (*M* :: (('*t* :: {*time*, *ring-1*}) *DBM*)) *n* ⟹ *dbm-int* (*up M*) *n*
⟨*proof*⟩

lemma *DBM-reset-int-preservation*′:
assumes *dbm-int M n DBM-reset M n k d M*′ *d* ∈ ℤ *k* ≤ *n*
shows *dbm-int M*′ *n*
⟨*proof*⟩

lemma *DBM-reset-int-preservation*:
fixes *M* :: ('*t* :: {*time*,*ring-1*}) *DBM*
assumes *dbm-int M n d* ∈ ℤ *0* < *k k* ≤ *n*
shows *dbm-int* (*reset M n k d*) *n*
⟨*proof*⟩

lemma *DBM-reset-int-all-preservation*:
fixes *M* :: ('*t* :: {*time*,*ring-1*}) *DBM*
assumes *dbm-int-all M d* ∈ ℤ
shows *dbm-int-all* (*reset M n k d*)
⟨*proof*⟩

lemma *DBM-reset*′*-int-all-preservation*:
fixes *M* :: ('*t* :: {*time*, *ring-1*}) *DBM*
assumes *dbm-int-all M d* ∈ ℤ
shows *dbm-int-all* (*reset*′ *M n cs v d*) ⟨*proof*⟩

lemma *DBM-reset'-int-preservation*:
fixes $M :: ('t :: \{time, ring\text{-}1\})\ DBM$
assumes *dbm-int* $M\ n\ d \in \mathbb{Z} \forall\, c.\ v\ c > 0 \forall\ c \in set\ cs.\ v\ c \leq n$
shows *dbm-int* $(reset'\ M\ n\ cs\ v\ d)\ n$ ⟨*proof*⟩

lemma *reset-set1*:
$\forall\, c \in set\ cs.\ ([cs \rightarrow d]u)\ c = d$
⟨*proof*⟩

lemma *reset-set11*:
$\forall\, c.\ c \notin set\ cs \longrightarrow ([cs \rightarrow d]u)\ c = u\ c$
⟨*proof*⟩

lemma *reset-set2*:
$\forall\, c.\ c \notin set\ cs \longrightarrow (set\text{-}clocks\ cs\ ts\ u)c = u\ c$
⟨*proof*⟩

lemma *reset-set*:
assumes $\forall\ c \in set\ cs.\ u\ c = d$
shows $[cs \rightarrow d](set\text{-}clocks\ cs\ ts\ u) = u$
⟨*proof*⟩

3.5.1 Unused theorems

lemma *canonical-cyc-free*:
$canonical\ M\ n \Longrightarrow \forall\, i \leq n.\ M\ i\ i \geq 0 \Longrightarrow cyc\text{-}free\ M\ n$
⟨*proof*⟩

lemma *canonical-cyc-free2*:
$canonical\ M\ n \Longrightarrow cyc\text{-}free\ M\ n \longleftrightarrow (\forall\, i \leq n.\ M\ i\ i \geq 0)$
⟨*proof*⟩

lemma *DBM-reset'-diag-preservation*:
fixes $M :: ('t :: time)\ DBM$
assumes $\forall\, k{\leq}n.\ M\ k\ k \leq 0\ clock\text{-}numbering\ v \forall\ c \in set\ cs.\ v\ c \leq n$
shows $\forall\, k{\leq}n.\ reset'\ M\ n\ cs\ v\ d\ k\ k \leq 0$ ⟨*proof*⟩

end
theory *DBM-Misc*
imports
Main
HOL.Real
begin

lemma *finite-set-of-finite-funs2*:
fixes $A :: 'a\ set$
and $B :: 'b\ set$
and $C :: 'c\ set$
and $d :: 'c$
assumes *finite* A
and *finite* B
and *finite* C
shows *finite* $\{f.\ \forall x.\ \forall y.\ (x \in A \wedge y \in B \longrightarrow f\ x\ y \in C) \wedge (x \notin A \longrightarrow f\ x\ y = d) \wedge (y \notin B \longrightarrow f\ x\ y = d)\}$
⟨*proof*⟩

end

3.6 Extrapolation of DBMs

theory *DBM-Normalization*
imports
DBM-Basics
DBM-Misc
HOL−Eisbach.Eisbach
begin

NB: The journal paper on extrapolations based on lower and upper bounds [1] provides slightly incorrect definitions that would always set (lower) bounds of the form $M\ 0\ i$ to ∞. To fix this, we use two invariants that can also be found in TChecker's DBM library, for instance:

1. Lower bounds are always nonnegative, i.e. $\forall i \leq n.\ M\ 0\ i \leq 0$ (see *extra-lup-lower-bounds*).
2. Entries to the diagonal is always normalized to *Le 0*, *Lt 0* or ∞. This makes it again obvious that the set of normalized DBMs is finite.

lemmas *dbm-less-simps*[*simp*] = *dbm-lt-code-simps*[*folded DBM.less*]

lemma *dbm-less-eq-simps*[*simp*]:
$Le\ a \leq Le\ b \longleftrightarrow a \leq b$
$Le\ a \leq Lt\ b \longleftrightarrow a < b$
$Lt\ a \leq Le\ b \longleftrightarrow a \leq b$
$Lt\ a \leq Lt\ b \longleftrightarrow a \leq b$
⟨*proof*⟩

lemma *Le-less-Lt*[*simp*]: $Le\ x < Lt\ x \longleftrightarrow False$
⟨*proof*⟩

3.6.1 Classical extrapolation

This is the implementation of the classical extrapolation operator ($Extra_M$).

fun *norm-upper* :: ('*t*::*linorder*) *DBMEntry* ⇒ '*t* ⇒ '*t DBMEntry*
where
norm-upper e t = (*if Le t* ≺ *e then* ∞ *else e*)

fun *norm-lower* :: ('*t*::*linorder*) *DBMEntry* ⇒ '*t* ⇒ '*t DBMEntry*
where
norm-lower e t = (*if e* ≺ *Lt t then Lt t else e*)

definition
norm-diag e = (*if e* ≺ *Le 0 then Lt 0 else if e* = *Le 0 then e else* ∞)

Note that literature pretends that **0** would have a bound of negative infinity in k and thus defines normalization uniformly. The easiest way to get around this seems to explicate this in the definition as below.

definition *norm* :: ('*t* :: *linordered-ab-group-add*) *DBM* ⇒ (*nat* ⇒ '*t*) ⇒ *nat* ⇒ '*t DBM*
where
norm M k n ≡ λ*i j*.
let ub = *if i* > *0 then k i else 0 in*
let lb = *if j* > *0 then* − *k j else 0 in*
if i ≤ *n* ∧ *j* ≤ *n then*
if i ≠ *j then norm-lower* (*norm-upper* (*M i j*) *ub*) *lb else norm-diag* (*M i j*)
else M i j

3.6.2 Extrapolations based on lower and upper bounds

This is the implementation of the LU-bounds based extrapolation operation (*Extra*-{LU}).

definition *extra-lu* ::
('*t* :: *linordered-ab-group-add*) *DBM* ⇒ (*nat* ⇒ '*t*) ⇒ (*nat* ⇒ '*t*) ⇒ *nat* ⇒ '*t DBM*
where
extra-lu M l u n ≡ λ*i j*.
let ub = *if i* > *0 then l i else 0 in*
let lb = *if j* > *0 then* − *u j else 0 in*
if i ≤ *n* ∧ *j* ≤ *n then*
if i ≠ *j then norm-lower* (*norm-upper* (*M i j*) *ub*) *lb else norm-diag* (*M i j*)

else M i j

lemma *norm-is-extra*:
norm M k n = extra-lu M k k n
⟨*proof*⟩

This is the implementation of the LU-bounds based extrapolation operation (*Extra-*$\{LU\}^\frown+$).

definition *extra-lup* ::
(′*t* :: *linordered-ab-group-add*) *DBM* ⇒ (*nat* ⇒ ′*t*) ⇒ (*nat* ⇒ ′*t*) ⇒ *nat* ⇒ ′*t DBM*
where
extra-lup M l u n ≡ λ*i j*.
let ub = *if i* > *0 then Lt* (*l i*) *else Le 0*;
lb = *if j* > *0 then Lt* (− *u j*) *else Lt 0*
in
if i ≤ *n* ∧ *j* ≤ *n then*
if i ≠ *j then*
if ub ≺ *M i j then* ∞
else if i > *0* ∧ *M 0 i* ≺ *Lt* (− *l i*) *then* ∞
else if i > *0* ∧ *M 0 j* ≺ *lb then* ∞
else if i = *0* ∧ *M 0 j* ≺ *lb then Lt* (− *u j*)
else M i j
else norm-diag (*M i j*)
else M i j

method *csimp* = (*clarsimp simp*: *extra-lup-def Let-def DBM.less*[*symmetric*] *not-less any-le-inf neutral*)

method *solve* = *csimp?*; *safe?*; (*csimp* | *meson Lt-le-LeI le-less le-less-trans less-asym*′); *fail*

lemma extrapolations-diag-preservation: extra-lu M L U n i i = M i i
extra-lup M L U n i i = M i i norm M k n i i = M i i unfolding extra-lu-def extra-lup-def norm-def Let-def by auto

lemma
assumes ∀ *i* ≤ *n*. *i* > *0* ⟶ *M 0 i* ≤ *0* ∀ *i* ≤ *n*. *U i* ≥ *0*
shows
extra-lu-lower-bounds: ∀ *i* ≤ *n*. *i* > *0* ⟶ *extra-lu M L U n 0 i* ≤ *0*
and
norm-lower-bounds: ∀ *i* ≤ *n*. *i* > *0* ⟶ *norm M U n 0 i* ≤ *0* **and**

extra-lup-lower-bounds: $\forall\, i \leq n.\ i > 0 \longrightarrow \mathit{extra\text{-}lup}\ M\ L\ U\ n\ 0\ i \leq 0$
⟨*proof*⟩

lemma *extra-lu-le-extra-lup*:
assumes *canonical*: *canonical* $M\ n$
and *canonical-lower-bounds*: $\forall\, i \leq n.\ i > 0 \longrightarrow M\ 0\ i \leq 0$
shows $\mathit{extra\text{-}lu}\ M\ l\ u\ n\ i\ j \leq \mathit{extra\text{-}lup}\ M\ l\ u\ n\ i\ j$
⟨*proof*⟩

lemma *extra-lu-subs-extra-lup*:
assumes *canonical*: *canonical* $M\ n$ **and** *canonical-lower-bounds*: $\forall\, i \leq n.\ i > 0 \longrightarrow M\ 0\ i \leq 0$
shows $[\mathit{extra\text{-}lu}\ M\ L\ U\ n]_{v,n} \subseteq [\mathit{extra\text{-}lup}\ M\ L\ U\ n]_{v,n}$
⟨*proof*⟩

3.6.3 Extrapolations are widening operators

lemma *extra-lu-mono*:
assumes $\forall\, c.\ v\ c > 0\ u \in [M]_{v,n}$
shows $u \in [\mathit{extra\text{-}lu}\ M\ L\ U\ n]_{v,n}$ (**is** $u \in [?M2]_{v,n}$)
⟨*proof*⟩

lemma *norm-mono*:
assumes $\forall\, c.\ v\ c > 0\ u \in [M]_{v,n}$
shows $u \in [\mathit{norm}\ M\ k\ n]_{v,n}$
⟨*proof*⟩

3.6.4 Finiteness of extrapolations

abbreviation $\mathit{dbm\text{-}default}\ M\ n \equiv (\forall\ i > n.\ \forall\ j.\ M\ i\ j = 0) \wedge (\forall\ j > n.\ \forall\ i.\ M\ i\ j = 0)$

lemma *norm-default-preservation*:
$\mathit{dbm\text{-}default}\ M\ n \Longrightarrow \mathit{dbm\text{-}default}\ (\mathit{norm}\ M\ k\ n)\ n$
⟨*proof*⟩

lemma *extra-lu-default-preservation*:
$\mathit{dbm\text{-}default}\ M\ n \Longrightarrow \mathit{dbm\text{-}default}\ (\mathit{extra\text{-}lu}\ M\ L\ U\ n)\ n$
⟨*proof*⟩

instance *int* :: *linordered-cancel-ab-monoid-add* ⟨*proof*⟩

lemmas *finite-subset-rev*[*intro?*] = *finite-subset*[*rotated*]
lemmas [*intro?*] = *finite-subset*

lemma *extra-lu-finite*:
fixes $L\ U :: nat \Rightarrow nat$
shows *finite* $\{extra\text{-}lu\ M\ L\ U\ n \mid M.\ dbm\text{-}default\ M\ n\}$
⟨*proof*⟩

lemma *normalized-integral-dbms-finite*:
finite $\{norm\ M\ (k :: nat \Rightarrow nat)\ n \mid M.\ dbm\text{-}default\ M\ n\}$
⟨*proof*⟩

end

4 DBMs as Constraint Systems

theory *DBM-Constraint-Systems*
imports
DBM-Operations
DBM-Normalization
begin

4.1 Misc

lemma *Max-le-MinI*:
assumes *finite* S *finite* T $S \neq \{\}$ $T \neq \{\}$ $\bigwedge x\ y.\ x \in S \Longrightarrow y \in T \Longrightarrow x \leq y$
shows $Max\ S \leq Min\ T$ ⟨*proof*⟩

lemma *Min-insert-cases*:
assumes $x = Min\ (insert\ a\ S)$ *finite* S
obtains (*default*) $x = a \mid$ (*elem*) $x \in S$
⟨*proof*⟩

lemma *cval-add-simp*[*simp*]:
$(u \oplus d)\ x = u\ x + d$
⟨*proof*⟩

lemmas [*simp*] = *any-le-inf*

lemma *Le-in-between*:
assumes $a < b$
obtains d **where** $a \leq Le\ d\ Le\ d \leq b$
⟨*proof*⟩

lemma *DBMEntry-le-to-sum*:

fixes $e\ e' :: 't :: time\ DBMEntry$
assumes $e' \neq \infty\ e \leq e'$
shows $-\ e' + e \leq 0$
⟨*proof*⟩

lemma *DBMEntry-le-add*:
fixes $a\ b\ c :: 't :: time\ DBMEntry$
assumes $a \leq b + c\ c \neq \infty$
shows $-c + a \leq b$
⟨*proof*⟩

lemma *DBM-triv-emptyI*:
assumes $M\ 0\ 0 < 0$
shows $[M]_{v,n} = \{\}$
⟨*proof*⟩

