

Difference Bound Matrices

Simon Wimmer, Peter Lammich

February 4, 2026

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

1	Difference Bound Matrices	3
1.1	Definitions	3
1.1.1	Definition and Semantics of DBMs	3
1.1.2	Ordering DBM Entries	5
1.1.3	Addition on DBM Entries	6
1.1.4	Negation of DBM Entries	7
1.2	DBM Entries Form a Linearly Ordered Abelian Monoid	8
1.3	Basic Properties of DBMs	10
1.3.1	DBMs and Length of Paths	10
2	Library for Paths, Arcs and Lengths	12
2.1	Arcs	12
2.2	Length of Paths	13
2.3	Cycle Rotation	14
2.4	More on Cycle-Freeness	14
2.5	Helper Lemmas for Bouyer’s Theorem on Approximation	15
2.5.1	Successive	17

2.5.2	Zones and DBMs	21
2.5.3	Useful definitions	23
2.5.4	Updating DBMs	23
2.5.5	DBMs Without Negative Cycles are Non-Empty	24
2.5.6	Negative Cycles in DBMs	25
2.5.7	Floyd-Warshall Algorithm Preservers Zones	27
2.6	The Characteristic Property of Canonical DBMs	27
2.6.1	Floyd-Warshall and Empty DBMs	28
2.6.2	Mixed Corollaries	29
2.7	Orderings of DBMs	30
2.8	Partial Floyd-Warshall Preserves Zones	31
3	DBM Operations	31
3.1	Auxiliary	32
3.2	Relaxation	32
3.3	Intersection	34
3.4	Variable Reset	34
3.5	Misc Preservation Lemmas	39
3.5.1	Unused theorems	42
3.6	Extrapolation of DBMs	43
3.6.1	Classical extrapolation	44
3.6.2	Extrapolations based on lower and upper bounds	44
3.6.3	Extrapolations are widening operators	46
3.6.4	Finiteness of extrapolations	46
4	DBMs as Constraint Systems	47
4.1	Misc	47
4.2	Definition and Semantics of Constraint Systems	48
4.3	Conversion of DBMs to Constraint Systems and Back	50
4.4	Application: Relaxation On Constraint Systems	52
5	Implementation of DBM Operations	55
5.1	Misc	55
5.2	Reset	55
5.3	Relaxation	62
5.4	Intersection	64
5.5	Inclusion	65
5.6	Extrapolations	67
5.6.1	Additional proof rules for typical looping constructs	73
5.7	Refinement	78
5.8	Pretty-Printing	85
5.9	Generate Code	86
5.10	Examples	87

```

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 ( $\langle \infty \rangle$ )

```

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 $'c \Rightarrow \text{nat}$ 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 $\mathbf{0}$, which shall be assigned to 0 in every valuation. In the following predicate we explicitly keep track of $\mathbf{0}$.

```

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

```

```

begin

```

```

lemma non-trivial-neg:  $\exists x. x < 0$ 
   $\langle \text{proof} \rangle$ 

```

```

end

```

```

instantiation real :: time
  begin
    instance  $\langle \text{proof} \rangle$ 
  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 ⇒ nat) ⇒ ('c, 't) cval ⇒ ('t::time) DBM ⇒ nat ⇒ bool

where

$$\begin{aligned} \text{DBM-val-bounded } v \ u \ m \ n &\equiv \text{Le } 0 \preceq m \ 0 \ 0 \wedge \\ &(\forall c. v \ c \leq n \longrightarrow (\text{dbm-entry-val } u \ \text{None} \ (\text{Some } c) \ (m \ 0 \ (v \ c))) \\ &\quad \wedge \text{dbm-entry-val } u \ (\text{Some } c) \ \text{None} \ (m \ (v \ c) \ 0))) \\ &\wedge (\forall c1 \ c2. v \ c1 \leq n \wedge v \ c2 \leq n \longrightarrow \text{dbm-entry-val } u \ (\text{Some } c1) \ (\text{Some } \\ &c2) \ (m \ (v \ c1) \ (v \ c2))) \end{aligned}$$

abbreviation *DBM-val-bounded-abbrev* ::

$$('c, 't) \text{ cval} \Rightarrow ('c \Rightarrow \text{nat}) \Rightarrow \text{nat} \Rightarrow ('t::\text{time}) \text{ DBM} \Rightarrow \text{bool}$$

(<- ⊢ -, - ⊢ -> [48, 48, 48, 48] 48)

where

$$u \vdash_{v,n} M \equiv \text{DBM-val-bounded } v \ u \ M \ n$$

1.1.2 Ordering DBM Entries

abbreviation

$$\text{dmin } a \ b \equiv \text{if } a \prec b \text{ then } a \text{ else } b$$

lemma *dbm-le-dbm-min*:

$$a \preceq b \implies a = \text{dmin } a \ b \ \langle \text{proof} \rangle$$

lemma *dbm-lt-asm*:

assumes $e \prec f$

shows $\sim f \prec e$

$\langle \text{proof} \rangle$

lemma *dbm-le-dbm-min2*:

$$a \preceq b \implies a = \text{dmin } b \ a$$

$\langle \text{proof} \rangle$

lemma *dmb-le-dbm-entry-bound-inf*:

$$a \preceq b \implies a = \infty \implies b = \infty$$

$\langle \text{proof} \rangle$

lemma *dbm-not-lt-eq*: $\neg a \prec b \implies \neg b \prec a \implies a = b$

$\langle \text{proof} \rangle$

lemma *dbm-not-lt-impl*: $\neg a \prec b \implies b \prec a \vee a = b \ \langle \text{proof} \rangle$

lemma $\text{dmin } a \ b = \text{dmin } b \ a$

$\langle \text{proof} \rangle$

lemma *dbm-lt-trans*: $a \prec b \implies b \prec c \implies a \prec c$

$\langle \text{proof} \rangle$

lemma *aux-3*: $\neg a \prec b \implies \neg b \prec c \implies a \prec c \implies c = a$

$\langle \text{proof} \rangle$

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

lemma *dbm-lt-asymmetric*[*simp*]: $x \prec y \implies y \prec x \implies \text{False}$

$\langle \text{proof} \rangle$

lemma *le-dbm-le*: $Le\ a \preceq Le\ b \implies a \leq b$ $\langle \text{proof} \rangle$

lemma *le-dbm-lt*: $Le\ a \preceq Lt\ b \implies a < b$ $\langle \text{proof} \rangle$

lemma *lt-dbm-le*: $Lt\ a \preceq Le\ b \implies a \leq b$ $\langle \text{proof} \rangle$

lemma *lt-dbm-lt*: $Lt\ a \preceq Lt\ b \implies a \leq b$ $\langle \text{proof} \rangle$

lemma *not-dbm-le-le-impl*: $\neg Le\ a \prec Le\ b \implies a \geq b$ $\langle \text{proof} \rangle$

lemma *not-dbm-lt-le-impl*: $\neg Lt\ a \prec Le\ b \implies a > b$ $\langle \text{proof} \rangle$

lemma *not-dbm-lt-lt-impl*: $\neg Lt\ a \prec Lt\ b \implies a \geq b$ $\langle \text{proof} \rangle$

lemma *not-dbm-le-lt-impl*: $\neg Le\ a \prec Lt\ b \implies a \geq b$ $\langle \text{proof} \rangle$

1.1.3 Addition on DBM Entries

fun *dbm-add* :: (*t*::*linordered-cancel-ab-semigroup-add*) *DBMEntry* \Rightarrow *t* *DBMEntry* \Rightarrow *t* *DBMEntry* (**infixl** $\langle \otimes \rangle$ 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*: $x \prec y \implies \neg \text{dbm-add } x\ z \prec \text{dbm-add } y\ z \implies \text{dbm-add } x\ z = \text{dbm-add } y\ z$

$\langle \text{proof} \rangle$

lemma aux-5: $\neg x \prec y \implies \text{dbm-add } x \ z \prec \text{dbm-add } y \ z \implies \text{dbm-add } y \ z = \text{dbm-add } x \ z$

<proof>

lemma aux-42: $x \prec y \implies \neg \text{dbm-add } z \ x \prec \text{dbm-add } z \ y \implies \text{dbm-add } z \ x = \text{dbm-add } z \ y$

<proof>

lemma aux-52: $\neg x \prec y \implies \text{dbm-add } z \ x \prec \text{dbm-add } z \ y \implies \text{dbm-add } z \ y = \text{dbm-add } z \ x$

<proof>

lemma dbm-add-not-inf:

$a \neq \infty \implies b \neq \infty \implies \text{dbm-add } a \ b \neq \infty$

<proof>

lemma dbm-le-not-inf:

$a \preceq b \implies b \neq \infty \implies a \neq \infty$

<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 ∞ = ∞

— This case does not make sense but we make this definition for technical convenience.

lemma *neg-entry*:

$\{u. \neg \text{dbm-entry-val } u \ a \ b \ e\} = \{u. \text{dbm-entry-val } u \ b \ a \ (\text{neg-dbm-entry } e)\}$

if $e \neq (\infty :: - \text{DBMEntry}) \ a \neq \text{None} \vee b \neq \text{None}$

<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 $\llbracket ?e \neq \infty; ?a \neq \text{None} \vee ?b \neq \text{None} \rrbracket \implies \{u. \neg \text{dbm-entry-val } u \text{ ?a ?b ?e}\} = \{u. \text{dbm-entry-val } u \text{ ?b ?a (neg-dbm-entry ?e)}\}$ would not hold any longer, however.