4.2 Definition and Semantics of Constraint Systems

datatype $('x, 'v)\ constr =$
Lower $'x\ 'v\ DBMEntry \mid Upper\ 'x\ 'v\ DBMEntry \mid Diff\ 'x\ 'x\ 'v\ DBMEntry$

type-synonym $('x, 'v)\ cs = ('x, 'v)\ constr\ set$

inductive *entry-sem* $(- \models_e - [62, 62]\ 62)$ **where**
$v \models_e Lt\ x$ **if** $v < x \mid$
$v \models_e Le\ x$ **if** $v \leq x \mid$
$v \models_e \infty$

inductive *constr-sem* $(- \models_c - [62, 62]\ 62)$ **where**
$u \models_c Lower\ x\ e$ **if** $-\ u\ x \models_e e \mid$
$u \models_c Upper\ x\ e$ **if** $u\ x \models_e e \mid$
$u \models_c Diff\ x\ y\ e$ **if** $u\ x - u\ y \models_e e$

definition *cs-sem* $(- \models_{cs} - [62, 62]\ 62)$ **where**
$u \models_{cs} cs \longleftrightarrow (\forall\, c \in cs.\ u \models_c c)$

definition *cs-models* $(- \models - [62, 62]\ 62)$ **where**
$cs \models c \equiv \forall\, u.\ u \models_{cs} cs \longrightarrow u \models_c c$

definition *cs-equiv* $(- \equiv_{cs} - [62, 62]\ 62)$ **where**
$cs \equiv_{cs} cs' \equiv \forall\, u.\ u \models_{cs} cs \longleftrightarrow u \models_{cs} cs'$

definition
$closure\ cs \equiv \{c.\ cs \models c\}$

definition
$bot\text{-}cs = \{Lower\ undefined\ (Lt\ 0),\ Upper\ undefined\ (Lt\ 0)\}$

lemma *constr-sem-less-eq-iff*:
$u \models_c Lower\ x\ e \longleftrightarrow Le\ (-u\ x) \leq e$
$u \models_c Upper\ x\ e \longleftrightarrow Le\ (u\ x) \leq e$
$u \models_c Diff\ x\ y\ e \longleftrightarrow Le\ (u\ x - u\ y) \leq e$
⟨*proof*⟩

lemma *constr-sem-mono*:
assumes $e \leq e'$
shows
$u \models_c Lower\ x\ e \implies u \models_c Lower\ x\ e'$
$u \models_c Upper\ x\ e \implies u \models_c Upper\ x\ e'$
$u \models_c Diff\ x\ y\ e \implies u \models_c Diff\ x\ y\ e'$
⟨*proof*⟩

lemma *constr-sem-triv*[*simp*,*intro*]:
$u \models_c Upper\ x\ \infty\ u \models_c Lower\ y\ \infty\ u \models_c Diff\ x\ y\ \infty$
⟨*proof*⟩

lemma *cs-sem-antimono*:
assumes $cs \subseteq cs'\ u \models_{cs} cs'$
shows $u \models_{cs} cs$
⟨*proof*⟩

lemma *cs-equivD*[*intro*, *dest*]:
assumes $u \models_{cs} cs\ cs \equiv_{cs} cs'$
shows $u \models_{cs} cs'$
⟨*proof*⟩

lemma *cs-equiv-sym*:
$cs \equiv_{cs} cs'$ **if** $cs' \equiv_{cs} cs$
⟨*proof*⟩

lemma *cs-equiv-union*:
$cs \equiv_{cs} cs \cup cs'$ **if** $cs \equiv_{cs} cs'$
⟨*proof*⟩

lemma *cs-equiv-alt-def*:
$cs \equiv_{cs} cs' \longleftrightarrow (\forall\, c.\ cs \models c \longleftrightarrow cs' \models c)$
⟨*proof*⟩

lemma *closure-equiv*:
closure cs $\equiv_{cs}$ *cs*
⟨*proof*⟩

lemma *closure-superset*:
cs $\subseteq$ *closure cs*
⟨*proof*⟩

lemma *bot-cs-empty*:
$\neg$ (*u* :: ('*c* $\Rightarrow$ '*t* :: *linordered-ab-group-add*)) $\models_{cs}$ *bot-cs*
⟨*proof*⟩

lemma *finite-bot-cs*:
finite bot-cs
⟨*proof*⟩

definition *cs-vars* **where**
cs-vars cs = $\bigcup$ (*set1-constr* ‘ *cs*)

definition *map-cs-vars* **where**
map-cs-vars v cs = *map-constr v id* ‘ *cs*

lemma *constr-sem-rename-vars*:
assumes *inj-on v S set1-constr c* $\subseteq$ *S*
shows (*u o inv-into S v*) $\models_c$ *map-constr v id c* $\longleftrightarrow$ *u* $\models_c$ *c*
⟨*proof*⟩

lemma *cs-sem-rename-vars*:
assumes *inj-on v* (*cs-vars cs*)
shows (*u o inv-into* (*cs-vars cs*) *v*) $\models_{cs}$ *map-cs-vars v cs* $\longleftrightarrow$ *u* $\models_{cs}$ *cs*
⟨*proof*⟩

4.3 Conversion of DBMs to Constraint Systems and Back

definition *dbm-to-cs* :: *nat* $\Rightarrow$ ('*x* $\Rightarrow$ *nat*) $\Rightarrow$ ('*v* :: {*linorder*,*zero*}) *DBM*
$\Rightarrow$ ('*x*, '*v*) *cs* **where**
dbm-to-cs n v M $\equiv$ *if M 0 0* $<$ *0 then bot-cs else*
{*Lower x* (*M 0* (*v x*)) | *x*. *v x* $\leq$ *n*} $\cup$
{*Upper x* (*M* (*v x*) *0*) | *x*. *v x* $\leq$ *n*} $\cup$
{*Diff x y* (*M* (*v x*) (*v y*)) | *x y*. *v x* $\leq$ *n* $\wedge$ *v y* $\leq$ *n*}

lemma *dbm-entry-val-Lower-iff*:
dbm-entry-val u None (*Some x*) *e* $\longleftrightarrow$ *u* $\models_c$ *Lower x e*
⟨*proof*⟩

lemma *dbm-entry-val-Upper-iff*:
dbm-entry-val u (*Some x*) *None e* $\longleftrightarrow$ *u* $\models_c$ *Upper x e*
⟨*proof*⟩

lemma *dbm-entry-val-Diff-iff*:
dbm-entry-val u (*Some x*) (*Some y*) *e* $\longleftrightarrow$ *u* $\models_c$ *Diff x y e*
⟨*proof*⟩

lemmas *dbm-entry-val-constr-sem-iff* =
dbm-entry-val-Lower-iff
dbm-entry-val-Upper-iff
dbm-entry-val-Diff-iff

theorem *dbm-to-cs-correct*:
u $\vdash_{v,n}$ *M* $\longleftrightarrow$ *u* $\models_{cs}$ *dbm-to-cs n v M*
⟨*proof*⟩

definition
cs-to-dbm v cs ≡ *if* (∀ *u*. ¬*u* $\models_{cs}$ *cs*) *then* (λ- -. *Lt 0*) *else* (
λ*i j*.
if i = *0 then*
if j = *0 then*
Le 0
else
Min (*insert* ∞ {*e*. ∃ *x*. *Lower x e* ∈ *cs* ∧ *v x* = *j*})
else
if j = *0 then*
Min (*insert* ∞ {*e*. ∃ *x*. *Upper x e* ∈ *cs* ∧ *v x* = *i*})
else
Min (*insert* ∞ {*e*. ∃ *x y*. *Diff x y e* ∈ *cs* ∧ *v x* = *i* ∧ *v y* = *j*})
)

lemma *finite-dbm-to-cs*:
assumes *finite* {*x*. *v x* ≤ *n*}
shows *finite* (*dbm-to-cs n v M*)
⟨*proof*⟩

lemma *empty-dbm-empty*:
u $\vdash_{v,n}$ (λ- -. *Lt 0*) $\longleftrightarrow$ *False*
⟨*proof*⟩

fun *expr-of-constr* **where**
expr-of-constr (*Lower* - *e*) = *e* |

expr-of-constr (*Upper* - *e*) = *e* |
expr-of-constr (*Diff* - - *e*) = *e*

lemma *cs-to-dbm1*:
assumes $\forall\, x \in$ *cs-vars cs*. *v x* $>$ *0* $\wedge$ *v x* $\leq$ *n finite cs*
assumes $u \vdash_{v,n}$ *cs-to-dbm v cs*
shows $u \models_{cs}$ *cs*
⟨*proof*⟩

lemma *cs-to-dbm2*:
assumes $\forall\, x$. *v x* $\leq$ *n* $\longrightarrow$ *v x* $>$ *0* $\forall\, x\ y$. *v x* $\leq$ *n* $\wedge$ *v y* $\leq$ *n* $\wedge$ *v x* = *v y* $\longrightarrow$ *x* = *y*
assumes *finite cs*
assumes $u \models_{cs}$ *cs*
shows $u \vdash_{v,n}$ *cs-to-dbm v cs*
⟨*proof*⟩

theorem *cs-to-dbm-correct*:
assumes $\forall\, x \in$ *cs-vars cs*. *v x* $\leq$ *n* $\forall\, x$. *v x* $\leq$ *n* $\longrightarrow$ *v x* $>$ *0*
$\forall\, x\ y$. *v x* $\leq$ *n* $\wedge$ *v y* $\leq$ *n* $\wedge$ *v x* = *v y* $\longrightarrow$ *x* = *y*
finite cs
shows $u \vdash_{v,n}$ *cs-to-dbm v cs* $\longleftrightarrow$ $u \models_{cs}$ *cs*
⟨*proof*⟩

corollary *cs-to-dbm-correct′*:
assumes
bij-betw v (*cs-vars cs*) {*1*..*n*} $\forall\, x$. *v x* $\leq$ *n* $\longrightarrow$ *v x* $>$ *0* $\forall\, x$. *x* $\notin$ *cs-vars cs* $\longrightarrow$ *v x* $>$ *n*
finite cs
shows $u \vdash_{v,n}$ *cs-to-dbm v cs* $\longleftrightarrow$ $u \models_{cs}$ *cs*
⟨*proof*⟩

4.4 Application: Relaxation On Constraint Systems

The following is a sample application of viewing DBMs as constraint systems. We show define an equivalent of the *up* operation on DBMs, prove it correct, and then derive an alternative correctness proof for *up*.

definition
up-cs cs = {*c*. *c* $\in$ *cs* $\wedge$ (*case c of Upper* - - $\Rightarrow$ *False* | - $\Rightarrow$ *True*)}

lemma *Lower-shiftI*:
u $\oplus$ *d* $\models_c$ *Lower x e* **if** *u* $\models_c$ *Lower x e* (*d* :: *′t* :: *linordered-ab-group-add*) $\geq$ *0*

⟨*proof*⟩

lemma *Upper-shiftI*:
$u \oplus d \models_c \mathit{Upper}\ x\ e$ **if** $u \models_c \mathit{Upper}\ x\ e\ (d :: 't :: \mathit{linordered\text{-}ab\text{-}group\text{-}add}) \leq 0$
⟨*proof*⟩

lemma *Diff-shift*:
$u \oplus d \models_c \mathit{Diff}\ x\ y\ e \longleftrightarrow u \models_c \mathit{Diff}\ x\ y\ e$ **for** $d :: 't :: \mathit{linordered\text{-}ab\text{-}group\text{-}add}$
⟨*proof*⟩

lemma *up-cs-complete*:
$u \oplus d \models_{cs} \mathit{up\text{-}cs}\ cs$ **if** $u \models_{cs} cs\ d \geq 0$ **for** $d :: 't :: \mathit{linordered\text{-}ab\text{-}group\text{-}add}$
⟨*proof*⟩

definition
lower-upper-closed $cs \equiv \forall\, x\ y\ e\ e'.$
$\mathit{Lower}\ x\ e \in cs \wedge \mathit{Upper}\ y\ e' \in cs \longrightarrow (\exists\ e''.\ \mathit{Diff}\ y\ x\ e'' \in cs \wedge e'' \leq e + e')$

lemma *up-cs-sound*:
assumes $u \models_{cs} \mathit{up\text{-}cs}\ cs\ \mathit{lower\text{-}upper\text{-}closed}\ cs\ \mathit{finite}\ cs$
obtains u' **and** $d :: 't :: \mathit{time}$ **where** $d \geq 0\ u' \models_{cs} cs\ u = u' \oplus d$
⟨*proof*⟩