1.2 DBM Entries Form a Linearly Ordered Abelian Monoid

instantiation *DBMEntry* :: (*linorder*) *linorder*

begin

definition *less-eq*: (\leq) \equiv *dbm-le*

definition *less*: ($<$) = *dbm-lt*

instance

$\langle \text{proof} \rangle$

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*

$\langle \text{proof} \rangle$

end

instantiation *DBMEntry* :: (*zero*) *zero*

begin

definition *neutral*: $0 = \text{Le } 0$

instance $\langle \text{proof} \rangle$

end

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

begin

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

instance $\langle \text{proof} \rangle$

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* \subseteq *linordered-cancel-ab-monoid-add* *<proof>*

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

lemma *dbm-lt-not-inf-less[intro]*: $A \neq \infty \implies A < \infty$ *<proof>*

lemma *add-inf[simp]*:
 $a + \infty = \infty \quad \infty + a = \infty$
<proof>

lemma *inf-lt[simp,dest!]*:
 $\infty < x \implies False$
<proof>

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

lemma *Le-Le-dbm-lt-D[dest]*: $Le a < Lt b \implies a < b$ *<proof>*

lemma *Le-Lt-dbm-lt-D[dest]*: $Le a < Le b \implies a < b$ *<proof>*

lemma *Lt-Le-dbm-lt-D[dest]*: $Lt a < Le b \implies a \leq b$ *<proof>*

lemma *Lt-Lt-dbm-lt-D[dest]*: $Lt a < Lt b \implies a < b$ *<proof>*

lemma *Le-le-LeI[intro]*: $a \leq b \implies Le a \leq Le b$ *<proof>*

lemma *Lt-le-LeI[intro]*: $a \leq b \implies Lt a \leq Le b$ *<proof>*

lemma *Lt-le-LtI[intro]*: $a \leq b \implies Lt a \leq Lt b$ *<proof>*

lemma *Le-le-LtI[intro]*: $a < b \implies Le a \leq Lt b$ *<proof>*

lemma *Lt-lt-LeI*: $x \leq y \implies Lt x < Le y$ *<proof>*

lemma *Le-le-LeD[dest]*: $Le a \leq Le b \implies a \leq b$ *<proof>*

lemma *Le-le-LtD[dest]*: $Le a \leq Lt b \implies a < b$ *<proof>*

lemma *Lt-le-LeD[dest]*: $Lt a \leq Le b \implies a \leq b$ *<proof>*

lemma *Lt-le-LtD[dest]*: $Lt a \leq Lt b \implies a \leq b$ *<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]*: $\infty < x = False$ *<proof>*

lemma *any-le-inf*: $x \leq (\infty :: - DBMEntry)$ $\langle proof \rangle$

lemma *dbm-lt-code-simps*[code]:

dbm-lt (Lt a) $\infty = True$
dbm-lt (Le a) $\infty = True$
dbm-lt (Le a) (Le b) = (a < b)
dbm-lt (Le a) (Lt b) = (a < b)
dbm-lt (Lt a) (Le b) = (a \leq b)
dbm-lt (Lt a) (Lt b) = (a < b)
dbm-lt ∞ x = False
 $\langle proof \rangle$

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 \implies
dbm-entry-val u (Some d) None b
 \implies *dbm-entry-val* u (Some c) None (dbm-add a b)
 $\langle proof \rangle$

lemma *dbm-entry-val-add-2*: *dbm-entry-val* u None (Some c) a \implies *dbm-entry-val*
u (Some c) (Some d) b
 \implies *dbm-entry-val* u None (Some d) (dbm-add a b)
 $\langle proof \rangle$

lemma *dbm-entry-val-add-3*:
dbm-entry-val u (Some c) (Some d) a \implies *dbm-entry-val* u (Some d)
(Some e) b
 \implies *dbm-entry-val* u (Some c) (Some e) (dbm-add a b)
 $\langle proof \rangle$

lemma *dbm-entry-val-add-4*:
dbm-entry-val u (Some c) None a \implies *dbm-entry-val* u None (Some d) b
 \implies *dbm-entry-val* u (Some c) (Some d) (dbm-add a b)
 $\langle proof \rangle$

no-notation *dbm-add* (infixl $\langle \otimes \rangle$ 70)

lemma *DBM-val-bounded-len-1'-aux*:

assumes *DBM-val-bounded* v u m n v c $\leq n \forall k \in set\ vs.\ k > 0 \wedge k \leq$
n $\wedge (\exists c.\ v\ c = k)$
shows *dbm-entry-val* u (Some c) None (len m (v c) 0 vs) $\langle proof \rangle$

lemma *DBM-val-bounded-len-3'-aux:*

DBM-val-bounded $v\ u\ m\ n \implies v\ c \leq n \implies v\ d \leq n \implies \forall k \in \text{set } vs. k > 0 \wedge k \leq n \wedge (\exists c. v\ c = k)$
 $\implies \text{dbm-entry-val } u\ (\text{Some } c)\ (\text{Some } d)\ (\text{len } m\ (v\ c)\ (v\ d)\ vs)$
 <proof>

lemma *DBM-val-bounded-len-2'-aux:*

DBM-val-bounded $v\ u\ m\ n \implies v\ c \leq n \implies \forall k \in \text{set } vs. k > 0 \wedge k \leq n \wedge (\exists c. v\ c = k)$
 $\implies \text{dbm-entry-val } u\ \text{None}\ (\text{Some } c)\ (\text{len } m\ 0\ (v\ c)\ vs)$
 <proof>

lemma *cnt-0-D:*

$\text{cnt } x\ xs = 0 \implies x \notin \text{set } xs$
 <proof>

lemma *cnt-at-most-1-D:*

$\text{cnt } x\ (xs\ @\ x\ \# \ ys) \leq 1 \implies x \notin \text{set } xs \wedge x \notin \text{set } ys$
 <proof>

lemma *nat-list-0 [intro]:*

$x \in \text{set } xs \implies 0 \notin \text{set } (xs :: \text{nat list}) \implies x > 0$
 <proof>

lemma *DBM-val-bounded-len'1:*

fixes v
assumes *DBM-val-bounded* $v\ u\ m\ n\ 0 \notin \text{set } vs\ v\ c \leq n$
 $\forall k \in \text{set } vs. k > 0 \longrightarrow k \leq n \wedge (\exists c. v\ c = k)$
shows $\text{dbm-entry-val } u\ (\text{Some } c)\ \text{None}\ (\text{len } m\ (v\ c)\ 0\ vs)$
 <proof>

lemma *DBM-val-bounded-len'2:*

fixes v
assumes *DBM-val-bounded* $v\ u\ m\ n\ 0 \notin \text{set } vs\ v\ c \leq n$
 $\forall k \in \text{set } vs. k > 0 \longrightarrow k \leq n \wedge (\exists c. v\ c = k)$
shows $\text{dbm-entry-val } u\ \text{None}\ (\text{Some } c)\ (\text{len } m\ 0\ (v\ c)\ vs)$
 <proof>

lemma *DBM-val-bounded-len'3:*

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

<proof>

Now unused lemma *DBM-val-bounded-len'*:

fixes v

defines $vo \equiv \lambda k. \text{if } k = 0 \text{ then None else Some (SOME } c. v\ c = k)$

assumes *DBM-val-bounded* $v\ u\ m\ n\ cnt\ 0\ (i\ \#\ j\ \#\ vs) \leq 1$

$\forall k \in \text{set } (i\ \#\ j\ \#\ vs). k > 0 \longrightarrow k \leq n \wedge (\exists c. v\ c = k)$

shows *dbm-entry-val* $u\ (vo\ i)\ (vo\ j)\ (len\ m\ i\ j\ vs)$

<proof>

lemma *DBM-val-bounded-len-1*: *DBM-val-bounded* $v\ u\ m\ n \implies v\ c \leq n$
 $\implies \forall c \in \text{set } cs. v\ c \leq n$

$\implies \text{dbm-entry-val } u\ (\text{Some } c)\ \text{None}\ (len\ m\ (v\ c)\ 0\ (\text{map } v\ cs))$

<proof>

lemma *DBM-val-bounded-len-3*: *DBM-val-bounded* $v\ u\ m\ n \implies v\ c \leq n$
 $\implies v\ d \leq n \implies \forall c \in \text{set } cs. v\ c \leq n$

$\implies \text{dbm-entry-val } u\ (\text{Some } c)\ (\text{Some } d)\ (len\ m\ (v\ c)\ (v\ d)\ (\text{map } v\ cs))$

<proof>

lemma *DBM-val-bounded-len-2*: *DBM-val-bounded* $v\ u\ m\ n \implies v\ c \leq n$
 $\implies \forall c \in \text{set } cs. v\ c \leq n$

$\implies \text{dbm-entry-val } u\ \text{None}\ (\text{Some } c)\ (len\ m\ 0\ (v\ c)\ (\text{map } v\ cs))$

<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 $\text{set } xs = \text{set } ys\ \text{distinct } xs\ \text{length } xs = \text{length } ys$

shows $\text{distinct } ys$

<proof>

2.1 Arcs

fun $\text{arcs} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list} \Rightarrow (\text{nat} * \text{nat}) \text{ list}$ **where**

$\text{arcs } a\ b\ [] = [(a,b)] \mid$

$$\text{arcs } a \ b \ (x \# \ xs) = (a, x) \# \ \text{arcs } x \ b \ xs$$

definition $\text{arcs}' :: \text{nat list} \Rightarrow (\text{nat} * \text{nat}) \text{ set}$ **where**
 $\text{arcs}' \ xs = \text{set} (\text{arcs} (\text{hd } xs) (\text{last } xs) (\text{butlast } (\text{tl } xs)))$

lemma *arcs'-decomp*:

$\text{length } xs > 1 \Longrightarrow (i, j) \in \text{arcs}' \ xs \Longrightarrow \exists \ zs \ ys. \ xs = zs \ @ \ i \ # \ j \ # \ ys$
 $\langle \text{proof} \rangle$

lemma *arcs-decomp-tail*:

$\text{arcs } j \ l \ (ys \ @ \ [i]) = \text{arcs } j \ i \ ys \ @ \ [(i, l)]$
 $\langle \text{proof} \rangle$

lemma *arcs-decomp*: $\text{arcs } x \ z \ xs = \text{arcs } x \ y \ ys \ @ \ \text{arcs } y \ z \ zs$
 $\langle \text{proof} \rangle$

lemma *distinct-arcs-ex*:

$\text{distinct } xs \Longrightarrow i \notin \text{set } xs \Longrightarrow xs \neq [] \Longrightarrow \exists \ a \ b. \ a \neq x \wedge (a, b) \in \text{set} (\text{arcs } i \ j \ xs)$
 $\langle \text{proof} \rangle$

lemma *cycle-rotate-2-aux*:

$(i, j) \in \text{set} (\text{arcs } a \ b \ (xs \ @ \ [c])) \Longrightarrow (i, j) \neq (c, b) \Longrightarrow (i, j) \in \text{set} (\text{arcs } a \ c \ xs)$
 $\langle \text{proof} \rangle$

lemma *arcs-set-elem1*:

assumes $j \neq k \ k \in \text{set} (i \ # \ xs)$
shows $\exists \ l. (k, l) \in \text{set} (\text{arcs } i \ j \ xs)$ $\langle \text{proof} \rangle$

lemma *arcs-set-elem2*:

assumes $i \neq k \ k \in \text{set} (j \ # \ xs)$
shows $\exists \ l. (l, k) \in \text{set} (\text{arcs } i \ j \ xs)$ $\langle \text{proof} \rangle$

2.2 Length of Paths

lemmas (in *linordered-ab-monoid-add*) $\text{comm} = \text{add.commute}$

lemma *len-add*:

fixes $M :: ('a :: \text{linordered-ab-monoid-add}) \text{ mat}$
shows $\text{len } M \ i \ j \ xs + \text{len } M \ i \ j \ xs = \text{len } (\lambda i \ j. M \ i \ j + M \ i \ j) \ i \ j \ xs$
 $\langle \text{proof} \rangle$

2.3 Cycle Rotation

lemma *cycle-rotate*:

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

lemma *cycle-rotate-2*:

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

lemma *cycle-rotate-len-arcs*:

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

lemma *cycle-rotate-2'*:

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

2.4 More on Cycle-Freeness

lemma *cyc-free-diag-dest*:

assumes $\text{cyc-free } M \ n \ i \leq n \ \text{set } xs \subseteq \{0..n\}$
shows $\text{len } M \ i \ i \ xs \geq 0$
 $\langle \text{proof} \rangle$

lemma *cycle-free-0-0*:

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

2.5 Helper Lemmas for Bouyer's Theorem on Approximation

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

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

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

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

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

lemma *arcs-distinct-dest1*:
 $\text{distinct } (i \# a \# xs) \implies (b,c) \in \text{set } (\text{arcs } a \text{ } j \text{ } xs) \implies b \neq i$
 ⟨proof⟩

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

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

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

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

shows $\exists c. (b, c) \in \text{set} (\text{arcs } i j xs) \langle \text{proof} \rangle$

lemma arcs-predecessor':

assumes $(a, b) \in \text{set} (\text{arcs } i j (x \# xs)) \ (a, b) \neq (i, x)$
shows $\exists c. (c, a) \in \text{set} (\text{arcs } i j (x \# xs)) \langle \text{proof} \rangle$

lemma arcs-cases:

assumes $(a, b) \in \text{set} (\text{arcs } i j xs) \ xs \neq []$
shows $(\exists ys. xs = b \# ys \wedge a = i) \vee (\exists ys. xs = ys @ [a] \wedge b = j)$
 $\vee (\exists c d ys. (a, b) \in \text{set} (\text{arcs } c d ys) \wedge xs = c \# ys @ [d])$
 $\langle \text{proof} \rangle$

lemma arcs-cases':

assumes $(a, b) \in \text{set} (\text{arcs } i j xs) \ xs \neq []$
shows $(\exists ys. xs = b \# ys \wedge a = i) \vee (\exists ys. xs = ys @ [a] \wedge b = j) \vee$
 $(\exists ys zs. xs = ys @ a \# b \# zs)$
 $\langle \text{proof} \rangle$

lemma arcs-successor':

assumes $(a, b) \in \text{set} (\text{arcs } i j xs) \ b \neq j$
shows $\exists c. xs = [b] \wedge a = i \vee (\exists ys. xs = b \# c \# ys \wedge a = i) \vee (\exists ys.$
 $xs = ys @ [a, b] \wedge c = j)$
 $\vee (\exists ys zs. xs = ys @ a \# b \# c \# zs)$
 $\langle \text{proof} \rangle$

lemma list-last:

$xs = [] \vee (\exists y ys. xs = ys @ [y])$
 $\langle \text{proof} \rangle$

lemma arcs-predecessor'':

assumes $(a, b) \in \text{set} (\text{arcs } i j xs) \ a \neq i$
shows $\exists c. xs = [a] \vee (\exists ys. xs = a \# b \# ys) \vee (\exists ys. xs = ys @ [c, a]$
 $\wedge b = j)$
 $\vee (\exists ys zs. xs = ys @ c \# a \# b \# zs)$
 $\langle \text{proof} \rangle$

lemma arcs-ex-middle:

$\exists b. (a, b) \in \text{set} (\text{arcs } i j (ys @ a \# xs))$
 $\langle \text{proof} \rangle$

lemma arcs-ex-head:

$\exists b. (i, b) \in \text{set} (\text{arcs } i j xs)$
 $\langle \text{proof} \rangle$

2.5.1 Successive

fun *successive* **where**

successive - [] = True |
successive P [-] = True |
successive P (x # y # xs) \longleftrightarrow \neg P y \wedge *successive* P xs \vee \neg P x \wedge
successive P (y # xs)

lemma \neg *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 \neg *successive* (λ x. x > (0 :: nat)) [Suc 0, 0, Suc 0, Suc 0] <proof>

lemma \neg *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 \neg *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) \implies \neg P x \implies *successive* P
xs
<proof>

lemma *successive-step-2*: *successive* P (x # y # xs) \implies \neg P y \implies *suc-*
cessive P xs
<proof>

lemma *successive-stepI*:

successive P xs \implies \neg P x \implies *successive* P (x # xs)
<proof>

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

lemma *successive-end-1*:

successive P xs \implies \neg P x \implies *successive* P (xs @ [x])
<proof>

lemma *successive-ends-1*:

successive P xs \implies \neg P x \implies *successive* P ys \implies *successive* P (xs @ x
ys)
<proof>

lemma *successive-ends-1'*:

successive P xs \implies \neg P x \implies P y \implies \neg P z \implies *successive* P ys \implies
successive P (xs @ x # y # z # ys)

$\langle \text{proof} \rangle$

lemma *successive-end-2:*

$$\text{successive } P \text{ } xs \implies \neg P \text{ } x \implies \text{successive } P \text{ } (xs @ [x,y])$$

$\langle \text{proof} \rangle$

lemma *successive-end-2':*

$$\text{successive } P \text{ } (xs @ [x]) \implies \neg P \text{ } x \implies \text{successive } P \text{ } (xs @ [x,y])$$

$\langle \text{proof} \rangle$

lemma *successive-end-3:*

$$\text{successive } P \text{ } (xs @ [x]) \implies \neg P \text{ } x \implies P \text{ } y \implies \neg P \text{ } z \implies \text{successive } P \text{ } (xs @ [x,y,z])$$

$\langle \text{proof} \rangle$

lemma *successive-step-rev:*

$$\text{successive } P \text{ } (xs @ [x]) \implies \neg P \text{ } x \implies \text{successive } P \text{ } xs$$

$\langle \text{proof} \rangle$

lemma *successive-glue:*

$$\text{successive } P \text{ } (zs @ [z]) \implies \text{successive } P \text{ } (x \# xs) \implies \neg P \text{ } z \vee \neg P \text{ } x \implies \text{successive } P \text{ } (zs @ [z] @ x \# xs)$$

$\langle \text{proof} \rangle$

lemma *successive-glue':*

$$\begin{aligned} & \text{successive } P \text{ } (zs @ [y]) \wedge \neg P \text{ } z \vee \text{successive } P \text{ } zs \wedge \neg P \text{ } y \\ & \implies \text{successive } P \text{ } (x \# xs) \wedge \neg P \text{ } w \vee \text{successive } P \text{ } xs \wedge \neg P \text{ } x \\ & \implies \neg P \text{ } z \vee \neg P \text{ } w \implies \text{successive } P \text{ } (zs @ y \# z \# w \# x \# xs) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *successive-dest-head:*

$$xs = w \# x \# ys \implies \text{successive } P \text{ } xs \implies \text{successive } P \text{ } (x \# xs) \wedge \neg P \text{ } w \vee \text{successive } P \text{ } xs \wedge \neg P \text{ } x$$

$\langle \text{proof} \rangle$

lemma *successive-dest-tail:*

$$\begin{aligned} & xs = zs @ [y,z] \implies \text{successive } P \text{ } xs \\ & \implies \text{successive } P \text{ } (zs @ [y]) \wedge \neg P \text{ } z \vee \text{successive } P \text{ } zs \wedge \neg P \text{ } y \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *successive-split:*

$$xs = ys @ zs \implies \text{successive } P \text{ } xs \implies \text{successive } P \text{ } ys \wedge \text{successive } P \text{ } zs$$

$\langle \text{proof} \rangle$

lemma *successive-decomp*:

$xs = x \# ys @ zs @ [z] \implies successive\ P\ xs \implies \neg P\ x \vee \neg P\ z \implies$
 $successive\ P\ (zs @ [z] @ (x \# ys))$

$\langle proof \rangle$

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 $\exists c. (xs = [a] \wedge c = i \vee (\exists ys. xs = a \# b \# ys \wedge c = i) \vee (\exists$
 $ys. xs = ys @ [c, a] \wedge b = j)$

$\vee (\exists ys\ zs. xs = ys @ c \# a \# b \# zs)) \wedge \neg P\ (c, a)$

$\langle proof \rangle$

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 $\exists c. (xs = [b] \wedge c = j \vee (\exists ys. xs = b \# c \# ys) \vee (\exists ys. xs = ys$
 $@ [a, b] \wedge c = j)$

$\vee (\exists ys\ zs. xs = ys @ a \# b \# c \# zs)) \wedge \neg P\ (b, c)$

$\langle proof \rangle$

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 $\exists ys. len\ M\ i\ j\ ys \leq len\ M\ i\ j\ xs \wedge set\ ys \subseteq \{0..n\}$

$\wedge successive\ (\lambda\ (a, b). M\ a\ b = A\ a\ b)\ (arcs\ i\ j\ ys)$

$\langle proof \rangle$

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 $\exists ys. len\ M\ i\ j\ ys \leq len\ M\ i\ j\ xs \wedge set\ ys \subseteq set\ xs$

$\wedge successive\ (\lambda\ (a, b). M\ a\ b = A\ a\ b)\ (arcs\ i\ j\ ys)$

$\wedge \text{distinct } ys \wedge i \notin \text{set } ys \wedge j \notin \text{set } ys$

$\langle \text{proof} \rangle$

lemma *successive-snd-last*: $\text{successive } P (xs @ [x, y]) \implies P y \implies \neg P x$

$\langle \text{proof} \rangle$

lemma *canonical-shorten-rotate-neg-cycle*:

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

$\langle \text{proof} \rangle$

lemma *successive-arcs-extend-last*:

$\text{successive } P (\text{arcs } i j xs) \implies \neg P (i, \text{hd } xs) \vee \neg P (\text{last } xs, j) \implies xs \neq []$
 $\implies \text{successive } P (\text{arcs } i j xs @ [(i, \text{hd } xs)])$

$\langle \text{proof} \rangle$

lemma *cycle-rotate-arcs*:

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

lemma *cycle-rotate-len-arcs-successive*:

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

$\langle \text{proof} \rangle$

lemma *successive-successors*:

$xs = ys @ a \# b \# c \# zs \implies \text{successive } P (\text{arcs } i j xs) \implies \neg P (a, b)$

$\vee \neg P (b, c)$
 $\langle \text{proof} \rangle$

lemma *successive-successors'*:

$xs = ys @ a \# b \# zs \implies \text{successive } P \ xs \implies \neg P \ a \vee \neg P \ b$
 $\langle \text{proof} \rangle$

lemma *cycle-rotate-len-arcs-successive'*:

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

lemma *cycle-rotate-3*:

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

lemma *cycle-rotate-3'*:

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

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 ⇒ 't

definition cval-add :: ('c,'t) cval ⇒ 't::plus ⇒ ('c,'t) cval (**infixr** ⟨⊕⟩ 64)
where

$$u \oplus d = (\lambda x. u x + d)$$

definition zone-delay :: ('c, ('t::time)) zone ⇒ ('c, 't) zone
(⟨-↑⟩ [71] 71)

where

$$Z^\uparrow = \{u \oplus d \mid u d. u \in Z \wedge d \geq (0::'t)\}$$

fun clock-set :: 'c list ⇒ 't::time ⇒ ('c,'t) cval ⇒ ('c,'t) cval
where

$$\begin{aligned} \text{clock-set } [] - u &= u \mid \\ \text{clock-set } (c\#cs) t u &= (\text{clock-set } cs t u)(c:=t) \end{aligned}$$

abbreviation clock-set-abbrev :: 'c list ⇒ 't::time ⇒ ('c,'t) cval ⇒ ('c,'t) cval
(⟨[-→-]⟩ [65,65,65] 65)

where

$$[r \rightarrow t]u \equiv \text{clock-set } r t u$$

definition zone-set :: ('c, 't::time) zone ⇒ 'c list ⇒ ('c, 't) zone
(⟨[- → 0]⟩ [71] 71)

where

$$\text{zone-set } Z r = \{[r \rightarrow (0::'t)]u \mid u . u \in Z\}$$

lemma clock-set-set[simp]:

$$\begin{aligned} ([r \rightarrow d]u) c = d &\text{ if } c \in \text{set } r \\ \langle \text{proof} \rangle \end{aligned}$$

lemma clock-set-id[simp]:

$$\begin{aligned} ([r \rightarrow d]u) c = u c &\text{ if } c \notin \text{set } r \\ \langle \text{proof} \rangle \end{aligned}$$

definition DBM-zone-repr :: ('t::time) DBM ⇒ ('c ⇒ nat) ⇒ nat ⇒ ('c, 't :: time) zone
(⟨[-]_,_⟩ [72,72,72] 72)

where

$$[M]_{v,n} = \{u . \text{DBM-val-bounded } v u M n\}$$

lemma *dbm-entry-val-mono1*:

dbm-entry-val u (Some c) (Some c') b \implies b \preceq b' \implies dbm-entry-val u (Some c) (Some c') b'
<proof>

lemma *dbm-entry-val-mono2*:

dbm-entry-val u None (Some c) b \implies b \preceq b' \implies dbm-entry-val u None (Some c) b'
<proof>

lemma *dbm-entry-val-mono3*:

dbm-entry-val u (Some c) None b \implies b \preceq b' \implies 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*:

$\forall i j. i \leq n \longrightarrow j \leq n \longrightarrow M i j \preceq M' i j \implies u \in [M]_{v,n} \implies u \in [M']_{v,n}$
<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 ($\lambda x y. \text{if } m = x \wedge n = y \text{ then } v \text{ else } M x y$)

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* *M* 1 *f* (*Suc* *m*) *n* *N* (*Suc* *m*) *j* = *DBM-upd* *M* 1 *f* (*Suc* *m*)
j *N* (*Suc* *m*) *j*

⟨*proof*⟩

lemma *upd-2*:

assumes $i \leq m$

shows *DBM-upd* *M* 1 *f* (*Suc* *m*) *n* *N* *i* *j* = *DBM-upd* *M* 1 *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* *M* 1 *f* *m* *n* *N*) *i* *j* = (*DBM-upd* *M* 1 *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* *M* 1 *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'*:

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

lemma *sum-gt-neutral-dest*:

$(a :: (('a :: \text{time}) \text{DBMEntry})) + b > 0 \implies \exists d. \text{Le } d \leq a \wedge \text{Le } (-d) \leq b$
 ⟨proof⟩

2.5.6 Negative Cycles in DBMs

lemma *DBM-val-bounded-neg-cycle1*:

fixes $i \text{ } xs$ **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: $\forall k \leq n. k > 0 \implies (\exists c. v \ c = k)$ **and** *at-most*: $i \neq 0$ *cnt* $0 \ xs \leq 1$

shows *False*

⟨proof⟩

lemma *cnt-0-I*:

$x \notin \text{set } xs \implies \text{cnt } x \ xs = 0$
 ⟨proof⟩

lemma *distinct-cnt*: *distinct* $xs \implies \text{cnt } x \ xs \leq 1$

⟨proof⟩

lemma *DBM-val-bounded-neg-cycle*:

fixes $i \text{ } xs$ **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: $\forall k \leq n. k > 0 \implies (\exists c. v \ c = k)$

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 $dbm\text{-entry}\text{-val } u \text{ None } (Some\ c) \ (len\ M\ 0\ (v\ c)\ vs) \ \langle proof \rangle$

lemma *DBM-val-bounded-len-3*:

fixes v

assumes $DBM\text{-val}\text{-bounded } v\ u\ M\ n\ v\ c1 \leq n\ v\ c2 \leq n \text{ set } vs \subseteq \{0..n\}$
 $\forall k \leq n. (\exists c. v\ c = k)$

shows $dbm\text{-entry}\text{-val } u \ (Some\ c1) \ (Some\ c2) \ (len\ M\ (v\ c1) \ (v\ c2) \ vs)$
 $\langle proof \rangle$

An equivalent way of handling $\mathbf{0}$

fun $val\text{-}0 :: ('c \Rightarrow ('a :: linordered\text{-}ab\text{-}group\text{-}add)) \Rightarrow 'c\ option \Rightarrow 'a$ **where**

$val\text{-}0\ u\ None = 0 \mid$

$val\text{-}0\ u\ (Some\ c) = u\ c$

notation $val\text{-}0 \ (\prec_{\mathbf{0}} \rightarrow [90,90] \ 90)$

lemma *dbm-entry-val-None-None[dest]*:

$dbm\text{-entry}\text{-val } u \ None \ None\ l \Longrightarrow l = \infty$

$\langle proof \rangle$

lemma *dbm-entry-val-dbm-lt*:

assumes $dbm\text{-entry}\text{-val } u\ x\ y\ l$

shows $Lt\ (u_{\mathbf{0}}\ x - u_{\mathbf{0}}\ y) \prec l$

$\langle proof \rangle$

lemma *dbm-lt-dbm-entry-val-1*:

assumes $Lt\ (u\ x) \prec l$

shows $dbm\text{-entry}\text{-val } u \ (Some\ x) \ None\ l$

$\langle proof \rangle$

lemma *dbm-lt-dbm-entry-val-2*:

assumes $Lt\ (-\ u\ x) \prec l$

shows $dbm\text{-entry}\text{-val } u \ None \ (Some\ x) \ l$

$\langle proof \rangle$

lemma *dbm-lt-dbm-entry-val-3*:

assumes $Lt\ (u\ x - u\ y) \prec l$

shows $dbm\text{-entry}\text{-val } u \ (Some\ x) \ (Some\ y) \ l$

$\langle proof \rangle$

A more uniform theorem for boundedness by paths

lemma *DBM-val-bounded-len*:

fixes v

defines $v' \equiv \lambda x. \text{if } x = \text{None} \text{ then } 0 \text{ else } v \text{ (the } x)$
assumes *DBM-val-bounded* $v \ u \ M \ n \ v' \ x \leq n \ v' \ y \leq n \ \text{set } vs \subseteq \{0..n\}$
 $\forall k \leq n. (\exists c. v \ c = k) \ x \neq \text{None} \vee y \neq \text{None}$
shows $Lt \ (u_0 \ x - u_0 \ y) \prec \text{len } M \ (v' \ x) \ (v' \ y) \ vs \ \langle \text{proof} \rangle$

2.5.7 Floyd-Warshall Algorithm Preservers Zones

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

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

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

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

2.6 The Characteristic Property of Canonical DBMs

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

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

lemma *fix-index:*

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

Putting it together lemma *FW-not-empty:*

$\text{DBM-val-bounded } v\ u\ (\text{FW } M'\ n)\ n \implies \text{DBM-val-bounded } v\ u\ M'\ n$
 $\langle \text{proof} \rangle$

lemma *fix-indices:*

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

lemma *cyc-free-obtains-valuation:*

$\text{cyc-free } M\ n \implies \forall\ c. v\ c \leq n \longrightarrow v\ c > 0 \implies \exists\ u. \text{DBM-val-bounded } v\ u\ M\ n$
 $\langle \text{proof} \rangle$

2.6.1 Floyd-Warshall and Empty DBMs

theorem *FW-detects-empty-zone:*

$\forall\ k \leq n. 0 < k \longrightarrow (\exists\ c. v\ c = k) \implies \forall\ c. v\ c \leq n \longrightarrow v\ c > 0$

$\implies [FW M n]_{v,n} = \{\} \iff (\exists i \leq n. (FW M n) i i < Le 0)$
 $\langle proof \rangle$

hide-const (open) D

2.6.2 Mixed Corollaries

lemma *cyc-free-not-empty*:

assumes *cyc-free* $M n \forall c. v c \leq n \longrightarrow 0 < v c$

shows $[(M :: ('a :: time) DBM)]_{v,n} \neq \{\}$

$\langle proof \rangle$

lemma *empty-not-cyc-free*:

assumes $\forall c. v c \leq n \longrightarrow 0 < v c [(M :: ('a :: time) DBM)]_{v,n} = \{\}$

shows $\neg \text{cyc-free } M n$

$\langle proof \rangle$

lemma *not-empty-cyc-free*:

assumes $\forall k \leq n. 0 < k \longrightarrow (\exists c. v c = k) [(M :: ('a :: time) DBM)]_{v,n} \neq \{\}$

shows *cyc-free* $M n$ $\langle proof \rangle$

lemma *neg-cycle-empty*:

assumes $\forall k \leq n. 0 < k \longrightarrow (\exists c. v c = k) \text{ set } xs \subseteq \{0..n\} i \leq n \text{ len } M i$
 $i xs < 0$

shows $[(M :: ('a :: time) DBM)]_{v,n} = \{\} \langle proof \rangle$

abbreviation *clock-numbering'* $:: ('c \Rightarrow nat) \Rightarrow nat \Rightarrow bool$

where

clock-numbering' $v n \equiv \forall c. v c > 0 \wedge (\forall x. \forall y. v x \leq n \wedge v y \leq n \wedge v$
 $x = v y \longrightarrow x = y)$

lemma *non-empty-dbm-diag-set*:

clock-numbering' $v n \implies [M]_{v,n} \neq \{\}$