Note that if we compare this proof to $[\![\forall\, c.\ 0 < ?v\ c \wedge (\forall\, x\ y.\ ?v\ x \leq ?n \wedge ?v\ y \leq ?n \wedge ?v\ x = ?v\ y \longrightarrow x = y);\ ?u \in [up\ ?M]_{?v,?n}]\!] \Longrightarrow ?u \in [?M]_{?v,?n}{}^{\uparrow}$, we can see that we have not gained much. Settling on DBM entry arithmetic as done above was not the optimal choice for this proof, while it can drastically simply some other proofs. Also, note that the final theorem we obtain below (*DBM-up-correct*) is slightly stronger than what we would get with $[\![\forall\, c.\ 0 < ?v\ c \wedge (\forall\, x\ y.\ ?v\ x \leq ?n \wedge ?v\ y \leq ?n \wedge ?v\ x = ?v\ y \longrightarrow x = y);\ ?u \in [up\ ?M]_{?v,?n}]\!] \Longrightarrow ?u \in [?M]_{?v,?n}{}^{\uparrow}$. Finally, note that a more elegant definition of *lower-upper-closed* would probably be: *definition lower-upper-closed* $cs \equiv \forall\, x\ y\ e\ e'.\ cs \models \mathit{Lower}\ x\ e \wedge cs \models \mathit{Upper}\ y\ e' \longrightarrow (\exists\ e''.\ cs \models \mathit{Diff}\ y\ x\ e'' \wedge e'' \leq e + e')$ This would mean that in the proof we would have to replace minimum and maximum by supremum and infimum. The advantage would be that the finiteness assumption could be removed. However, as our DBM entries do not come with $-\infty$, they do not form a complete lattice. Thus we would either have to make this extension or directly refer to the embedded values directly, which would again have to form a complete lattice. Both variants come with some technical inconvenience.

lemma *up-cs-sem*:
fixes *cs* :: (*'x*, *'v* :: *time*) *cs*
assumes *lower-upper-closed cs finite cs*
shows $\{u.\ u \models_{cs} \textit{up-cs}\ cs\} = \{u \oplus d \mid u\ d.\ u \models_{cs} cs \wedge d \geq 0\}$
⟨*proof*⟩

definition
close-lu :: (*'t*::*linordered-cancel-ab-semigroup-add*) *DBM* ⇒ *'t DBM*
where
close-lu M ≡ λ*i j*. *if i* > *0 then min* (*dbm-add* (*M i 0*) (*M 0 j*)) (*M i j*)
else M i j

definition
up' :: (*'t*::*linordered-cancel-ab-semigroup-add*) *DBM* ⇒ *'t DBM*
where
up' M ≡ λ*i j*. *if i* > *0* ∧ *j* = *0 then* ∞ *else M i j*

lemma *up-alt-def*:
up M = *up'* (*close-lu M*)
⟨*proof*⟩

lemma *close-lu-equiv*:
fixes *M* :: *'t* ::*time DBM*
shows *dbm-to-cs n v M* ≡$_{cs}$ *dbm-to-cs n v* (*close-lu M*)
⟨*proof*⟩

lemma *close-lu-closed*:
lower-upper-closed (*dbm-to-cs n v* (*close-lu M*)) **if** *M 0 0* ≥ *0*
⟨*proof*⟩

lemma *close-lu-closed'*: — Unused
lower-upper-closed (*dbm-to-cs n v* (*close-lu M*) ∪ *dbm-to-cs n v M*) **if** *M*
0 0 ≥ *0*
⟨*proof*⟩

lemma *up-cs-up'-equiv*:
fixes *M* :: *'t* ::*time DBM*
assumes *M 0 0* ≥ *0 clock-numbering v*
shows *up-cs* (*dbm-to-cs n v M*) ≡$_{cs}$ *dbm-to-cs n v* (*up' M*)
⟨*proof*⟩

lemma *up-equiv-cong*: — Unused
fixes *cs cs'* :: (*'x*, *'v* :: *time*) *cs*
assumes *cs* ≡$_{cs}$ *cs' finite cs finite cs' lower-upper-closed cs lower-upper-closed*

cs'
shows *up-cs cs* $\equiv_{cs}$ *up-cs cs'*
⟨*proof*⟩

lemma *DBM-up-correct*:
fixes $M :: 't ::time\ DBM$
assumes *clock-numbering v finite* $\{x.\ v\ x \leq n\}$
shows $u \in ([M]_{v,n})^{\uparrow} \longleftrightarrow u \in [up\ M]_{v,n}$
⟨*proof*⟩

end

5 Implementation of DBM Operations

theory *DBM-Operations-Impl*
imports
DBM-Operations
DBM-Normalization
Refine-Imperative-HOL.IICF
HOL−Library.IArray
begin

5.1 Misc

lemma *fold-last*:
fold f $(xs\ @\ [x])\ a = f\ x\ (fold\ f\ xs\ a)$
⟨*proof*⟩

5.2 Reset

definition
reset-canonical M k d =
$(\lambda\ i\ j.$ *if* $i = k \wedge j = 0$ *then Le d*
else if $i = 0 \wedge j = k$ *then Le* $(-d)$
else if $i = k \wedge j \neq k$ *then Le d* $+$ *M 0 j*
else if $i \neq k \wedge j = k$ *then Le* $(-d)$ $+$ *M i 0*
else M i j
)

— However, DBM entries are NOT a member of this typeclass.
lemma *canonical-is-cyc-free*:
fixes $M :: nat \Rightarrow nat \Rightarrow ('b :: \{linordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add, linordered\text{-}ab\text{-}monoid\text{-}add\})$
assumes *canonical M n*
shows *cyc-free M n*

⟨*proof*⟩

lemma *dbm-neg-add*:
  fixes $a :: ('t :: time)\ DBMEntry$
  assumes $a < 0$
  shows $a + a < 0$
⟨*proof*⟩

instance *linordered-ab-group-add* ⊆ *linordered-cancel-ab-monoid-add* ⟨*proof*⟩

lemma *Le-cancel-1*[*simp*]:
  fixes $d :: 'c :: linordered\text{-}ab\text{-}group\text{-}add$
  shows $Le\ d + Le\ (-d) = Le\ 0$
⟨*proof*⟩

lemma *Le-cancel-2*[*simp*]:
  fixes $d :: 'c :: linordered\text{-}ab\text{-}group\text{-}add$
  shows $Le\ (-d) + Le\ d = Le\ 0$
⟨*proof*⟩

lemma *reset-canonical-canonical'*:
  $canonical\ (reset\text{-}canonical\ M\ k\ (d :: 'c :: linordered\text{-}ab\text{-}group\text{-}add))\ n$
  if $M\ 0\ 0 = 0\ M\ k\ k = 0\ canonical\ M\ n\ k > 0$ **for** $k\ n :: nat$
⟨*proof*⟩

lemma *reset-canonical-canonical*:
  $canonical\ (reset\text{-}canonical\ M\ k\ (d :: 'c :: linordered\text{-}ab\text{-}group\text{-}add))\ n$
  if $\forall\ i \leq n.\ M\ i\ i = 0\ canonical\ M\ n\ k > 0$ **for** $k\ n :: nat$
⟨*proof*⟩

lemma *canonicalD*[*simp*]:
  assumes $canonical\ M\ n\ i \leq n\ j \leq n\ k \leq n$
  shows $min\ (dbm\text{-}add\ (M\ i\ k)\ (M\ k\ j))\ (M\ i\ j) = M\ i\ j$
⟨*proof*⟩

lemma *reset-reset-canonical*:
  assumes $canonical\ M\ n\ k > 0\ k \leq n\ clock\text{-}numbering\ v$
  shows $[reset\ M\ n\ k\ d]_{v,n} = [reset\text{-}canonical\ M\ k\ d]_{v,n}$
⟨*proof*⟩

lemma *reset-canonical-diag-preservation*:
  fixes $k :: nat$
  assumes $k > 0$

shows $\forall\ i \leq n.\ (\mathit{reset\text{-}canonical}\ M\ k\ d)\ i\ i = M\ i\ i$
⟨*proof*⟩

definition *reset″* **where**
$\mathit{reset''}\ M\ n\ cs\ v\ d = \mathit{fold}\ (\lambda\ c\ M.\ \mathit{reset\text{-}canonical}\ M\ (v\ c)\ d)\ cs\ M$

lemma *reset″-diag-preservation*:
assumes *clock-numbering v*
shows $\forall\ i \leq n.\ (\mathit{reset''}\ M\ n\ cs\ v\ d)\ i\ i = M\ i\ i$
⟨*proof*⟩

lemma *reset-resets*:
assumes $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)\ \mathit{clock\text{-}numbering'}\ v\ n\ v\ c \leq n$
shows $[\mathit{reset}\ M\ n\ (v\ c)\ d]_{v,n} = \{u(c := d) \mid u.\ u \in [M]_{v,n}\}$
⟨*proof*⟩

lemma *reset-eq′*:
assumes *prems*: $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)\ \mathit{clock\text{-}numbering'}\ v\ n\ v\ c \leq n$
and *eq*: $[M]_{v,n} = [M']_{v,n}$
shows $[\mathit{reset}\ M\ n\ (v\ c)\ d]_{v,n} = [\mathit{reset}\ M'\ n\ (v\ c)\ d]_{v,n}$
⟨*proof*⟩

lemma *reset-eq*:
assumes *prems*: $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)\ \mathit{clock\text{-}numbering'}\ v\ n$
and *k*: $k > 0\ k \leq n$
and *eq*: $[M]_{v,n} = [M']_{v,n}$
shows $[\mathit{reset}\ M\ n\ k\ d]_{v,n} = [\mathit{reset}\ M'\ n\ k\ d]_{v,n}$
⟨*proof*⟩

lemma *FW-reset-commute*:
assumes *prems*: $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)\ \mathit{clock\text{-}numbering'}\ v\ n\ k > 0\ k \leq n$
shows $[FW\ (\mathit{reset}\ M\ n\ k\ d)\ n]_{v,n} = [\mathit{reset}\ (FW\ M\ n)\ n\ k\ d]_{v,n}$
⟨*proof*⟩

lemma *reset-canonical-diag-presv*:
fixes $k :: \mathit{nat}$
assumes $M\ i\ i = \mathit{Le}\ 0\ k > 0$
shows $(\mathit{reset\text{-}canonical}\ M\ k\ d)\ i\ i = \mathit{Le}\ 0$
⟨*proof*⟩

lemma *reset-cycle-free*:
fixes $M :: ('t :: \mathit{time})\ \mathit{DBM}$

assumes *cycle-free M n*
and *prems*: $\forall k{\leq}n.\ k > 0 \longrightarrow (\exists c.\ v\ c = k)$ *clock-numbering′ v n k* > *0 k* ≤ *n*
shows *cycle-free* (*reset M n k d*) *n*
⟨*proof*⟩

lemma *reset′-reset″-equiv*:
assumes *canonical M n d* ≥ *0* $\forall i \leq n.\ M\ i\ i = 0$
clock-numbering′ v n $\forall\ c \in$ *set cs. v c* ≤ *n*
and *surj*: $\forall\ k \leq n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k)$
shows $[reset'\ M\ n\ cs\ v\ d]_{v,n} = [reset''\ M\ n\ cs\ v\ d]_{v,n}$
⟨*proof*⟩

Eliminating the clock numbering

definition *reset‴* **where**
reset‴ M n cs d = *fold* (λ *c M*. *reset-canonical M c d*) *cs M*

lemma *reset″-reset‴*:
assumes $\forall\ c \in$ *set cs. v c* = *c*
shows *reset″ M n cs v d* = *reset‴ M n cs d*
⟨*proof*⟩

type-synonym *′a DBM′* = *nat* × *nat* ⇒ *′a DBMEntry*

definition
reset-canonical-upd
(*M* :: (*′a* :: {*linordered-cancel-ab-monoid-add*,*uminus*}) *DBM′*) (*n*:: *nat*) (*k*:: *nat*) *d* =
fold (λ *i M*. *if i* = *k then M else M*((*k*, *i*) := *Le d* + *M*(*0*,*i*), (*i*, *k*) := *Le* (−*d*) + *M*(*i*, *0*)))
(*map nat* [*1*..*n*])
(*M*((*k*, *0*) := *Le d*, (*0*, *k*) := *Le* (−*d*)))

lemma *one-upto-Suc*:
[*1*..<*Suc i* + *1*] = [*1*..<*i*+*1*] @ [*Suc i*]
⟨*proof*⟩

lemma *one-upto-Suc′*:
[*1*..*Suc i*] = [*1*..*i*] @ [*Suc i*]
⟨*proof*⟩

lemma *one-upto-Suc″*:
[*1*..*1* + *i*] = [*1*..*i*] @ [*Suc i*]