$\implies [M]_{v,n} = [(\lambda i j. \text{if } i = j \text{ then } 0 \text{ else } M i j)]_{v,n}$

$\langle proof \rangle$

lemma *non-empty-cycle-free*:

assumes $[M]_{v,n} \neq \{\}$

and $\forall k \leq n. 0 < k \longrightarrow (\exists c. v c = k)$

shows *cycle-free* $M n$

$\langle proof \rangle$

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} ⊆ [M']_{v,n} [M]_{v,n} ≠ {} i ≤ n j ≤ n i ≠ j*

assumes *clock-numbering: clock-numbering' v n*
 $\forall k \leq n. 0 < k \longrightarrow (\exists c. v c = k)$

shows *M i j ≤ 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:*

$i \leq n \implies j \leq n \implies FWI\ M\ n\ k\ i\ j \leq M\ i\ j$

<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* $h \{x. h x \in F\}$
 $\langle proof \rangle$

lemma *gt-swap*:
fixes $a b c :: 't :: time$
assumes $c < a + b$
shows $c < b + a$
 $\langle proof \rangle$

lemma *le-swap*:
fixes $a b c :: 't :: time$
assumes $c \leq a + b$
shows $c \leq b + a$
 $\langle proof \rangle$

abbreviation *clock-numbering* $:: ('c \Rightarrow nat) \Rightarrow bool$
where
 $clock-numbering v \equiv \forall c. v c > 0$

lemma *DBM-triv*:
 $u \vdash_{v,n} (\lambda i j. \infty)$
 $\langle proof \rangle$

3.2 Relaxation

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

definition

$up :: ('t::linordered-cancel-ab-semigroup-add) DBM \Rightarrow 't DBM$
where
 $up M \equiv$
 $\lambda i j. \text{if } i > 0 \text{ then if } j = 0 \text{ then } \infty \text{ else } \min (dbm-add (M i 0) (M 0 j))$
 $(M i j) \text{ else } M i j$

lemma *dbm-entry-dbm-lt*:
assumes $dbm-entry-val u (Some c1) (Some c2) a a \prec b$
shows $dbm-entry-val u (Some c1) (Some c2) b$
 $\langle proof \rangle$

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* \implies $u \in ([M]_{v,n})^\dagger \implies u \in [up\ M]_{v,n}$
⟨*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 $u \in ([M]_{v,n})^\dagger$
⟨*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 $DBM\text{-reset } M \ n \ k \ d \ M' \ k \notin \text{set } xs \ i \neq k \ j \neq k \ \text{set } (i \ \# \ j \ \# \ xs) \subseteq \{0..n\}$
shows $\text{len } M' \ i \ j \ xs \leq \text{len } M \ i \ j \ xs$
 $\langle \text{proof} \rangle$

lemma *DBM-reset-neg-cycle-preservation:*

assumes $DBM\text{-reset } M \ n \ k \ d \ M' \ \text{len } M \ i \ i \ xs < Le \ 0 \ \text{set } (k \ \# \ i \ \# \ xs) \subseteq \{0..n\}$
shows $\exists j. \exists ys. \text{set } (j \ \# \ ys) \subseteq \{0..n\} \wedge \text{len } M' \ j \ j \ ys < Le \ 0$
 $\langle \text{proof} \rangle$

Implementation of DBM reset

definition

$\text{reset} :: ('t :: \{\text{linordered-cancel-ab-semigroup-add, uminus}\}) \text{ DBM} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 't \Rightarrow 't \text{ DBM}$

where

$\text{reset } M \ n \ k \ d =$
 $(\lambda \ i \ j.$
 $\quad \text{if } i = k \wedge j = 0 \text{ then } Le \ d$
 $\quad \text{else if } i = 0 \wedge j = k \text{ then } Le \ (-d)$
 $\quad \text{else if } i = k \wedge j \neq k \text{ then } \infty$
 $\quad \text{else if } i \neq k \wedge j = k \text{ then } \infty$
 $\quad \text{else if } i = k \wedge j = k \text{ then } M \ k \ k$
 $\quad \text{else } \min (\text{dbm-add } (M \ i \ k) (M \ k \ j)) (M \ i \ j)$
 $\quad)$

fun

$\text{reset}' ::$
 $('t :: \{\text{linordered-cancel-ab-semigroup-add, uminus}\}) \text{ DBM}$
 $\Rightarrow \text{nat} \Rightarrow 'c \ \text{list} \Rightarrow ('c \Rightarrow \text{nat}) \Rightarrow 't \Rightarrow 't \text{ DBM}$

where

$\text{reset}' \ M \ n \ [] \ v \ d = M \ |$
 $\text{reset}' \ M \ n \ (c \ \# \ cs) \ v \ d = \text{reset} (\text{reset}' \ M \ n \ cs \ v \ d) \ n \ (v \ c) \ d$

lemma *DBM-reset-reset:*

$0 < k \implies k \leq n \implies DBM\text{-reset } M \ n \ k \ d \ (\text{reset } M \ n \ k \ d)$
 $\langle \text{proof} \rangle$

lemma *DBM-reset-complete:*

assumes $\text{clock-numbering}' \ v \ n \ v \ c \leq n \ DBM\text{-reset } M \ n \ (v \ c) \ d \ M'$
 $DBM\text{-val-bounded } v \ u \ M \ n$
shows $DBM\text{-val-bounded } v \ (u(c := d)) \ M' \ n$
 $\langle \text{proof} \rangle$

lemma *DBM-reset-sound-empty*:

assumes $\text{clock-numbering}' v n v c \leq n$ $\text{DBM-reset } M n (v c) d M'$
 $\forall u . \neg \text{DBM-val-bounded } v u M' n$

shows $\neg \text{DBM-val-bounded } v u M n$

<proof>

lemma *DBM-reset-diag-preservation*:

$\forall k \leq n. M' k k \leq 0$ **if** $\forall k \leq n. M k k \leq 0$ $\text{DBM-reset } M n i d M'$

<proof>

lemma *FW-diag-preservation*:

$\forall k \leq n. M k k \leq 0 \implies \forall k \leq n. (\text{FW } M n) k k \leq 0$

<proof>

lemma *DBM-reset-not-cyc-free-preservation*:

assumes $\neg \text{cyc-free } M n$ $\text{DBM-reset } M n k d M' k \leq n$

shows $\neg \text{cyc-free } M' n$

<proof>

lemma *DBM-reset-complete-empty'*:

assumes $\forall k \leq n. k > 0 \implies (\exists c. v c = k)$ $\text{clock-numbering } v k \leq n$
 $\text{DBM-reset } M n k d M' \forall u . \neg \text{DBM-val-bounded } v u M n$

shows $\neg \text{DBM-val-bounded } v u M' n$

<proof>

lemma *DBM-reset-complete-empty*:

assumes $\forall k \leq n. k > 0 \implies (\exists c. v c = k)$ $\text{clock-numbering } v$

$\text{DBM-reset } (\text{FW } M n) n (v c) d M' \forall u . \neg \text{DBM-val-bounded } v u$

$(\text{FW } M n) n$

shows $\neg \text{DBM-val-bounded } v u M' n$

<proof>

lemma *DBM-reset-complete-empty1*:

assumes $\forall k \leq n. k > 0 \implies (\exists c. v c = k)$ $\text{clock-numbering } v$

$\text{DBM-reset } (\text{FW } M n) n (v c) d M' \forall u . \neg \text{DBM-val-bounded } v u$

$M n$

shows $\neg \text{DBM-val-bounded } v u M' n$

<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 $\forall k \leq n. k > 0 \implies (\exists c. v c = k)$ $\text{clock-numbering}' v n v c \leq n$

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

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 $[FW\ M\ n]_{v,n} = \{\} \longleftrightarrow [M']_{v,n} = \{\}$
 \langle *proof* \rangle

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 $[M]_{v,n} = \{\} \longleftrightarrow [M']_{v,n} = \{\}$
 \langle *proof* \rangle

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* $v\ (u(c := d'))\ M\ n$
 \langle *proof* \rangle

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 $u\ c = d$
 \langle *proof* \rangle

lemma *DBM-reset-sound''*:

fixes $M\ v\ c\ n\ d$
defines $M' \equiv \text{reset } M\ n\ (v\ c)\ d$
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* $v\ (u(c := d'))\ M\ n$
 \langle *proof* \rangle

lemma *DBM-reset-sound*:

fixes $M\ v\ c\ n\ d$
defines $M' \equiv \text{reset } M\ n\ (v\ c)\ d$
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** $u(c := d') \in [M]_{v,n}$
 \langle *proof* \rangle

lemma *DBM-reset'-complete'*:

assumes *DBM-val-bounded* v u M n *clock-numbering'* v n $\forall c \in \text{set } cs. v$
 $c \leq n$
shows $\exists u'. \text{DBM-val-bounded } v$ u' $(\text{reset}' M n cs v d) n$
 $\langle \text{proof} \rangle$

lemma *DBM-reset'-complete*:

assumes *DBM-val-bounded* v u M n *clock-numbering'* v n $\forall c \in \text{set } cs. v$
 $c \leq n$
shows *DBM-val-bounded* v $([cs \rightarrow d]u)$ $(\text{reset}' M n cs v d) n$
 $\langle \text{proof} \rangle$

lemma *DBM-reset'-sound-empty*:

assumes *clock-numbering'* v n $\forall c \in \text{set } cs. v$ $c \leq n$
 $\forall u. \neg \text{DBM-val-bounded } v$ u $(\text{reset}' M n cs v d) n$
shows $\neg \text{DBM-val-bounded } v$ u M n
 $\langle \text{proof} \rangle$

fun *set-clocks* :: $'c$ list \Rightarrow $'t::\text{time}$ list \Rightarrow (c,t) cval \Rightarrow (c,t) cval

where

set-clocks [] - $u = u$ |
set-clocks - [] $u = u$ |
set-clocks $(c\#cs)$ $(t\#ts)$ $u = (\text{set-clocks } cs ts (u(c:=t)))$

lemma *DBM-reset'-sound'*:

fixes M v c n d cs
assumes *clock-numbering'* v n $\forall c \in \text{set } cs. v$ $c \leq n$
 $\text{DBM-val-bounded } v$ u $(\text{reset}' M n cs v d) n$ $\text{DBM-val-bounded } v$ u''
 M n
shows $\exists ts. \text{DBM-val-bounded } v$ $(\text{set-clocks } cs ts u)$ M n
 $\langle \text{proof} \rangle$

lemma *DBM-reset'-resets*:

fixes M v c n d cs
assumes $\forall k \leq n. k > 0 \longrightarrow (\exists c. v c = k)$ *clock-numbering'* v n $\forall c \in \text{set}$
 $cs. v c \leq n$
 $\text{DBM-val-bounded } v$ u $(\text{reset}' M n cs v d) n$
shows $\forall c \in \text{set } cs. u c = d$
 $\langle \text{proof} \rangle$

lemma *DBM-reset'-resets'*:

fixes M :: $(t :: \text{time})$ *DBM* **and** v c n d cs
assumes *clock-numbering'* v n $\forall c \in \text{set } cs. v$ $c \leq n$ *DBM-val-bounded* v

u ($reset' M n cs v d$) n
 $DBM\text{-val-bounded } v u'' M n$
shows $\forall c \in set\ cs. u\ c = d$
 $\langle proof \rangle$

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 $reset' M n cs v d\ k\ k < 0\ \langle proof \rangle$