⟨*proof*⟩

lemma *reset-canonical-upd-diag-id*:
fixes $k\ n :: nat$
assumes $k > 0$
shows $(reset\text{-}canonical\text{-}upd\ M\ n\ k\ d)\ (k,\ k) = M\ (k,\ k)$
⟨*proof*⟩

lemma *reset-canonical-upd-out-of-bounds-id1*:
fixes $i\ j\ k\ n :: nat$
assumes $i \neq k\ i > n$
shows $(reset\text{-}canonical\text{-}upd\ M\ n\ k\ d)\ (i,\ j) = M\ (i,\ j)$
⟨*proof*⟩

lemma *reset-canonical-upd-out-of-bounds-id2*:
fixes $i\ j\ k\ n :: nat$
assumes $j \neq k\ j > n$
shows $(reset\text{-}canonical\text{-}upd\ M\ n\ k\ d)\ (i,\ j) = M\ (i,\ j)$
⟨*proof*⟩

lemma *reset-canonical-upd-out-of-bounds1*:
fixes $i\ j\ k\ n :: nat$
assumes $k \leq n\ i > n$
shows $(reset\text{-}canonical\text{-}upd\ M\ n\ k\ d)\ (i,\ j) = M\ (i,\ j)$
⟨*proof*⟩

lemma *reset-canonical-upd-out-of-bounds2*:
fixes $i\ j\ k\ n :: nat$
assumes $k \leq n\ j > n$
shows $(reset\text{-}canonical\text{-}upd\ M\ n\ k\ d)\ (i,\ j) = M\ (i,\ j)$
⟨*proof*⟩

lemma *reset-canonical-upd-id1*:
fixes $k\ n :: nat$
assumes $k > 0\ i > 0\ i \leq n\ i \neq k$
shows $(reset\text{-}canonical\text{-}upd\ M\ n\ k\ d)\ (i,\ k) = Le\ (-d) + M(i,0)$
⟨*proof*⟩

lemma *reset-canonical-upd-id2*:
fixes $k\ n :: nat$
assumes $k > 0\ i > 0\ i \leq n\ i \neq k$
shows $(reset\text{-}canonical\text{-}upd\ M\ n\ k\ d)\ (k,\ i) = Le\ d + M(0,i)$
⟨*proof*⟩

lemma *reset-canonical-updid-0-1*:
fixes $n :: nat$
assumes $k > 0$
shows $(reset\text{-}canonical\text{-}upd\ M\ n\ k\ d)\ (0,\ k) = Le\ (-d)$
⟨*proof*⟩

lemma *reset-canonical-updid-0-2*:
fixes $n :: nat$
assumes $k > 0$
shows $(reset\text{-}canonical\text{-}upd\ M\ n\ k\ d)\ (k,\ 0) = Le\ d$
⟨*proof*⟩

lemma *reset-canonical-upd-id*:
fixes $n :: nat$
assumes $i \neq k\ j \neq k$
shows $(reset\text{-}canonical\text{-}upd\ M\ n\ k\ d)\ (i,j) = M\ (i,j)$
⟨*proof*⟩

lemma *reset-canonical-upd-reset-canonical*:
fixes $i\ j\ k\ n :: nat$ **and** $M :: nat \times nat \Rightarrow ('a :: \{linordered\text{-}cancel\text{-}ab\text{-}monoid\text{-}add, uminus\})$ *DBMEntry*
assumes $k > 0\ i \leq n\ j \leq n\ \forall\ i \leq n.\ \forall\ j \leq n.\ M\ (i,\ j) = M'\ i\ j$
shows $(reset\text{-}canonical\text{-}upd\ M\ n\ k\ d)(i,j) = (reset\text{-}canonical\ M'\ k\ d)\ i\ j$
(**is** *?M*(i,j) = -)
⟨*proof*⟩

lemma *reset-canonical-upd-reset-canonical'*:
fixes $i\ j\ k\ n :: nat$
assumes $k > 0\ i \leq n\ j \leq n$
shows $(reset\text{-}canonical\text{-}upd\ M\ n\ k\ d)(i,j) = (reset\text{-}canonical\ (curry\ M)\ k\ d)\ i\ j$ (**is** *?M*(i,j) = -)
⟨*proof*⟩

lemma *reset-canonical-upd-canonical*:
$canonical\ (curry\ (reset\text{-}canonical\text{-}upd\ M\ n\ k\ (d :: 'c :: \{linordered\text{-}ab\text{-}group\text{-}add, uminus\}))))\ n$
if $\forall\ i \leq n.\ M\ (i,\ i) = 0\ canonical\ (curry\ M)\ n\ k > 0$ **for** $k\ n :: nat$
⟨*proof*⟩

definition *reset'-upd* **where**
$reset'\text{-}upd\ M\ n\ cs\ d = fold\ (\lambda\ c\ M.\ reset\text{-}canonical\text{-}upd\ M\ n\ c\ d)\ cs\ M$

lemma *reset'''-reset'-upd*:
fixes $n :: nat$ **and** $cs :: nat\ list$

assumes $\forall\ c \in set\ cs.\ c \neq 0\ i \leq n\ j \leq n \forall\ i \leq n.\ \forall\ j \leq n.\ M\ (i, j) = M'\ i\ j$
shows $(reset'\text{-}upd\ M\ n\ cs\ d)\ (i, j) = (reset'''\ M'\ n\ cs\ d)\ i\ j$
⟨*proof*⟩

lemma *reset'''-reset'-upd'*:
fixes n:: *nat* **and** cs :: *nat list* **and** M :: $('a :: \{linordered\text{-}cancel\text{-}ab\text{-}monoid\text{-}add, uminus\})$ DBM'
assumes $\forall\ c \in set\ cs.\ c \neq 0\ i \leq n\ j \leq n$
shows $(reset'\text{-}upd\ M\ n\ cs\ d)\ (i, j) = (reset'''\ (curry\ M)\ n\ cs\ d)\ i\ j$
⟨*proof*⟩

lemma *reset'-upd-out-of-bounds1*:
fixes $i\ j\ k\ n :: nat$
assumes $\forall\ c \in set\ cs.\ c \leq n\ i > n$
shows $(reset'\text{-}upd\ M\ n\ cs\ d)\ (i, j) = M\ (i, j)$
⟨*proof*⟩

lemma *reset'-upd-out-of-bounds2*:
fixes $i\ j\ k\ n :: nat$
assumes $\forall\ c \in set\ cs.\ c \leq n\ j > n$
shows $(reset'\text{-}upd\ M\ n\ cs\ d)\ (i, j) = M\ (i, j)$
⟨*proof*⟩

lemma *reset-canonical-int-preservation*:
fixes $n :: nat$
assumes $dbm\text{-}int\ M\ n\ d \in \mathbb{Z}$
shows $dbm\text{-}int\ (reset\text{-}canonical\ M\ k\ d)\ n$
⟨*proof*⟩

lemma *reset-canonical-upd-int-preservation*:
assumes $dbm\text{-}int\ (curry\ M)\ n\ d \in \mathbb{Z}\ k > 0$
shows $dbm\text{-}int\ (curry\ (reset\text{-}canonical\text{-}upd\ M\ n\ k\ d))\ n$
⟨*proof*⟩

lemma *reset'-upd-int-preservation*:
assumes $dbm\text{-}int\ (curry\ M)\ n\ d \in \mathbb{Z} \forall\ c \in set\ cs.\ c \neq 0$
shows $dbm\text{-}int\ (curry\ (reset'\text{-}upd\ M\ n\ cs\ d))\ n$
⟨*proof*⟩

lemma *reset-canonical-upd-diag-preservation*:
fixes $i :: nat$
assumes $k > 0$
shows $\forall\ i \leq n.\ (reset\text{-}canonical\text{-}upd\ M\ n\ k\ d)\ (i, i) = M\ (i, i)$

⟨*proof*⟩

lemma *reset'-upd-diag-preservation*:
assumes $\forall\ c \in set\ cs.\ c > 0\ i \leq n$
shows $(reset'\text{-}upd\ M\ n\ cs\ d)\ (i,\ i) = M\ (i,\ i)$
⟨*proof*⟩

lemma *upto-from-1-upt*:
fixes $n :: nat$
shows $map\ nat\ [1..int\ n] = [1..<n+1]$
⟨*proof*⟩

lemma *reset-canonical-upd-alt-def*:
reset-canonical-upd $(M :: ('a :: \{linordered\text{-}cancel\text{-}ab\text{-}monoid\text{-}add, uminus\}))$
$DBM')\ (\ n::\ nat)\ (k :: nat)\ d =$
fold
$(\lambda\ i\ M.$
if $i = k$ *then*
M
else do {
let $m0i = op\text{-}mtx\text{-}get\ M(0,i)$;
let $mi0 = op\text{-}mtx\text{-}get\ M(i,\ 0)$;
$M((k,\ i) := Le\ d + m0i,\ (i,\ k) := Le\ (-d) + mi0)$
}
)
$[1..<n+1]$
$(M((k,\ 0) := Le\ d,\ (0,\ k) := Le\ (-d)))$

⟨*proof*⟩

5.3 Relaxation

named-theorems *dbm-entry-simps*

lemma [*dbm-entry-simps*]:
$a + \infty = \infty$
⟨*proof*⟩

lemma [*dbm-entry-simps*]:
$\infty + b = \infty$
⟨*proof*⟩

lemmas *any-le-inf*[*dbm-entry-simps*]

lemma *up-canonical-preservation*:
assumes *canonical M n*
shows *canonical* (*up M*) *n*
⟨*proof*⟩

definition *up-canonical* :: *'t DBM* ⇒ *'t DBM* **where**
up-canonical M = (λ *i j*. *if i* > *0* ∧ *j* = *0 then* ∞ *else M i j*)

lemma *up-canonical-eq-up*:
assumes *canonical M n i* ≤ *n j* ≤ *n*
shows *up-canonical M i j* = *up M i j*
⟨*proof*⟩

lemma *DBM-up-to-equiv*:
assumes ∀ *i* ≤ *n*. ∀ *j* ≤ *n*. *M i j* = *M' i j*
shows $[M]_{v,n} = [M']_{v,n}$
⟨*proof*⟩

lemma *up-canonical-equiv-up*:
assumes *canonical M n*
shows $[up\text{-}canonical\ M]_{v,n} = [up\ M]_{v,n}$
⟨*proof*⟩

lemma *up-canonical-diag-preservation*:
assumes ∀ *i* ≤ *n*. *M i i* = *0*
shows ∀ *i* ≤ *n*. (*up-canonical M*) *i i* = *0*
⟨*proof*⟩

no-notation *Ref.update* (- := - *62*)

definition *up-canonical-upd* :: *'t DBM'* ⇒ *nat* ⇒ *'t DBM'* **where**
up-canonical-upd M n = *fold* (λ *i M*. *M*((*i*,*0*) := ∞)) [*1*..<*n*+*1*] *M*

lemma *up-canonical-upd-rec*:
up-canonical-upd M (*Suc n*) = (*up-canonical-upd M n*) ((*Suc n*, *0*) := ∞)
⟨*proof*⟩

lemma *up-canonical-out-of-bounds1*:
fixes *i* :: *nat*
assumes *i* > *n*
shows *up-canonical-upd M n* (*i*, *j*) = *M*(*i*,*j*)
⟨*proof*⟩

lemma *up-canonical-out-of-bounds2*:
fixes $j :: nat$
assumes $j > 0$
shows *up-canonical-upd* $M\ n\ (i,\ j) = M(i,j)$
⟨*proof*⟩

lemma *up-canonical-upd-up-canonical*:
assumes $i \leq n\ j \leq n \forall\ i \leq n. \forall\ j \leq n.\ M\ (i,\ j) = M'\ i\ j$
shows (*up-canonical-upd* $M\ n$) $(i,\ j) =$ (*up-canonical* M') $i\ j$
⟨*proof*⟩

lemma *up-canonical-int-preservation*:
assumes *dbm-int* $M\ n$
shows *dbm-int* (*up-canonical* M) n
⟨*proof*⟩

lemma *up-canonical-upd-int-preservation*:
assumes *dbm-int* (*curry* M) n
shows *dbm-int* (*curry* (*up-canonical-upd* $M\ n$)) n
⟨*proof*⟩

lemma *up-canonical-diag-preservation'*:
(*up-canonical* M) $i\ i = M\ i\ i$
⟨*proof*⟩

lemma *up-canonical-upd-diag-preservation*:
(*up-canonical-upd* $M\ n$) $(i,\ i) = M\ (i,\ i)$
⟨*proof*⟩

5.4 Intersection

definition
unbounded-dbm $n = (\lambda\ (i,\ j).$ (*if* $i = j \vee i > n \vee j > n$ *then Le 0 else* ∞))

definition *And-upd* :: $nat \Rightarrow ('t::\{linorder,zero\})\ DBM' \Rightarrow 't\ DBM' \Rightarrow 't\ DBM'$ **where**
And-upd $n\ A\ B =$
fold $(\lambda i\ M.$
fold $(\lambda j\ M.\ M((i,j) := min\ (A(i,j))\ (B(i,j))))\ [0..<n+1]\ M)$
$[0..<n+1]$
(*unbounded-dbm* n)

lemma *fold-loop-inv-rule*:

assumes $I\ 0\ x$
assumes $\bigwedge i\ x.\ I\ i\ x \Longrightarrow i \leq n \Longrightarrow I\ (Suc\ i)\ (f\ i\ x)$
assumes $\bigwedge x.\ I\ n\ x \Longrightarrow Q\ x$
shows $Q\ (fold\ f\ [0..<n]\ x)$
⟨*proof*⟩

lemma *And-upd-min*:
assumes $i \leq n\ j \leq n$
shows $And\text{-}upd\ n\ A\ B\ (i,\ j) = min\ (A(i,j))\ (B(i,j))$
⟨*proof*⟩

lemma *And-upd-And*:
assumes $i \leq n\ j \leq n$
$\forall\ i \leq n.\ \forall\ j \leq n.\ A\ (i,\ j) = A'\ i\ j\ \forall\ i \leq n.\ \forall\ j \leq n.\ B\ (i,\ j) = B'\ i\ j$
shows $And\text{-}upd\ n\ A\ B\ (i,\ j) = And\ A'\ B'\ i\ j$
⟨*proof*⟩

5.5 Inclusion

definition *pointwise-cmp* **where**
$pointwise\text{-}cmp\ P\ n\ M\ M' = (\forall\ i \leq n.\ \forall\ j \leq n.\ P\ (M\ i\ j)\ (M'\ i\ j))$

lemma *subset-eq-pointwise-le*:
fixes $M :: real\ DBM$
assumes $canonical\ M\ n\ \forall\ i \leq n.\ M\ i\ i = 0\ \forall\ i \leq n.\ M'\ i\ i = 0$
and *prems*: $clock\text{-}numbering'\ v\ n\ \forall k{\leq}n.\ 0 < k \longrightarrow (\exists\, c.\ v\ c = k)$
shows $[M]_{v,n} \subseteq [M']_{v,n} \longleftrightarrow pointwise\text{-}cmp\ (\leq)\ n\ M\ M'$
⟨*proof*⟩

definition *check-diag* :: $nat \Rightarrow ('t :: \{linorder,\ zero\})\ DBM' \Rightarrow bool$ **where**
$check\text{-}diag\ n\ M \equiv \exists\ i \leq n.\ M\ (i,\ i) < Le\ 0$

lemma *check-diag-empty*:
fixes $n :: nat$ **and** v
assumes *surj*: $\forall\ k{\leq}n.\ 0 < k \longrightarrow (\exists\, c.\ v\ c = k)$
assumes $check\text{-}diag\ n\ M$
shows $[curry\ M]_{v,n} = \{\}$
⟨*proof*⟩

lemma *check-diag-alt-def*:
$check\text{-}diag\ n\ M = list\text{-}ex\ (\lambda\ i.\ op\text{-}mtx\text{-}get\ M\ (i,\ i) < Le\ 0)\ [0..<Suc\ n]$
⟨*proof*⟩

definition *dbm-subset* :: $nat \Rightarrow ('t :: \{linorder,\ zero\})\ DBM' \Rightarrow 't\ DBM'$

$\Rightarrow$ *bool* **where**
dbm-subset n M M′ $\equiv$ *check-diag n M* $\vee$ *pointwise-cmp* $(\leq)$ *n* (*curry M*) (*curry M′*)

lemma *dbm-subset-refl*:
dbm-subset n M M
⟨*proof*⟩

lemma *dbm-subset-trans*:
assumes *dbm-subset n M1 M2 dbm-subset n M2 M3*
shows *dbm-subset n M1 M3*
⟨*proof*⟩

lemma *canonical-nonneg-diag-non-empty*:
assumes *canonical M n* $\forall i{\leq}n.\ 0 \leq M\ i\ i\ \forall c.\ v\ c \leq n \longrightarrow 0 < v\ c$
shows $[M]_{v,n} \neq \{\}$
⟨*proof*⟩

The type constraint in this lemma is due to $[\![$*canonical ?M ?n*; $[?M]_{?v,?n} \subseteq [?M']_{?v,?n}$; $[?M]_{?v,?n} \neq \{\}$; *?i* $\leq$ *?n*; *?j* $\leq$ *?n*; *?i* $\neq$ *?j*; $\forall c.\ 0 < ?v\ c \wedge (\forall x\ y.\ ?v\ x \leq ?n \wedge ?v\ y \leq ?n \wedge ?v\ x = ?v\ y \longrightarrow x = y)$; $\forall k{\leq}?n.\ 0 < k \longrightarrow (\exists c.\ ?v\ c = k)$$]\!] \Longrightarrow$ *?M ?i ?j* $\leq$ *?M′ ?i ?j*. Proving it for a more general class of types is possible but also tricky due to a missing setup for arithmetic.

lemma *subset-eq-dbm-subset*:
fixes *M* :: *real DBM′*
assumes *canonical* (*curry M*) *n* $\vee$ *check-diag n M* $\forall\ i \leq n.\ M\ (i,\ i) \leq 0\ \forall\ i \leq n.\ M'\ (i,\ i) \leq 0$
and *cn*: *clock-numbering′ v n* **and** *surj*: $\forall\ k{\leq}n.\ 0 < k \longrightarrow (\exists c.\ v\ c = k)$
shows $[$*curry M*$]_{v,n} \subseteq [$*curry M′*$]_{v,n} \longleftrightarrow$ *dbm-subset n M M′*
⟨*proof*⟩

lemma *pointwise-cmp-alt-def*:
pointwise-cmp P n M M′ =
list-all (λ *i. list-all* (λ *j. P* (*M i j*) (*M′ i j*)) [*0..<Suc n*]) [*0..<Suc n*]
⟨*proof*⟩

lemma *dbm-subset-alt-def*[*code*]:
dbm-subset n M M′ =
(*list-ex* (λ *i. op-mtx-get M* (*i*, *i*) $<$ *Le 0*) [*0..<Suc n*] $\vee$
list-all (λ *i. list-all* (λ *j.* (*op-mtx-get M* (*i*, *j*) $\leq$ *op-mtx-get M′* (*i*, *j*))) [*0..<Suc n*]) [*0..<Suc n*])
⟨*proof*⟩

definition *pointwise-cmp-alt-def* **where**
pointwise-cmp-alt-def P n M M′ = *fold* (λ *i b*. *fold* (λ *j b*. *P* (*M i j*) (*M′ i j*) ∧ *b*) [*1*..<*Suc n*] *b*) [*1*..<*Suc n*] *True*

lemma *list-all-foldli*:
list-all P xs = *foldli xs* (λ*x*. *x* = *True*) (λ *x* -. *P x*) *True*
⟨*proof*⟩

lemma *list-ex-foldli*:
list-ex P xs = *foldli xs Not* (λ *x y*. *P x* ∨ *y*) *False*
⟨*proof*⟩

5.6 Extrapolations

context
fixes
upd-entry :: *nat* ⇒ *nat* ⇒ *′t* ⇒ *′t* ⇒ (*′t*::{*linordered-ab-group-add*}) *DBMEntry* ⇒ *′t DBMEntry*
and *upd-entry-0* :: *nat* ⇒ *′t* ⇒ *′t DBMEntry* ⇒ *′t DBMEntry*
begin

definition *extra* ::
′t DBM ⇒ (*nat* ⇒ *′t*) ⇒ (*nat* ⇒ *′t*) ⇒ *nat* ⇒ *′t DBM*
where
extra M l u n ≡ λ*i j*.
let ub = *if i* > *0 then l i else 0 in*
let lb = *if j* > *0 then u j else 0 in*
if i ≤ *n* ∧ *j* ≤ *n then*
if i ≠ *j then*
if i > *0 then upd-entry i j lb ub* (*M i j*) *else upd-entry-0 j lb* (*M i j*)
else norm-diag (*M i j*)
else M i j

definition *upd-line-0* ::
′t DBM′ ⇒ *′t list* ⇒ *nat* ⇒ *′t DBM′*
where
upd-line-0 M k n =
fold
(λ*j M*.
M((*0*, *j*) := *upd-entry-0 j* (*op-list-get k j*) (*M*(*0*, *j*))))
[*1*..<*Suc n*]
(*M*((*0*, *0*) := *norm-diag* (*M* (*0*, *0*))))

definition *upd-line* ::
't DBM' ⇒ 't list ⇒ 't ⇒ nat ⇒ nat ⇒ 't DBM'
where
upd-line M k ub i n =
fold
(λj M.
if i ≠ j then
M((i, j) := upd-entry i j (op-list-get k j) ub (M(i, j)))
else M((i, j) := norm-diag (M (i, j))))
[1..<Suc n]
(M((i, 0) := upd-entry i 0 0 ub (M(i, 0))))

lemma *upd-line-Suc-unfold*:
upd-line M k ub i (Suc n) = (let M' = upd-line M k ub i n in
if i ≠ Suc n then
M' ((i, Suc n) := upd-entry i (Suc n) (op-list-get k (Suc n)) ub (M'(i, Suc n)))
else M' ((i, Suc n) := norm-diag (M' (i, Suc n))))
⟨*proof*⟩

lemma *upd-line-out-of-bounds*:
assumes *j > n*
shows *upd-line M k ub i n (i', j) = M (i', j)*
⟨*proof*⟩

lemma *upd-line-alt-def*:
assumes *i > 0*
shows
upd-line M k ub i n (i', j) = (
let lb = if j > 0 then op-list-get k j else 0 in
if i' = i ∧ j ≤ n then
if i ≠ j then
upd-entry i j lb ub (M (i, j))
else
norm-diag (M (i, j))
else M (i', j)
)
⟨*proof*⟩

lemma *upd-line-0-alt-def*:
upd-line-0 M k n (i', j) = (
if i' = 0 ∧ j ≤ n then
if j > 0 then upd-entry-0 j (op-list-get k j) (M (0, j)) else norm-diag (M (0, 0))

else $M\ (i', j)$
)
⟨*proof*⟩

definition *extra-upd* :: *'t DBM' ⇒ 't list ⇒ 't list ⇒ nat ⇒ 't DBM'*
where
extra-upd M l u n ≡
fold (*λi M. upd-line M u* (*op-list-get l i*) *i n*) [*1..<Suc n*] (*upd-line-0 M u n*)

lemma *upd-line-0-out-ouf-bounds1*:
assumes $i > 0$
shows *upd-line-0 M k n* $(i, j) = M\ (i, j)$
⟨*proof*⟩

lemma *upd-line-0-out-ouf-bounds2*:
assumes $j > n$
shows *upd-line-0 M k n* $(i, j) = M\ (i, j)$
⟨*proof*⟩

lemma *upd-out-of-bounds-aux1*:
assumes $i > n$
shows *fold* (*λi M. upd-line M k* (*op-list-get l i*) *i m*) [*1..<Suc n*] *M* (i, j) $= M\ (i, j)$
⟨*proof*⟩

lemma *upd-out-of-bounds-aux2*:
assumes $j > m$
shows *fold* (*λi M. upd-line M k* (*op-list-get l i*) *i m*) [*1..<Suc n*] *M* (i, j) $= M\ (i, j)$
⟨*proof*⟩

lemma *upd-out-of-bounds1*:
assumes $i > n$
shows *extra-upd M l u n* $(i, j) = M\ (i, j)$
⟨*proof*⟩

lemma *upd-out-of-bounds2*:
assumes $j > n$
shows *extra-upd M l u n* $(i, j) = M\ (i, j)$
⟨*proof*⟩

definition *norm-entry* **where**
norm-entry x l u i j = (

let ub = if i > 0 then (l ! i) else 0 in
let lb = if j > 0 then (u ! j) else 0 in
if i ≠ j then if i = 0 then upd-entry-0 j lb x else upd-entry i j lb ub x else norm-diag x)

lemma *upd-extra-aux*:
assumes *i ≤ n j ≤ m*
shows
fold (λi M. upd-line M u (op-list-get l i) i m) [1..<Suc n] (upd-line-0 M u m) (i, j)
= norm-entry (M (i, j)) l u i j
⟨*proof*⟩

lemma *upd-extra-aux′*:
assumes *i ≤ n j ≤ n*
shows *extra-upd M l u n (i, j) = extra (curry M) (λi. l ! i) (λi. u ! i) n i j*
⟨*proof*⟩

lemma *extra-upd-extra″*:
extra-upd M l u n (i, j) = extra (curry M) (λi. l ! i) (λi. u ! i) n i j
⟨*proof*⟩

lemma *extra-upd-extra′*:
curry (extra-upd M l u n) = extra (curry M) (λi. l ! i) (λi. u ! i) n
⟨*proof*⟩

lemma *extra-upd-extra*:
extra-upd = (λM l u n (i, j). extra (curry M) (λi. l ! i) (λi. u ! i) n i j)
⟨*proof*⟩

end

lemma *norm-is-extra*:
norm M k n =
extra
(λ- - lb ub e. norm-lower (norm-upper e ub) (−lb))
(λ- lb e. norm-lower (norm-upper e 0) (−lb)) M k k n
⟨*proof*⟩

lemma *extra-lu-is-extra*:
extra-lu M l u n =
extra
(λ- - lb ub e. norm-lower (norm-upper e ub) (−lb))

$(\lambda\text{-}\ lb\ e.\ norm\text{-}lower\ (norm\text{-}upper\ e\ 0)\ (-lb))\ M\ l\ u\ n$
⟨*proof*⟩

lemma *extra-lup-is-extra*:
extra-lup M l u n =
extra
$(\lambda i\ j\ lb\ ub\ e.\ if\ Lt\ ub \prec e\ then\ \infty$
else if M 0 i $\prec$ *Lt* $(-\ ub)$ *then* ∞
else if M 0 j $\prec$ *(if j* $>$ *0 then Lt* $(-\ lb)$ *else Lt 0) then* ∞
else e)
$(\lambda j\ lb\ e.\ if\ Le\ 0 \prec M\ 0\ j\ then\ \infty$
else if M 0 j $\prec$ *(if j* $>$ *0 then Lt* $(-\ lb)$ *else Lt 0) then Lt* $(-\ lb)$
else M 0 j) *M l u n*
⟨*proof*⟩

definition
norm-upd M k =
extra-upd
$(\lambda\text{-}\ \text{-}\ lb\ ub\ e.\ norm\text{-}lower\ (norm\text{-}upper\ e\ ub)\ (-lb))$
$(\lambda\text{-}\ lb\ e.\ norm\text{-}lower\ (norm\text{-}upper\ e\ 0)\ (-lb))\ M\ k\ k$

definition
extra-lu-upd =
extra-upd
$(\lambda\text{-}\ \text{-}\ lb\ ub\ e.\ norm\text{-}lower\ (norm\text{-}upper\ e\ ub)\ (-lb))$
$(\lambda\text{-}\ lb\ e.\ norm\text{-}lower\ (norm\text{-}upper\ e\ 0)\ (-lb))$

definition
extra-lup-upd M =
extra-upd
$(\lambda i\ j\ lb\ ub\ e.\ if\ Lt\ ub \prec e\ then\ \infty$
else if M $(0, i) \prec$ *Lt* $(-\ ub)$ *then* ∞
else if M $(0, j) \prec$ *(if j* $>$ *0 then Lt* $(-\ lb)$ *else Lt 0) then* ∞
else e)
$(\lambda j\ lb\ e.\ if\ Le\ 0 \prec M\ (0, j)\ then\ \infty$
else if M $(0, j) \prec$ *(if j* $>$ *0 then Lt* $(-\ lb)$ *else Lt 0) then Lt* $(-\ lb)$
else M $(0, j))$ *M*

lemma *extra-upd-cong*:
assumes $\bigwedge i\ j\ x\ y\ e.\ i \leq n \Longrightarrow j \leq n \Longrightarrow$ *upd-entry i j x y e* = *upd-entry′ i j x y e*
$\bigwedge i\ x\ e.\ i \leq n \Longrightarrow$ *upd-entry-0 i x e* = *upd-entry-0′ i x e*
shows *extra-upd upd-entry upd-entry-0 M l u n* = *extra-upd upd-entry′ upd-entry-0′ M l u n*