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

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

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 $[M]_{v,n} = \{\} \longleftrightarrow [reset' (FW\ M\ n)\ n\ cs\ v\ d]_{v,n} = \{\}$
 $\langle proof \rangle$

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 $[M]_{v,n} = \{\} \longleftrightarrow [reset' M n cs v d]_{v,n} = \{\}$
 $\langle proof \rangle$

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 $\exists ts. set\ clocks\ cs\ ts\ u \in [M]_{v,n}$
 $\langle proof \rangle$

3.5 Misc Preservation Lemmas

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

$\langle \text{proof} \rangle$

lemma *sum-not-inf-dest*:

assumes $a + b \neq (\infty :: - \text{DBMEntry})$

shows $a \neq (\infty :: - \text{DBMEntry}) \wedge b \neq (\infty :: - \text{DBMEntry})$

$\langle \text{proof} \rangle$

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* $(a + b) \in \mathbb{Z}$

$\langle \text{proof} \rangle$

lemma *int-fw-upd*:

$\forall i \leq n. \forall j \leq n. m \ i \ j \neq \infty \longrightarrow \text{get-const} (m \ i \ j) \in \mathbb{Z} \implies k \leq n \implies i \leq n \implies j \leq n$

$\implies i' \leq n \implies j' \leq n \implies (\text{fw-upd } m \ k \ i \ j \ i' \ j') \neq \infty$

$\implies \text{get-const} (\text{fw-upd } m \ k \ i \ j \ i' \ j') \in \mathbb{Z}$

$\langle \text{proof} \rangle$

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

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

lemma *dbm-intI*:

dbm-int-all $M \implies \text{dbm-int } M \ n$

$\langle \text{proof} \rangle$

lemma *fwi-int-preservation*:

dbm-int $(\text{fwi } M \ n \ k \ i \ j) \ n$ **if** *dbm-int* $M \ n \ k \leq n$

$\langle \text{proof} \rangle$

lemma *fw-int-preservation*:

dbm-int $(\text{fw } M \ n \ k) \ n$ **if** *dbm-int* $M \ n \ k \leq n$

$\langle \text{proof} \rangle$

lemma *FW-int-preservation*:

assumes *dbm-int* $M \ n$

shows *dbm-int* $(\text{FW } M \ n) \ n$

$\langle \text{proof} \rangle$

lemma *FW-int-all-preservation*:

assumes *dbm-int-all* M

shows $dbm\text{-int}\text{-all}$ (FW M n)
 $\langle proof \rangle$

lemma *And-int-all-preservation[intro]:*
assumes $dbm\text{-int}\text{-all}$ $M1$ $dbm\text{-int}\text{-all}$ $M2$
shows $dbm\text{-int}\text{-all}$ (And $M1$ $M2$)
 $\langle proof \rangle$

lemma *And-int-preservation:*
assumes $dbm\text{-int}$ $M1$ n $dbm\text{-int}$ $M2$ n
shows $dbm\text{-int}$ (And $M1$ $M2$) n
 $\langle proof \rangle$

lemma *up-int-all-preservation:*
 $dbm\text{-int}\text{-all}$ ($M :: ('t :: \{time, ring-1\})$ DBM) \implies $dbm\text{-int}\text{-all}$ (up M)
 $\langle proof \rangle$

lemma *up-int-preservation:*
 $dbm\text{-int}$ ($M :: ('t :: \{time, ring-1\})$ DBM) $n \implies dbm\text{-int}$ (up M) n
 $\langle proof \rangle$

lemma *DBM-reset-int-preservation':*
assumes $dbm\text{-int}$ M n $DBM\text{-reset}$ M n k d M' $d \in \mathbb{Z}$ $k \leq n$
shows $dbm\text{-int}$ M' n
 $\langle proof \rangle$

lemma *DBM-reset-int-preservation:*
fixes $M :: ('t :: \{time, ring-1\})$ DBM
assumes $dbm\text{-int}$ M n $d \in \mathbb{Z}$ $0 < k$ $k \leq n$
shows $dbm\text{-int}$ ($reset$ M n k d) n
 $\langle proof \rangle$

lemma *DBM-reset-int-all-preservation:*
fixes $M :: ('t :: \{time, ring-1\})$ DBM
assumes $dbm\text{-int}\text{-all}$ M $d \in \mathbb{Z}$
shows $dbm\text{-int}\text{-all}$ ($reset$ M n k d)
 $\langle proof \rangle$

lemma *DBM-reset'-int-all-preservation:*
fixes $M :: ('t :: \{time, ring-1\})$ DBM
assumes $dbm\text{-int}\text{-all}$ M $d \in \mathbb{Z}$
shows $dbm\text{-int}\text{-all}$ ($reset'$ M n cs v d) $\langle proof \rangle$

lemma *DBM-reset'-int-preservation*:
fixes $M :: ('t :: \{time, ring-1\}) DBM$
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*:
 $\forall c. c \notin set\ cs \longrightarrow (set-clocks\ cs\ ts\ u)\ c = u\ c$
<proof>

lemma *reset-set*:
assumes $\forall c \in set\ cs. u\ c = d$
shows $[cs \rightarrow d](set-clocks\ cs\ ts\ u) = u$
<proof>

3.5.1 Unused theorems

lemma *canonical-cyc-free*:
 $canonical\ M\ n \implies \forall i \leq n. M\ i\ i \geq 0 \implies cyc-free\ M\ n$
<proof>

lemma *canonical-cyc-free2*:
 $canonical\ M\ n \implies cyc-free\ M\ n \iff (\forall i \leq n. M\ i\ i \geq 0)$
<proof>

lemma *DBM-reset'-diag-preservation*:
fixes $M :: ('t :: time) DBM$
assumes $\forall k \leq n. M\ k\ k \leq 0\ clock-numbering\ v \ \forall c \in set\ cs. v\ c \leq n$
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 \Rightarrow 't \Rightarrow 't DBMEntry
where
norm-upper e t = (if Le t < e then ∞ else e)

fun *norm-lower* :: ('t::linorder) DBMEntry \Rightarrow 't \Rightarrow '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 $\mathbf{0}$ 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 \Rightarrow (nat \Rightarrow 't) \Rightarrow nat \Rightarrow 't DBM

where

norm M k n \equiv $\lambda i j$.
 let ub = if i > 0 then k i else 0 in
 let lb = if j > 0 then - k j else 0 in
 if i \leq n \wedge j \leq n then
 if i \neq 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 \Rightarrow (nat \Rightarrow 't) \Rightarrow (nat \Rightarrow 't) \Rightarrow nat \Rightarrow 't DBM

where

extra-lu M l u n \equiv $\lambda i j$.
 let ub = if i > 0 then l i else 0 in
 let lb = if j > 0 then - u j else 0 in
 if i \leq n \wedge j \leq n then
 if i \neq j then *norm-lower* (*norm-upper* (M i j) ub) lb else *norm-diag*
 (M i j)

extra-lup-lower-bounds: $\forall i \leq n. i > 0 \longrightarrow \text{extra-lup } M L U n 0 i \leq 0$
 ⟨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 *extra-lu* $M l u n i j \leq \text{extra-lup } M l u n i j$

⟨proof⟩

lemma *extra-lu-sub-extra-lup*:

assumes *canonical*: *canonical* $M n$ **and** *canonical-lower-bounds*: $\forall i \leq n. i > 0 \longrightarrow M 0 i \leq 0$

shows $[\text{extra-lu } M L U n]_{v,n} \subseteq [\text{extra-lup } M L U n]_{v,n}$

⟨proof⟩

3.6.3 Extrapolations are widening operators

lemma *extra-lu-mono*:

assumes $\forall c. v c > 0 u \in [M]_{v,n}$

shows $u \in [\text{extra-lu } M L U n]_{v,n}$ (**is** $u \in [?M2]_{v,n}$)

⟨proof⟩

lemma *norm-mono*:

assumes $\forall c. v c > 0 u \in [M]_{v,n}$

shows $u \in [\text{norm } M k n]_{v,n}$

⟨proof⟩

3.6.4 Finiteness of extrapolations

abbreviation *dbm-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*:

dbm-default $M n \implies \text{dbm-default } (\text{norm } M k n) n$

⟨proof⟩

lemma *extra-lu-default-preservation*:

dbm-default $M n \implies \text{dbm-default } (\text{extra-lu } M L U n) n$

⟨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 $L U :: \text{nat} \Rightarrow \text{nat}$
shows $\text{finite} \{ \text{extra-lu } M L U n \mid M. \text{dbm-default } M n \}$
 $\langle \text{proof} \rangle$

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

end

4 DBMs as Constraint Systems

theory *DBM-Constraint-Systems*

imports

DBM-Operations

DBM-Normalization

begin

4.1 Misc

lemma *Max-le-MinI*:
assumes $\text{finite } S \text{ finite } T \ S \neq \{\} \ T \neq \{\} \ \wedge x y. x \in S \implies y \in T \implies x \leq y$
shows $\text{Max } S \leq \text{Min } T \ \langle \text{proof} \rangle$

lemma *Min-insert-cases*:
assumes $x = \text{Min} (\text{insert } a \ S) \ \text{finite } S$
obtains $(\text{default}) \ x = a \mid (\text{elem}) \ x \in S$
 $\langle \text{proof} \rangle$

lemma *cval-add-simp[simp]*:
 $(u \oplus d) \ x = u \ x + d$
 $\langle \text{proof} \rangle$

lemmas $[\text{simp}] = \text{any-le-inf}$

lemma *Le-in-between*:
assumes $a < b$
obtains d **where** $a \leq \text{Le } d \ \text{Le } d \leq b$
 $\langle \text{proof} \rangle$

lemma *DBMEntry-le-to-sum*:

fixes $e e' :: 't :: \text{time DBMEntry}$
assumes $e' \neq \infty \ e \leq e'$
shows $-e' + e \leq 0$
 $\langle \text{proof} \rangle$

lemma *DBMEntry-le-add*:
fixes $a b c :: 't :: \text{time DBMEntry}$
assumes $a \leq b + c \ c \neq \infty$
shows $-c + a \leq b$
 $\langle \text{proof} \rangle$

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

4.2 Definition and Semantics of Constraint Systems

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

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

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

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

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

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

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

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

definition

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

lemma *constr-sem-less-eq-iff*:

$u \models_c Lower\ x\ e \iff Le\ (-u\ x) \leq e$
 $u \models_c Upper\ x\ e \iff Le\ (u\ x) \leq e$