⟨*proof*⟩

lemma *extra-lup-upd-alt-def*:
extra-lup-upd M l u n = (
let xs = *IArray* (*map* (λ*i*. *M* (*0*, *i*)) [*0*..<*Suc n*]) *in*
extra-upd
(λ*i j lb ub e*. *if Lt ub* ≺ *e then* ∞
else if (*xs* !! *i*) ≺ *Lt* (− *ub*) *then* ∞
else if (*xs* !! *j*) ≺ (*if j* > *0 then Lt* (− *lb*) *else Lt 0*) *then* ∞
else e)
(λ*j lb e*. *if Le 0* ≺ (*xs* !! *j*) *then* ∞
else if (*xs* !! *j*) ≺ (*if j* > *0 then Lt* (− *lb*) *else Lt 0*) *then Lt* (− *lb*)
else (*xs* !! *j*))) *M l u n*
⟨*proof*⟩

lemma *extra-lup-upd-alt-def2*:
extra-lup-upd M l u n = (
let xs = *map* (λ*i*. *M* (*0*, *i*)) [*0*..<*Suc n*] *in*
extra-upd
(λ*i j lb ub e*. *if Lt ub* ≺ *e then* ∞
else if (*xs* ! *i*) ≺ *Lt* (− *ub*) *then* ∞
else if (*xs* ! *j*) ≺ (*if j* > *0 then Lt* (− *lb*) *else Lt 0*) *then* ∞
else e)
(λ*j lb e*. *if Le 0* ≺ (*xs* ! *j*) *then* ∞
else if (*xs* ! *j*) ≺ (*if j* > *0 then Lt* (− *lb*) *else Lt 0*) *then Lt* (− *lb*)
else (*xs* ! *j*)) *M l u n*)
⟨*proof*⟩

lemma *norm-upd-norm*: *norm-upd* = (λ*M k n* (*i*, *j*). *norm* (*curry M*) (λ*i*. *k* ! *i*) *n i j*)
and *extra-lu-upd-extra-lu*:
extra-lu-upd = (λ*M l u n* (*i*, *j*). *extra-lu* (*curry M*) (λ*i*. *l* ! *i*) (λ*i*. *u* ! *i*) *n i j*)
and *extra-lup-upd-extra-lup*:
extra-lup-upd = (λ*M l u n* (*i*, *j*). *extra-lup* (*curry M*) (λ*i*. *l* ! *i*) (λ*i*. *u* ! *i*) *n i j*)
⟨*proof*⟩

lemma *norm-upd-norm′*:
curry (*norm-upd M k n*) = *norm* (*curry M*) (λ*i*. *k* ! *i*) *n*
⟨*proof*⟩
lemma *norm-int-preservation*:
assumes *dbm-int M n* ∀ *c* ≤ *n*. *k c* ∈ ℤ
shows *dbm-int* (*norm M k n*) *n*

⟨*proof*⟩

lemma
assumes *dbm-int M n* ∀ $c \leq n$. *l c* ∈ ℤ ∀ $c \leq n$. *u c* ∈ ℤ
shows *extra-lu-preservation*: *dbm-int* (*extra-lu M l u n*) *n*
and *extra-lup-preservation*: *dbm-int* (*extra-lup M l u n*) *n*
⟨*proof*⟩

lemma *norm-upd-int-preservation*:
fixes *M* :: ('*t* :: {*linordered-ab-group-add*, *ring-1*}) *DBM'*
assumes *dbm-int* (*curry M*) *n* ∀ *c* ∈ *set k*. *c* ∈ ℤ *length k* = *Suc n*
shows *dbm-int* (*curry* (*norm-upd M k n*)) *n*
⟨*proof*⟩

lemma
fixes *M* :: ('*t* :: {*linordered-ab-group-add*, *ring-1*}) *DBM'*
assumes *dbm-int* (*curry M*) *n*
∀ *c* ∈ *set l*. *c* ∈ ℤ *length l* = *Suc n* ∀ *c* ∈ *set u*. *c* ∈ ℤ *length u* = *Suc n*
shows *extra-lu-upd-int-preservation*: *dbm-int* (*curry* (*extra-lu-upd M l u n*)) *n*
and *extra-lup-upd-int-preservation*: *dbm-int* (*curry* (*extra-lup-upd M l u n*)) *n*
⟨*proof*⟩

lemma
assumes *dbm-default* (*curry M*) *n*
shows *norm-upd-default*: *dbm-default* (*curry* (*norm-upd M k n*)) *n*
and *extra-lu-upd-default*: *dbm-default* (*curry* (*extra-lu-upd M l u n*)) *n*
and *extra-lup-upd-default*: *dbm-default* (*curry* (*extra-lup-upd M l u n*)) *n*
⟨*proof*⟩

end
theory *DBM-Imperative-Loops*
imports
Refine-Imperative-HOL.IICF
begin

5.6.1 Additional proof rules for typical looping constructs

Heap-Monad.fold-map **lemma** *fold-map-ht*:
assumes *list-all* (λ*x*. <*A* ∗ *true*> *f x* <λ*r*. ↑(*Q x r*) ∗ *A*>$_t$) *xs*
shows <*A* ∗ *true*> *Heap-Monad.fold-map f xs* <λ*rs*. ↑(*list-all2* (λ*x r*. *Q x r*) *xs rs*) ∗ *A*>$_t$

⟨*proof*⟩

lemma *fold-map-ht′*:
assumes *list-all* (λ*x*. <*true*> *f x* <λ*r*. ↑(*Q x r*)>$_t$) *xs*
shows <*true*> *Heap-Monad.fold-map f xs* <λ*rs*. ↑(*list-all2* (λ*x r*. *Q x r*) *xs rs*)>$_t$
⟨*proof*⟩

lemma *fold-map-ht1*:
assumes ⋀*x xi*. <*A* ∗ *R x xi* ∗ *true*> *f xi* <λ*r*. *A* ∗ ↑(*Q x r*)>$_t$
shows
<*A* ∗ *list-assn R xs xsi* ∗ *true*>
Heap-Monad.fold-map f xsi
<λ*rs*. *A* ∗ ↑(*list-all2* (λ*x r*. *Q x r*) *xs rs*)>$_t$
⟨*proof*⟩

lemma *fold-map-ht2*:
assumes ⋀*x xi*. <*A* ∗ *R x xi* ∗ *true*> *f xi* <λ*r*. *A* ∗ *R x xi* ∗ ↑(*Q x r*)>$_t$
shows
<*A* ∗ *list-assn R xs xsi* ∗ *true*>
Heap-Monad.fold-map f xsi
<λ*rs*. *A* ∗ *list-assn R xs xsi* ∗ ↑(*list-all2* (λ*x r*. *Q x r*) *xs rs*)>$_t$
⟨*proof*⟩

lemma *fold-map-ht3*:
assumes ⋀*x xi*. <*A* ∗ *R x xi* ∗ *true*> *f xi* <λ*r*. *A* ∗ *Q x r*>$_t$
shows <*A* ∗ *list-assn R xs xsi* ∗ *true*> *Heap-Monad.fold-map f xsi* <λ*rs*. *A* ∗ *list-assn Q xs rs*>$_t$
⟨*proof*⟩

imp-for′ **and** *imp-for* **lemma** *imp-for-rule2*:
assumes
emp ⟹$_A$ *I i a*
⋀*i a*. <*A* ∗ *I i a* ∗ *true*> *ci a* <λ*r*. *A* ∗ *I i a* ∗ ↑(*r* ⟷ *c a*)>$_t$
⋀*i a*. *i* < *j* ⟹ *c a* ⟹ <*A* ∗ *I i a* ∗ *true*> *f i a* <λ*r*. *A* ∗ *I* (*i* + *1*) *r*>$_t$
⋀*a*. *I j a* ⟹$_A$ *Q a* ⋀*i a*. *i* < *j* ⟹ ¬ *c a* ⟹ *I i a* ⟹$_A$ *Q a*
i ≤ *j*
shows <*A* ∗ *true*> *imp-for i j ci f a* <λ*r*. *A* ∗ *Q r*>$_t$
⟨*proof*⟩

lemma *imp-for-rule*:
assumes

emp $\Longrightarrow_A$ *I i a*
$\bigwedge$*i a.* <*I i a* $*$ *true*> *ci a* <λ*r. I i a* $*$ $\uparrow$(*r* $\longleftrightarrow$ *c a*)>$_t$
$\bigwedge$*i a. i* $<$ *j* $\Longrightarrow$ *c a* $\Longrightarrow$ <*I i a* $*$ *true*> *f i a* <λ*r. I* (*i* + *1*) *r*>$_t$
$\bigwedge$*a. I j a* $\Longrightarrow_A$ *Q a* $\bigwedge$*i a. i* $<$ *j* $\Longrightarrow$ $\neg$ *c a* $\Longrightarrow$ *I i a* $\Longrightarrow_A$ *Q a*
i $\leq$ *j*
shows <*true*> *imp-for i j ci f a* <λ*r. Q r*>$_t$
⟨*proof*⟩

lemma *imp-for′-rule2*:
assumes
emp $\Longrightarrow_A$ *I i a*
$\bigwedge$*i a. i* $<$ *j* $\Longrightarrow$ <*A* $*$ *I i a* $*$ *true*> *f i a* <λ*r. A* $*$ *I* (*i* + *1*) *r*>$_t$
$\bigwedge$*a. I j a* $\Longrightarrow_A$ *Q a*
i $\leq$ *j*
shows <*A* $*$ *true*> *imp-for′ i j f a* <λ*r. A* $*$ *Q r*>$_t$
⟨*proof*⟩

lemma *imp-for′-rule*:
assumes
emp $\Longrightarrow_A$ *I i a*
$\bigwedge$*i a. i* $<$ *j* $\Longrightarrow$ <*I i a* $*$ *true*> *f i a* <λ*r. I* (*i* + *1*) *r*>$_t$
$\bigwedge$*a. I j a* $\Longrightarrow_A$ *Q a*
i $\leq$ *j*
shows <*true*> *imp-for′ i j f a* <λ*r. Q r*>$_t$
⟨*proof*⟩

lemma *nth-rule*:
assumes *is-pure S*
and *b* $<$ *length a*
shows
<*nat-assn b bi* $*$ *array-assn S a ai*> *Array.nth ai bi*
<λ*r.* $\exists_A$*x. nat-assn b bi* $*$ *array-assn S a ai* $*$ *S x r* $*$ *true* $*$ $\uparrow$ (*x* = *a*
! *b*)>
⟨*proof*⟩

lemma *imp-for-list-all*:
assumes
is-pure R n $\leq$ *length xs*
$\bigwedge$*x xi.* <*A* $*$ *R x xi* $*$ *true*> *Pi xi* <λ*r. A* $*$ $\uparrow$ (*r* $\longleftrightarrow$ *P x*)>$_t$
shows
<*A* $*$ *array-assn R xs a* $*$ *true*>
imp-for 0 n Heap-Monad.return
(λ*i -. do* {
x $\leftarrow$ *Array.nth a i*; *Pi x*

})
True
<λr. *A* ∗ *array-assn R xs a* ∗ ↑(*r* ⟷ *list-all P* (*take n xs*))>$_t$
⟨*proof*⟩

lemma *imp-for-list-ex*:
assumes
is-pure R *n* ≤ *length xs*
⋀*x xi*. <*A* ∗ *R x xi* ∗ *true*> *Pi xi* <λr. *A* ∗ ↑ (*r* ⟷ *P x*)>$_t$
shows
<*A* ∗ *array-assn R xs a* ∗ *true*>
imp-for 0 n (λ*x*. *Heap-Monad.return* (¬ *x*))
(λ*i* -. *do* {
x ← *Array.nth a i*; *Pi x*
})
False
<λr. *A* ∗ *array-assn R xs a* ∗ ↑(*r* ⟷ *list-ex P* (*take n xs*))>$_t$
⟨*proof*⟩

lemma *imp-for-list-all2*:
assumes
is-pure R is-pure S *n* ≤ *length xs* *n* ≤ *length ys*
⋀*x xi y yi*. <*A* ∗ *R x xi* ∗ *S y yi* ∗ *true*> *Pi xi yi* <λr. *A* ∗ ↑ (*r* ⟷ *P x y*)>$_t$
shows
<*A* ∗ *array-assn R xs a* ∗ *array-assn S ys b* ∗ *true*>
imp-for 0 n Heap-Monad.return
(λ*i* -. *do* {
x ← *Array.nth a i*; *y* ← *Array.nth b i*; *Pi x y*
})
True
<λr. *A* ∗ *array-assn R xs a* ∗ *array-assn S ys b* ∗ ↑(*r* ⟷ *list-all2 P* (*take n xs*) (*take n ys*))>$_t$
⟨*proof*⟩

lemma *imp-for-list-all2'*:
assumes
is-pure R is-pure S *n* ≤ *length xs* *n* ≤ *length ys*
⋀*x xi y yi*. <*R x xi* ∗ *S y yi*> *Pi xi yi* <λr. ↑ (*r* ⟷ *P x y*)>$_t$
shows
<*array-assn R xs a* ∗ *array-assn S ys b*>
imp-for 0 n Heap-Monad.return
(λ*i* -. *do* {
x ← *Array.nth a i*; *y* ← *Array.nth b i*; *Pi x y*

$\})$
$True$
$<\lambda r.\ array\text{-}assn\ R\ xs\ a\ *\ array\text{-}assn\ S\ ys\ b * \uparrow(r \longleftrightarrow list\text{-}all2\ P\ (take\ n\ xs)\ (take\ n\ ys))>_t$
$\langle proof \rangle$

end
theory *DBM-Operations-Impl-Refine*
imports
DBM-Operations-Impl
HOL−Library.IArray
DBM-Imperative-Loops
begin

lemma *rev-map-fold-append-aux*:
$fold\ (\lambda\ x\ xs.\ f\ x\ \#\ xs)\ xs\ zs\ @\ ys = fold\ (\lambda\ x\ xs.\ f\ x\ \#\ xs)\ xs\ (zs@ys)$
$\langle proof \rangle$

lemma *rev-map-fold*:
$rev\ (map\ f\ xs) = fold\ (\lambda\ x\ xs.\ f\ x\ \#\ xs)\ xs\ []$
$\langle proof \rangle$

lemma *map-rev-fold*:
$map\ f\ xs = rev\ (fold\ (\lambda\ x\ xs.\ f\ x\ \#\ xs)\ xs\ [])$
$\langle proof \rangle$

lemma *pointwise-cmp-iff*:
$pointwise\text{-}cmp\ P\ n\ M\ M' \longleftrightarrow list\text{-}all2\ P\ (take\ ((n\ +\ 1)\ *\ (n\ +\ 1))\ xs)\ (take\ ((n\ +\ 1)\ *\ (n\ +\ 1))\ ys)$
if $\forall i{\leq}n.\ \forall j{\leq}n.\ xs\ !\ (i\ +\ i\ *\ n\ +\ j) = M\ i\ j$
$\forall i{\leq}n.\ \forall j{\leq}n.\ ys\ !\ (i\ +\ i\ *\ n\ +\ j) = M'\ i\ j$
$(n\ +\ 1)\ *\ (n\ +\ 1) \leq length\ xs\ (n\ +\ 1)\ *\ (n\ +\ 1) \leq length\ ys$
$\langle proof \rangle$

fun *intersperse* :: $'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
$intersperse\ sep\ (x\ \#\ y\ \#\ xs) = x\ \#\ sep\ \#\ intersperse\ sep\ (y\ \#\ xs)\ |$
$intersperse\ -\ xs = xs$

lemma *the-pure-id-assn-eq*[*simp*]:
$the\text{-}pure\ (\lambda a\ c.\ \uparrow\ (c = a)) = Id$
$\langle proof \rangle$

lemma *pure-eq-conv*:
$(\lambda a\ c.\ \uparrow\ (c = a)) = id\text{-}assn$

⟨*proof*⟩

5.7 Refinement

instance *DBMEntry* :: ({*countable*}) *countable*
⟨*proof*⟩

instance *DBMEntry* :: ({*heap*}) *heap* ⟨*proof*⟩

definition *dbm-subset'* :: *nat* ⇒ ('*t* :: {*linorder*, *zero*}) *DBM'* ⇒ '*t DBM'* ⇒ *bool* **where**
dbm-subset' n M M' ≡ *pointwise-cmp* (≤) *n* (*curry M*) (*curry M'*)

lemma *dbm-subset'-alt-def*:
dbm-subset' n M M' ≡
list-all (λ*i*. *list-all* (λ*j*. (*op-mtx-get M* (*i*, *j*) ≤ *op-mtx-get M'* (*i*, *j*))) [*0*..<*Suc n*]) [*0*..<*Suc n*]
⟨*proof*⟩

lemma *dbm-subset-alt-def'*[*code*]:
dbm-subset n M M' ⟷
list-ex (λ*i*. *op-mtx-get M* (*i*, *i*) < *0*) [*0*..<*Suc n*] ∨
list-all (λ*i*. *list-all* (λ*j*. (*op-mtx-get M* (*i*, *j*) ≤ *op-mtx-get M'* (*i*, *j*))) [*0*..<*Suc n*]) [*0*..<*Suc n*]
⟨*proof*⟩

definition
mtx-line-to-iarray m M = *IArray* (*map* (λ*i*. *M* (*0*, *i*)) [*0*..<*Suc m*])

definition
mtx-line m (*M* :: - *DBM'*) = *map* (λ*i*. *M* (*0*, *i*)) [*0*..<*Suc m*]

locale *DBM-Impl* =
fixes *n* :: *nat*
begin

abbreviation
mtx-assn :: (*nat* × *nat* ⇒ ('*a* :: {*linordered-ab-monoid-add*, *heap*})) ⇒ '*a array* ⇒ *assn*
where
mtx-assn ≡ *asmtx-assn* (*Suc n*) *id-assn*

abbreviation *clock-assn* ≡ *nbn-assn* (*Suc n*)

lemmas *Relation.IdI*[**where** *a* = ∞, *sepref-import-param*]
lemma [*sepref-import-param*]: ((+),(+)) ∈ *Id*→*Id*→*Id* ⟨*proof*⟩
lemma [*sepref-import-param*]: (*uminus*,*uminus*) ∈ (*Id*::(-∗-)*set*)→*Id* ⟨*proof*⟩
lemma [*sepref-import-param*]: (*Lt*,*Lt*) ∈ *Id*→*Id* ⟨*proof*⟩
lemma [*sepref-import-param*]: (*Le*,*Le*) ∈ *Id*→*Id* ⟨*proof*⟩
lemma [*sepref-import-param*]: (∞,∞) ∈ *Id* ⟨*proof*⟩
lemma [*sepref-import-param*]: (*min* :: - *DBMEntry* ⇒ -, *min*) ∈ *Id* → *Id* → *Id* ⟨*proof*⟩
lemma [*sepref-import-param*]: (*Suc*, *Suc*) ∈ *Id* → *Id* ⟨*proof*⟩

lemma [*sepref-import-param*]: (*norm-lower*, *norm-lower*) ∈ *Id*→*Id*→*Id* ⟨*proof*⟩
lemma [*sepref-import-param*]: (*norm-upper*, *norm-upper*) ∈ *Id*→*Id*→*Id* ⟨*proof*⟩
lemma [*sepref-import-param*]: (*norm-diag*, *norm-diag*) ∈ *Id*→*Id* ⟨*proof*⟩

end

definition *zero-clock* :: - :: *linordered-cancel-ab-monoid-add* **where**
zero-clock = *0*

sepref-register *zero-clock*

lemma [*sepref-import-param*]: (*zero-clock*, *zero-clock*) ∈ *Id* ⟨*proof*⟩

lemmas [*sepref-opt-simps*] = *zero-clock-def*

context
fixes *n* :: *nat*
begin

interpretation *DBM-Impl n* ⟨*proof*⟩

sepref-definition *reset-canonical-upd-impl′* **is**
uncurry2 (*uncurry* (λ*x*. *RETURN ooo reset-canonical-upd x*)) ::
[λ(((-,*i*),*j*),-). *i*≤*n* ∧ *j*≤*n*]$_a$ *mtx-assn*^{d} ∗$_a$ *nat-assn*^{k} ∗$_a$ *nat-assn*^{k} ∗$_a$ *id-assn*^{k} → *mtx-assn*
⟨*proof*⟩

sepref-definition *reset-canonical-upd-impl* **is**
uncurry2 (*uncurry* (λ*x*. *RETURN ooo reset-canonical-upd x*)) ::
[λ(((-,*i*),*j*),-). *i*≤*n* ∧ *j*≤*n*]$_a$ *mtx-assn*^{d} ∗$_a$ *nat-assn*^{k} ∗$_a$ *nat-assn*^{k} ∗$_a$

$id\text{-}assn^k \to mtx\text{-}assn$
⟨*proof*⟩

sepref-definition *up-canonical-upd-impl* **is**
uncurry (*RETURN oo up-canonical-upd*) :: $[\lambda(\text{-},i).\ i{\le}n]_a$ $mtx\text{-}assn^d$ $*_a$ $nat\text{-}assn^k \to mtx\text{-}assn$
⟨*proof*⟩

lemma [*sepref-import-param*]:
(*Le 0*, *0*) ∈ *Id*
⟨*proof*⟩
sepref-register *0*

sepref-definition *check-diag-impl′* **is**
uncurry (*RETURN oo check-diag*) ::
$[\lambda(i,\text{-}).\ i{\le}n]_a$ $nat\text{-}assn^k$ $*_a$ $mtx\text{-}assn^k \to bool\text{-}assn$
⟨*proof*⟩

lemma [*sepref-opt-simps*]:
(*x* = *True*) = *x*
⟨*proof*⟩

sepref-definition *dbm-subset′-impl2* **is**
uncurry2 (*RETURN ooo dbm-subset′*) ::
$[\lambda((i,\text{-}),\text{-}).\ i{\le}n]_a$ $nat\text{-}assn^k$ $*_a$ $mtx\text{-}assn^k$ $*_a$ $mtx\text{-}assn^k \to bool\text{-}assn$
⟨*proof*⟩

definition
dbm-subset′-impl′ ≡ λ*m a b*.
do {
imp-for 0 ((*m* + *1*) ∗ (*m* + *1*)) *Heap-Monad.return*
(λ*i* -. *do* {
x ← *Array.nth a i*; *y* ← *Array.nth b i*; *Heap-Monad.return* (*x* ≤ *y*)
})
True
}

lemma *imp-for-list-all2-spec*:

$<a \mapsto_a xs * b \mapsto_a ys>$
imp-for 0 n′ Heap-Monad.return
(λ*i* -. *do* {
x ← *Array.nth a i*; *y* ← *Array.nth b i*; *Heap-Monad.return* (*P x y*)
})

True
$<\lambda r.\ \uparrow(r \longleftrightarrow \textit{list-all2 P (take } n' \textit{ xs) (take } n' \textit{ ys)}) * a \mapsto_a \textit{xs} * b \mapsto_a \textit{ys}>_t$
if $n' \leq \textit{length xs}\ n' \leq \textit{length ys}$
⟨*proof*⟩

lemma *dbm-subset′-impl′-refine*:
(*uncurry2 dbm-subset′-impl′*, *uncurry2* (*RETURN* ∘∘∘ *dbm-subset′*))
∈ $[\lambda((i,\ \text{-}),\ \text{-}).\ i = n]_a$ *nat-assn*^{k} $*_a$ *local.mtx-assn*^{k} $*_a$ *local.mtx-assn*^{k} → *bool-assn*
⟨*proof*⟩

sepref-register *check-diag* ::
nat ⇒ - :: {*linordered-cancel-ab-monoid-add*,*heap*} *DBMEntry i-mtx* ⇒ *bool*

sepref-register *dbm-subset′* ::
nat ⇒ *′a* :: {*linordered-cancel-ab-monoid-add*,*heap*} *DBMEntry i-mtx* ⇒ *′a DBMEntry i-mtx* ⇒ *bool*

lemmas [*sepref-fr-rules*] = *dbm-subset′-impl′-refine check-diag-impl′.refine*

sepref-definition *dbm-subset-impl′* **is**
uncurry2 (*RETURN ooo dbm-subset*) ::
$[\lambda((i,\ \text{-}),\ \text{-}).\ i{=}n]_a$ *nat-assn*^{k} $*_a$ *mtx-assn*^{k} $*_a$ *mtx-assn*^{k} → *bool-assn*
⟨*proof*⟩

context
notes [*id-rules*] = *itypeI*[*of n TYPE* (*nat*)]
and [*sepref-import-param*] = *IdI*[*of n*]
begin

sepref-definition *dbm-subset-impl* **is**
uncurry (*RETURN oo PR-CONST* (*dbm-subset n*)) :: *mtx-assn*^{k} $*_a$ *mtx-assn*^{k} $\to_a$ *bool-assn*
⟨*proof*⟩

sepref-definition *check-diag-impl* **is**
RETURN o PR-CONST (*check-diag n*) :: *mtx-assn*^{k} $\to_a$ *bool-assn*
⟨*proof*⟩

sepref-definition *dbm-subset′-impl* **is**
uncurry (*RETURN oo PR-CONST* (*dbm-subset′ n*)) :: *mtx-assn*^{k} $*_a$ *mtx-assn*^{k} $\to_a$ *bool-assn*
⟨*proof*⟩

end

abbreviation
iarray-assn x y ≡ *pure* (*br IArray* (λ-. *True*)) *y x*

lemma [*sepref-fr-rules*]:
(*uncurry* (*return oo IArray.sub*), *uncurry* (*RETURN oo op-list-get*))
∈ *iarray-assn*^{k} $*_a$ *id-assn*^{k} $\to_a$ *id-assn*
⟨*proof*⟩