 $u \models_c Diff\ x\ y\ e \iff Le\ (u\ x - u\ y) \leq e$
 <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*]:

$u \models_c Upper\ x\ \infty \iff u \models_c Lower\ y\ \infty \iff u \models_c Diff\ x\ y\ \infty$
 <proof>

lemma *cs-sem-antimono*:

assumes $cs \subseteq cs' \iff u \models_{cs} cs'$

shows $u \models_{cs} cs$

<proof>

lemma *cs-equivD*[*intro, dest*]:

assumes $u \models_{cs} cs \iff cs \equiv_{cs} cs'$

shows $u \models_{cs} cs'$

<proof>

lemma *cs-equiv-sym*:

$cs \equiv_{cs} cs' \iff cs' \equiv_{cs} cs$

<proof>

lemma *cs-equiv-union*:

$cs \equiv_{cs} cs \cup cs' \iff cs \equiv_{cs} cs'$

<proof>

lemma *cs-equiv-alt-def*:

$cs \equiv_{cs} cs' \iff (\forall c. cs \models c \iff cs' \models c)$

<proof>

lemma *closure-equiv*:

$\text{closure } cs \equiv_{cs} cs$

$\langle \text{proof} \rangle$

lemma *closure-superset*:

$cs \subseteq \text{closure } cs$

$\langle \text{proof} \rangle$

lemma *bot-cs-empty*:

$\neg (u :: ('c \Rightarrow 't :: \text{linordered-ab-group-add})) \models_{cs} \text{bot-cs}$

$\langle \text{proof} \rangle$

lemma *finite-bot-cs*:

$\text{finite } \text{bot-cs}$

$\langle \text{proof} \rangle$

definition *cs-vars where*

$\text{cs-vars } cs = \bigcup (\text{set1-constr } ' cs)$

definition *map-cs-vars where*

$\text{map-cs-vars } v cs = \text{map-constr } v \text{ id } ' cs$

lemma *constr-sem-rename-vars*:

assumes $\text{inj-on } v S \text{ set1-constr } c \subseteq S$

shows $(u \text{ o inv-into } S v) \models_c \text{map-constr } v \text{ id } c \longleftrightarrow u \models_c c$

$\langle \text{proof} \rangle$

lemma *cs-sem-rename-vars*:

assumes $\text{inj-on } v (\text{cs-vars } cs)$

shows $(u \text{ o inv-into } (\text{cs-vars } cs) v) \models_{cs} \text{map-cs-vars } v cs \longleftrightarrow u \models_{cs} cs$

$\langle \text{proof} \rangle$

4.3 Conversion of DBMs to Constraint Systems and Back

definition *dbm-to-cs* $:: \text{nat} \Rightarrow ('x \Rightarrow \text{nat}) \Rightarrow ('v :: \{\text{linorder}, \text{zero}\}) \text{ DBM}$
 $\Rightarrow ('x, 'v) cs$ **where**

$\text{dbm-to-cs } n v M \equiv \text{if } M \text{ } 0 \text{ } 0 < 0 \text{ then } \text{bot-cs} \text{ else}$

$\{\text{Lower } x (M \text{ } 0 (v x)) \mid x. v x \leq n\} \cup$

$\{\text{Upper } x (M (v x) \text{ } 0) \mid x. v x \leq n\} \cup$

$\{\text{Diff } x y (M (v x) (v y)) \mid x y. v x \leq n \wedge v y \leq n\}$

lemma *dbm-entry-val-Lower-iff*:

$\text{dbm-entry-val } u \text{ None } (\text{Some } x) e \longleftrightarrow u \models_c \text{Lower } x e$

$\langle \text{proof} \rangle$

lemma *dbm-entry-val-Upper-iff*:

$dbm\text{-entry-val } u \text{ (Some } x) \text{ None } e \longleftrightarrow u \models_c \text{Upper } x \ e$
 $\langle \text{proof} \rangle$

lemma *dbm-entry-val-Diff-iff*:

$dbm\text{-entry-val } u \text{ (Some } x) \text{ (Some } y) \ e \longleftrightarrow u \models_c \text{Diff } x \ y \ e$
 $\langle \text{proof} \rangle$

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*:

$u \vdash_{v,n} M \longleftrightarrow u \models_{cs} dbm\text{-to-cs } n \ v \ M$
 $\langle \text{proof} \rangle$

definition

$cs\text{-to-dbm } v \ cs \equiv \text{if } (\forall u. \neg u \models_{cs} \ cs) \text{ then } (\lambda _ \ . \ Lt \ 0) \text{ else } ($
 $\lambda i \ j.$
 $\text{if } i = 0 \text{ then}$
 $\text{if } j = 0 \text{ then}$
 $\quad Le \ 0$
 else
 $\quad Min \ (\text{insert } \infty \ \{e. \exists x. \text{Lower } x \ e \in cs \wedge v \ x = j\})$
 else
 $\text{if } j = 0 \text{ then}$
 $\quad Min \ (\text{insert } \infty \ \{e. \exists x. \text{Upper } x \ e \in cs \wedge v \ x = i\})$
 else
 $\quad Min \ (\text{insert } \infty \ \{e. \exists x \ y. \text{Diff } x \ y \ e \in cs \wedge v \ x = i \wedge v \ y = j\})$
 $\left. \right)$

lemma *finite-dbm-to-cs*:

assumes *finite* $\{x. v \ x \leq n\}$
shows *finite* $(dbm\text{-to-cs } n \ v \ M)$
 $\langle \text{proof} \rangle$

lemma *empty-dbm-empty*:

$u \vdash_{v,n} (\lambda _ \ . \ Lt \ 0) \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

fun *expr-of-constr* **where**

expr-of-constr $(\text{Lower } _ \ e) = e \mid$

$expr\text{-of-constr } (Upper - e) = e \mid$
 $expr\text{-of-constr } (Diff - - e) = e$

lemma *cs-to-dbm1*:

assumes $\forall x \in cs\text{-vars } cs. v x > 0 \wedge v x \leq n$ *finite cs*

assumes $u \vdash_{v,n} cs\text{-to-dbm } v cs$

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 $u \vdash_{v,n} cs\text{-to-dbm } v cs$

<proof>

theorem *cs-to-dbm-correct*:

assumes $\forall x \in cs\text{-vars } cs. v x \leq n \forall x. v x \leq n \longrightarrow v x > 0$

$\forall x y. v x \leq n \wedge v y \leq n \wedge v x = v y \longrightarrow x = y$

finite cs

shows $u \vdash_{v,n} cs\text{-to-dbm } v cs \longleftrightarrow u \models_{cs} cs$

<proof>

corollary *cs-to-dbm-correct'*:

assumes

bij-betw $v (cs\text{-vars } cs) \{1..n\} \forall x. v x \leq n \longrightarrow v x > 0 \forall x. x \notin cs\text{-vars}$
 $cs \longrightarrow v x > n$

finite cs

shows $u \vdash_{v,n} cs\text{-to-dbm } v cs \longleftrightarrow u \models_{cs} cs$

<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

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

lemma *Lower-shiftI*:

$u \oplus d \models_c Lower\ x\ e$ **if** $u \models_c Lower\ x\ e$ ($d :: 't :: linordered\text{-ab-group-add}$)
 ≥ 0

$\langle \text{proof} \rangle$

lemma *Upper-shiftI*:

$u \oplus d \models_c \text{Upper } x \ e \ \mathbf{if} \ u \models_c \text{Upper } x \ e \ (d :: 't :: \text{linordered-ab-group-add})$
 ≤ 0
 $\langle \text{proof} \rangle$

lemma *Diff-shift*:

$u \oplus d \models_c \text{Diff } x \ y \ e \longleftrightarrow u \models_c \text{Diff } x \ y \ e \ \mathbf{for} \ d :: 't :: \text{linordered-ab-group-add}$
 $\langle \text{proof} \rangle$

lemma *up-cs-complete*:

$u \oplus d \models_{cs} \text{up-cs } cs \ \mathbf{if} \ u \models_{cs} cs \ d \geq 0 \ \mathbf{for} \ d :: 't :: \text{linordered-ab-group-add}$
 $\langle \text{proof} \rangle$

definition

lower-upper-closed $cs \equiv \forall x \ y \ e \ e'.$

$\text{Lower } x \ e \in cs \wedge \text{Upper } y \ e' \in cs \longrightarrow (\exists e''. \text{Diff } y \ x \ e'' \in cs \wedge e'' \leq e + e')$

lemma *up-cs-sound*:

assumes $u \models_{cs} \text{up-cs } cs$ *lower-upper-closed* cs *finite* cs

obtains u' **and** $d :: 't :: \text{time}$ **where** $d \geq 0$ $u' \models_{cs} cs$ $u = u' \oplus d$

$\langle \text{proof} \rangle$

Note that if we compare this proof to $\llbracket \forall c. 0 < ?v \ c \wedge (\forall x \ y. ?v \ x \leq ?n \wedge ?v \ y \leq ?n \wedge ?v \ x = ?v \ y \longrightarrow x = y); ?u \in [up \ ?M]_{?v, ?n} \Longrightarrow ?u \in [?M]_{?v, ?n}^\uparrow \rrbracket$, 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 simplify some other proofs. Also, note that the final theorem we obtain below (*DBM-up-correct*) is slightly stronger than what we would get with $\llbracket \forall c. 0 < ?v \ c \wedge (\forall x \ y. ?v \ x \leq ?n \wedge ?v \ y \leq ?n \wedge ?v \ x = ?v \ y \longrightarrow x = y); ?u \in [up \ ?M]_{?v, ?n} \Longrightarrow ?u \in [?M]_{?v, ?n}^\uparrow \rrbracket$. Finally, note that a more elegant definition of *lower-upper-closed* would probably be: *definition* *lower-upper-closed* $cs \equiv \forall x \ y \ e \ e'. cs \models \text{Lower } x \ e \wedge cs \models \text{Upper } y \ e' \longrightarrow (\exists e''. cs \models \text{Diff } y \ x \ e'' \wedge e'' \leq e + e')$ 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 $\{u. u \models_{cs} up\text{-}cs\ cs\} = \{u \oplus d \mid u \ d. u \models_{cs} cs \wedge d \geq 0\}$

<proof>

definition

$close\text{-}lu :: ('t::linordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add) DBM \Rightarrow 't DBM$

where

$close\text{-}lu\ M \equiv \lambda i\ j. \text{if } i > 0 \text{ then } min\ (dbm\text{-}add\ (M\ i\ 0)\ (M\ 0\ j))\ (M\ i\ j)$
 $else\ M\ i\ j$

definition

$up' :: ('t::linordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add) DBM \Rightarrow 't DBM$

where

$up'\ M \equiv \lambda i\ j. \text{if } i > 0 \wedge j = 0 \text{ then } \infty \text{ else } M\ i\ j$

lemma *up-alt-def*:

$up\ M = up'\ (close\text{-}lu\ M)$

<proof>

lemma *close-lu-equiv*:

fixes $M :: 't :: time DBM$

shows $dbm\text{-}to\text{-}cs\ n\ v\ M \equiv_{cs}\ dbm\text{-}to\text{-}cs\ n\ v\ (close\text{-}lu\ M)$

<proof>

lemma *close-lu-closed*:

lower-upper-closed $(dbm\text{-}to\text{-}cs\ n\ v\ (close\text{-}lu\ M))$ **if** $M\ 0\ 0 \geq 0$

<proof>

lemma *close-lu-closed'*: — Unused

lower-upper-closed $(dbm\text{-}to\text{-}cs\ n\ v\ (close\text{-}lu\ M) \cup dbm\text{-}to\text{-}cs\ n\ v\ M)$ **if** $M\ 0\ 0 \geq 0$

<proof>