lemmas *extra-defs* = *extra-upd-def upd-line-def upd-line-0-def*

sepref-definition *norm-upd-impl* **is**
uncurry2 (*RETURN ooo norm-upd*) ::
[λ((-, *xs*), *i*). *length xs* > *n* ∧ *i*≤*n*]$_a$ *mtx-assn*^{d} $*_a$ *iarray-assn*^{k} $*_a$ *nat-assn*^{k} → *mtx-assn*
⟨*proof*⟩

sepref-definition *norm-upd-impl′* **is**
uncurry2 (*RETURN ooo norm-upd*) ::
[λ((-, *xs*), *i*). *length xs* > *n* ∧ *i*≤*n*]$_a$ *mtx-assn*^{d} $*_a$ (*list-assn id-assn*)^k $*_a$ *nat-assn*^{k} → *mtx-assn*
⟨*proof*⟩

sepref-definition *extra-lu-upd-impl* **is**
uncurry3 (λ*x*. *RETURN ooo* (*extra-lu-upd x*)) ::
[λ(((-, *ys*), *xs*), *i*). *length xs* > *n* ∧ *length ys* > *n* ∧ *i*≤*n*]$_a$
mtx-assn^d $*_a$ *iarray-assn*^{k} $*_a$ *iarray-assn*^{k} $*_a$ *nat-assn*^{k} → *mtx-assn*
⟨*proof*⟩

sepref-definition *mtx-line-to-list-impl* **is**
uncurry (*RETURN oo PR-CONST mtx-line*) ::
[λ(*m*, -). *m* ≤ *n*]$_a$ *nat-assn*^{k} $*_a$ *mtx-assn*^{k} → *list-assn id-assn*
⟨*proof*⟩

context
fixes *m* :: *nat* **assumes** *m* ≤ *n*
notes [*id-rules*] = *itypeI*[*of m TYPE* (*nat*)]
and [*sepref-import-param*] = *IdI*[*of m*]
begin

sepref-definition *mtx-line-to-list-impl2* **is**
RETURN o PR-CONST mtx-line m :: *mtx-assn*^{k} $\to_a$ *list-assn id-assn*

⟨*proof*⟩

end

lemma *IArray-impl*:
(*return o IArray*, *RETURN o id*) ∈ (*list-assn id-assn*)^k →$_a$ *iarray-assn*
⟨*proof*⟩

definition
mtx-line-to-iarray-impl m M = (*mtx-line-to-list-impl2 m M* ⤜ *return o IArray*)

lemmas *mtx-line-to-iarray-impl-ht* =
mtx-line-to-list-impl2.refine[*to-hnr*, *unfolded hn-refine-def hn-ctxt-def*, *simplified*]

lemmas *IArray-ht* = *IArray-impl*[*to-hnr*, *unfolded hn-refine-def hn-ctxt-def*, *simplified*]

lemma *mtx-line-to-iarray-impl-refine*[*sepref-fr-rules*]:
(*uncurry mtx-line-to-iarray-impl*, *uncurry* (*RETURN* ∘∘ *mtx-line*))
∈ [λ(*m*, -). *m* ≤ *n*]$_a$ *nat-assn*^{k} *$_a$ *mtx-assn*^{k} → *iarray-assn*
⟨*proof*⟩

sepref-register *mtx-line* :: *nat* ⇒ (′*ef*) *DBMEntry i-mtx* ⇒ ′*ef DBMEntry list*

lemma [*sepref-import-param*]: (*dbm-lt* :: - *DBMEntry* ⇒ -, *dbm-lt*) ∈ *Id* → *Id* → *Id* ⟨*proof*⟩

sepref-definition *extra-lup-upd-impl* **is**
uncurry3 (λ*x*. *RETURN ooo* (*extra-lup-upd x*)) ::
[λ((-, *ys*), *xs*), *i*). *length xs* > *n* ∧ *length ys* > *n* ∧ *i*≤*n*]$_a$
mtx-assn^d *$_a$ *iarray-assn*^{k} *$_a$ *iarray-assn*^{k} *$_a$ *nat-assn*^{k} → *mtx-assn*
⟨*proof*⟩

context
notes [*id-rules*] = *itypeI*[*of n TYPE* (*nat*)]
and [*sepref-import-param*] = *IdI*[*of n*]
begin

definition
unbounded-dbm′ = *unbounded-dbm n*

lemma *unbounded-dbm-alt-def*:
unbounded-dbm n = *op-amtx-new* (*Suc n*) (*Suc n*) (*unbounded-dbm'*)
⟨*proof*⟩

We need the custom rule here because *unbounded-dbm* is a higher-order constant

lemma [*sepref-fr-rules*]:
(*uncurry0* (*return unbounded-dbm'*), *uncurry0* (*RETURN* (*PR-CONST* (*unbounded-dbm'*))))
∈ *unit-assn*^{k} →$_a$ *pure* (*nat-rel* ×$_r$ *nat-rel* → *Id*)
⟨*proof*⟩

sepref-register *PR-CONST* (*unbounded-dbm n*) :: *nat* × *nat* ⇒ *int DBMEntry* :: *'b DBMEntry i-mtx*
sepref-register *unbounded-dbm'* :: *nat* × *nat* ⇒ *- DBMEntry*

Necessary to solve side conditions of *op-amtx-new*

lemma *unbounded-dbm'-bounded*:
mtx-nonzero unbounded-dbm' ⊆ {*0..<Suc n*} × {*0..<Suc n*}
⟨*proof*⟩

We need to pre-process the lemmas due to a failure of *TRADE*

lemma *unbounded-dbm'-bounded-1*:
(*a*, *b*) ∈ *mtx-nonzero unbounded-dbm'* ⟹ *a* < *Suc n*
⟨*proof*⟩

lemma *unbounded-dbm'-bounded-2*:
(*a*, *b*) ∈ *mtx-nonzero unbounded-dbm'* ⟹ *b* < *Suc n*
⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *dbm-subset-impl.refine*

sepref-register *PR-CONST* (*dbm-subset n*) :: *'e DBMEntry i-mtx* ⇒ *'e DBMEntry i-mtx* ⇒ *bool*

lemma [*def-pat-rules*]:
dbm-subset $ *n* ≡ *PR-CONST* (*dbm-subset n*)
⟨*proof*⟩

sepref-definition *unbounded-dbm-impl* **is**
uncurry0 (*RETURN* (*PR-CONST* (*unbounded-dbm n*))) :: *unit-assn*^{k} →$_a$ *mtx-assn*
⟨*proof*⟩

DBM to List

definition *dbm-to-list* :: $(nat \times nat \Rightarrow 'a) \Rightarrow 'a\ list$ **where**
dbm-to-list M ≡
rev $ *fold* (λ*i xs*. *fold* (λ*j xs*. *M* (*i*, *j*) # *xs*) [*0*..<*Suc n*] *xs*) [*0*..<*Suc n*] []

sepref-definition *dbm-to-list-impl* **is**
RETURN o PR-CONST dbm-to-list :: $mtx\text{-}assn^k \rightarrow_a list\text{-}assn\ id\text{-}assn$
⟨*proof*⟩

5.8 Pretty-Printing

context
fixes *show-clock* :: *nat* ⇒ *string*
and *show-num* :: *'a* :: {*linordered-ab-group-add*,*heap*} ⇒ *string*
begin

definition
make-string e i j ≡
if i = *j then if e* < *0 then Some* (*''EMPTY''*) *else None*
else
if i = *0 then*
case e of
DBMEntry.Le a ⇒ *if a* = *0 then None else Some* (*show-clock j* @ *''* >= *''* @ *show-num* (− *a*))
| *DBMEntry.Lt a* ⇒ *Some* (*show-clock j* @ *''* > *''* @ *show-num* (− *a*))
| *-* ⇒ *None*
else if j = *0 then*
case e of
DBMEntry.Le a ⇒ *Some* (*show-clock i* @ *''* <= *''* @ *show-num a*)
| *DBMEntry.Lt a* ⇒ *Some* (*show-clock i* @ *''* < *''* @ *show-num a*)
| *-* ⇒ *None*
else
case e of
DBMEntry.Le a ⇒ *Some* (*show-clock i* @ *''* − *''* @ *show-clock j* @ *''* <= *''* @ *show-num a*)
| *DBMEntry.Lt a* ⇒ *Some* (*show-clock i* @ *''* − *''* @ *show-clock j* @ *''* < *''* @ *show-num a*)
| *-* ⇒ *None*

definition
dbm-list-to-string xs ≡
(*concat o intersperse ''*, *'' o rev o snd o snd*) $ *fold* (λ*e* (*i*, *j*, *acc*).
let

```
    v = make-string e i j;
    j = (j + 1) mod (n + 1);
    i = (if j = 0 then i + 1 else i)
  in
  case v of
    None ⇒ (i, j, acc)
  | Some s ⇒ (i, j, s # acc)
) xs (0, 0, [])
```

lemma [*sepref-import-param*]:
(*dbm-list-to-string*, *PR-CONST dbm-list-to-string*) ∈ ⟨*Id*⟩*list-rel* → ⟨*Id*⟩*list-rel*
⟨*proof*⟩

definition *show-dbm* **where**
show-dbm M ≡ *PR-CONST dbm-list-to-string* (*dbm-to-list M*)

sepref-register *PR-CONST local.dbm-list-to-string*
sepref-register *dbm-to-list* :: *'b i-mtx* ⇒ *'b list*

lemmas [*sepref-fr-rules*] = *dbm-to-list-impl.refine*

sepref-definition *show-dbm-impl* **is**
RETURN o show-dbm :: $mtx\text{-}assn^k \rightarrow_a$ *list-assn id-assn*
⟨*proof*⟩

end

end

end

5.9 Generate Code

lemma [*code*]:
dbm-le a b = (*a* = *b* ∨ (*a* ≺ *b*))
⟨*proof*⟩

export-code
norm-upd-impl
reset-canonical-upd-impl
up-canonical-upd-impl
dbm-subset-impl
dbm-subset

show-dbm-impl
checking *SML*

export-code
norm-upd-impl
reset-canonical-upd-impl
up-canonical-upd-impl
dbm-subset-impl
dbm-subset
show-dbm-impl
checking *SML-imp*

end
theory *DBM-Examples*
imports
DBM-Operations-Impl-Refine
FW-More
Show.Show-Instances
begin

5.10 Examples

no-notation *Ref.update* (- := - *62*)

Let us represent the zone $y \leq x \wedge x \;\; y \leq 2 \wedge y \geq 1$ as a DBM:

definition *test-dbm* :: *int DBM'* **where**
test-dbm = ((((λ(*i*, *j*). *Le 0*)((*1*,*2*) := *Le 2*))((*0*, *2*) := *Le* (−*1*)))((*1*, *0*) := ∞))((*2*, *0*) := ∞)

— Pretty-printing
definition *show-test-dbm* **where**
show-test-dbm M = *String.implode* (
show-dbm 2
(λ*i*. *if i* = *1 then* ″*x*″ *else if i* = *2 then* ″*y*″ *else* ″*f*″) *show*
M
)

— Pretty-printing
value [*code*] *show-test-dbm test-dbm*

— Canonical form
value [*code*] *show-test-dbm* (*FW' test-dbm 2*)

— Projection onto x axis

value [*code*] *show-test-dbm* (*reset′-upd* (*FW′ test-dbm 2*) *2* [*2*] *0*)
— Note that if *reset′-upd* is not applied to the canonical form, the result is incorrect:
value [*code*] *show-test-dbm* (*reset′-upd test-dbm 2* [*2*] *0*)
— In this case, we already obtained a new canonical form after reset:
value [*code*] *show-test-dbm* (*FW′* (*reset′-upd* (*FW′ test-dbm 2*) *2* [*2*] *0*) *2*)
— Note that *FWI* can be used to restore the canonical form without running a full *FW′*.

— Relaxation, a.k.a computing the "future", or "letting time elapse":
value [*code*] *show-test-dbm* (*up-canonical-upd* (*reset′-upd* (*FW′ test-dbm 2*) *2* [*2*] *0*) *2*)
— Note that *up-canonical-upd* always preservers canonical form.

— Intersection
value [*code*] *show-test-dbm* (*FW′* (*And-upd 2*
(*up-canonical-upd* (*reset′-upd* (*FW′ test-dbm 2*) *2* [*2*] *0*) *2*)
((λ(*i*, *j*). ∞)((*1*, *0*):=*Lt 1*))) *2*)
— Note that *up-canonical-upd* always preservers canonical form.

— Checking if DBM represents the empty zone
value [*code*] *check-diag 2* (*FW′* (*And-upd 2*
(*up-canonical-upd* (*reset′-upd* (*FW′ test-dbm 2*) *2* [*2*] *0*) *2*)
((λ(*i*, *j*). ∞)((*1*, *0*):=*Lt 1*))) *2*)

— Instead of λ(*i*, *j*). ∞ we could also have been using *unbounded-dbm*.

end

References

[1] G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. Lower and upper bounds in zone-based abstractions of timed automata. *Int. J. Softw. Tools Technol. Transf.*, 8(3):204–215, 2006.

[2] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 197–212, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[3] S. Wimmer. *Trustworthy Verification of Realtime Systems*. PhD thesis, Technical University of Munich, Germany, 2020.

[4] S. Wimmer and P. Lammich. Verified model checking of timed automata. In *TACAS (1)*, volume 10805 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 2018.