lemma *up-cs-up'-equiv*:

fixes $M :: 't :: time DBM$

assumes $M\ 0\ 0 \geq 0$ *clock-numbering v*

shows $up\text{-}cs\ (dbm\text{-}to\text{-}cs\ n\ v\ M) \equiv_{cs}\ dbm\text{-}to\text{-}cs\ n\ v\ (up'\ M)$

<proof>

lemma *up-equiv-cong*: — Unused

fixes $cs\ cs' :: ('x, 'v :: time) cs$

assumes $cs \equiv_{cs}\ cs'$ *finite cs finite cs' lower-upper-closed cs lower-upper-closed*

cs'
shows $up\text{-}cs\ cs \equiv_{cs}\ up\text{-}cs\ cs'$
 ⟨*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
 $reset\text{-}canonical\ M\ k\ d =$
 ($\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* $M\ n$

$\langle proof \rangle$

lemma *dbm-neg-add*:

fixes $a :: ('t :: time) DBMEntry$

assumes $a < 0$

shows $a + a < 0$

$\langle proof \rangle$

instance *linordered-ab-group-add* \subseteq *linordered-cancel-ab-monoid-add* $\langle proof \rangle$

lemma *Le-cancel-1[simp]*:

fixes $d :: 'c :: linordered-ab-group-add$

shows $Le\ d + Le\ (-d) = Le\ 0$

$\langle proof \rangle$

lemma *Le-cancel-2[simp]*:

fixes $d :: 'c :: linordered-ab-group-add$

shows $Le\ (-d) + Le\ d = Le\ 0$

$\langle proof \rangle$

lemma *reset-canonical-canonical'*:

canonical (*reset-canonical* $M\ k\ (d :: 'c :: linordered-ab-group-add)$) n

if $M\ 0\ 0 = 0\ M\ k\ k = 0$ *canonical* $M\ n\ k > 0$ **for** $k\ n :: nat$

$\langle proof \rangle$

lemma *reset-canonical-canonical*:

canonical (*reset-canonical* $M\ k\ (d :: 'c :: linordered-ab-group-add)$) n

if $\forall\ i \leq n. M\ i\ i = 0$ *canonical* $M\ n\ k > 0$ **for** $k\ n :: nat$

$\langle proof \rangle$

lemma *canonicalD[simp]*:

assumes *canonical* $M\ n\ i \leq n\ j \leq n\ k \leq n$

shows $min\ (dbm-add\ (M\ i\ k)\ (M\ k\ j))\ (M\ i\ j) = M\ i\ j$

$\langle proof \rangle$

lemma *reset-reset-canonical*:

assumes *canonical* $M\ n\ k > 0\ k \leq n$ *clock-numbering* v

shows $[reset\ M\ n\ k\ d]_{v,n} = [reset-canonical\ M\ k\ d]_{v,n}$

$\langle proof \rangle$

lemma *reset-canonical-diag-preservation*:

fixes $k :: nat$

assumes $k > 0$

shows $\forall i \leq n. (\text{reset-canonical } M \ k \ d) \ i \ i = M \ i \ i$
 $\langle \text{proof} \rangle$

definition *reset''* **where**

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

lemma *reset''-diag-preservation*:

assumes *clock-numbering* v

shows $\forall i \leq n. (\text{reset'' } M \ n \ cs \ v \ d) \ i \ i = M \ i \ i$

$\langle \text{proof} \rangle$

lemma *reset-resets*:

assumes $\forall k \leq n. k > 0 \longrightarrow (\exists c. v \ c = k) \text{ clock-numbering}' \ v \ n \ v \ c \leq n$

shows $[\text{reset } M \ n \ (v \ c) \ d]_{v,n} = \{u(c := d) \mid u. u \in [M]_{v,n}\}$

$\langle \text{proof} \rangle$

lemma *reset-eq'*:

assumes *prems*: $\forall k \leq n. k > 0 \longrightarrow (\exists c. v \ c = k) \text{ clock-numbering}' \ v \ n \ v \ c \leq n$

and *eq*: $[M]_{v,n} = [M']_{v,n}$

shows $[\text{reset } M \ n \ (v \ c) \ d]_{v,n} = [\text{reset } M' \ n \ (v \ c) \ d]_{v,n}$

$\langle \text{proof} \rangle$

lemma *reset-eq*:

assumes *prems*: $\forall k \leq n. k > 0 \longrightarrow (\exists c. v \ c = k) \text{ clock-numbering}' \ v \ n$

and *k*: $k > 0 \ k \leq n$

and *eq*: $[M]_{v,n} = [M']_{v,n}$

shows $[\text{reset } M \ n \ k \ d]_{v,n} = [\text{reset } M' \ n \ k \ d]_{v,n}$

$\langle \text{proof} \rangle$

lemma *FW-reset-commute*:

assumes *prems*: $\forall k \leq n. k > 0 \longrightarrow (\exists c. v \ c = k) \text{ clock-numbering}' \ v \ n \ k > 0 \ k \leq n$

shows $[\text{FW } (\text{reset } M \ n \ k \ d) \ n]_{v,n} = [\text{reset } (\text{FW } M \ n) \ n \ k \ d]_{v,n}$

$\langle \text{proof} \rangle$

lemma *reset-canonical-diag-presv*:

fixes $k :: \text{nat}$

assumes $M \ i \ i = \text{Le } 0 \ k > 0$

shows $(\text{reset-canonical } M \ k \ d) \ i \ i = \text{Le } 0$

$\langle \text{proof} \rangle$

lemma *reset-cycle-free*:

fixes $M :: ('t :: \text{time}) \ \text{DBM}$

assumes *cycle-free* $M\ n$
and *prems*: $\forall k \leq n. k > 0 \longrightarrow (\exists c. v\ c = k)$ *clock-numbering'* $v\ n\ k > 0\ k \leq n$
shows *cycle-free* (*reset* $M\ n\ k\ d$) n
 \langle *proof* \rangle

lemma *reset'-reset''-equiv*:

assumes *canonical* $M\ n\ d \geq 0\ \forall i \leq n. M\ i\ i = 0$
clock-numbering' $v\ n\ \forall c \in \text{set } cs. v\ c \leq n$
and *surj*: $\forall k \leq n. k > 0 \longrightarrow (\exists c. v\ c = k)$
shows $[\text{reset}'\ M\ n\ cs\ v\ d]_{v,n} = [\text{reset}''\ M\ n\ cs\ v\ d]_{v,n}$
 \langle *proof* \rangle

Eliminating the clock numbering

definition *reset'''* **where**

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

lemma *reset''-reset'''*:

assumes $\forall c \in \text{set } cs. v\ c = c$
shows $\text{reset}''\ M\ n\ cs\ v\ d = \text{reset}'''\ M\ n\ cs\ d$
 \langle *proof* \rangle

type-synonym $'a\ DBM' = \text{nat} \times \text{nat} \Rightarrow 'a\ DBM\text{Entry}$

definition

reset-canonical-upd
 $(M :: ('a :: \{\text{linordered-cancel-ab-monoid-add, uminus}\})\ DBM')\ (n :: \text{nat})\ (k :: \text{nat})\ d =$
 $\text{fold } (\lambda\ i\ M. \text{if } i = k \text{ then } M \text{ else } M((k, i) := \text{Le } d + M(0, i), (i, k) :=$
 $\text{Le } (-d) + M(i, 0)))$
 $(\text{map } \text{nat } [1..n])$
 $(M((k, 0) := \text{Le } d, (0, k) := \text{Le } (-d)))$

lemma *one-upto-Suc*:

$[1..<\text{Suc } i + 1] = [1..<i+1] @ [\text{Suc } i]$
 \langle *proof* \rangle

lemma *one-upto-Suc'*:

$[1..\text{Suc } i] = [1..i] @ [\text{Suc } i]$
 \langle *proof* \rangle

lemma *one-upto-Suc''*:

$[1..1 + i] = [1..i] @ [\text{Suc } i]$

$\langle \text{proof} \rangle$

lemma *reset-canonical-upd-diag-id:*

fixes $k\ n :: \text{nat}$

assumes $k > 0$

shows $(\text{reset-canonical-upd } M\ n\ k\ d)\ (k, k) = M\ (k, k)$

$\langle \text{proof} \rangle$

lemma *reset-canonical-upd-out-of-bounds-id1:*

fixes $i\ j\ k\ n :: \text{nat}$

assumes $i \neq k\ i > n$

shows $(\text{reset-canonical-upd } M\ n\ k\ d)\ (i, j) = M\ (i, j)$

$\langle \text{proof} \rangle$

lemma *reset-canonical-upd-out-of-bounds-id2:*

fixes $i\ j\ k\ n :: \text{nat}$

assumes $j \neq k\ j > n$

shows $(\text{reset-canonical-upd } M\ n\ k\ d)\ (i, j) = M\ (i, j)$

$\langle \text{proof} \rangle$

lemma *reset-canonical-upd-out-of-bounds1:*

fixes $i\ j\ k\ n :: \text{nat}$

assumes $k \leq n\ i > n$

shows $(\text{reset-canonical-upd } M\ n\ k\ d)\ (i, j) = M\ (i, j)$

$\langle \text{proof} \rangle$

lemma *reset-canonical-upd-out-of-bounds2:*

fixes $i\ j\ k\ n :: \text{nat}$

assumes $k \leq n\ j > n$

shows $(\text{reset-canonical-upd } M\ n\ k\ d)\ (i, j) = M\ (i, j)$

$\langle \text{proof} \rangle$

lemma *reset-canonical-upd-id1:*

fixes $k\ n :: \text{nat}$

assumes $k > 0\ i > 0\ i \leq n\ i \neq k$

shows $(\text{reset-canonical-upd } M\ n\ k\ d)\ (i, k) = Le\ (-d) + M(i, 0)$

$\langle \text{proof} \rangle$

lemma *reset-canonical-upd-id2:*

fixes $k\ n :: \text{nat}$

assumes $k > 0\ i > 0\ i \leq n\ i \neq k$

shows $(\text{reset-canonical-upd } M\ n\ k\ d)\ (k, i) = Le\ d + M(0, i)$

$\langle \text{proof} \rangle$

lemma *reset-canonical-updid-0-1*:

fixes $n :: \text{nat}$

assumes $k > 0$

shows $(\text{reset-canonical-upd } M \ n \ k \ d) \ (0, k) = \text{Le } (-d)$

$\langle \text{proof} \rangle$

lemma *reset-canonical-updid-0-2*:

fixes $n :: \text{nat}$

assumes $k > 0$

shows $(\text{reset-canonical-upd } M \ n \ k \ d) \ (k, 0) = \text{Le } d$

$\langle \text{proof} \rangle$

lemma *reset-canonical-upd-id*:

fixes $n :: \text{nat}$

assumes $i \neq k \ j \neq k$

shows $(\text{reset-canonical-upd } M \ n \ k \ d) \ (i, j) = M \ (i, j)$

$\langle \text{proof} \rangle$

lemma *reset-canonical-upd-reset-canonical*:

fixes $i \ j \ k \ n :: \text{nat}$ **and** $M :: \text{nat} \times \text{nat} \Rightarrow ('a :: \{\text{linordered-cancel-ab-monoid-add, uminus}\})$
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 $(\text{reset-canonical-upd } M \ n \ k \ d) \ (i, j) = (\text{reset-canonical } M' \ k \ d) \ i \ j$

(is ?M(i,j) = -)

$\langle \text{proof} \rangle$

lemma *reset-canonical-upd-reset-canonical'*:

fixes $i \ j \ k \ n :: \text{nat}$

assumes $k > 0 \ i \leq n \ j \leq n$

shows $(\text{reset-canonical-upd } M \ n \ k \ d) \ (i, j) = (\text{reset-canonical } (\text{curry } M) \ k \ d) \ i \ j$ **(is ?M(i,j) = -)**

$\langle \text{proof} \rangle$

lemma *reset-canonical-upd-canonical*:

$\text{canonical } (\text{curry } (\text{reset-canonical-upd } M \ n \ k \ (d :: 'c :: \{\text{linordered-ab-group-add, uminus}\})))$
 n

if $\forall \ i \leq n. \ M \ (i, i) = 0$ $\text{canonical } (\text{curry } M) \ n \ k > 0$ **for** $k \ n :: \text{nat}$

$\langle \text{proof} \rangle$

definition *reset'-upd where*

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

lemma *reset'''-reset'-upd*:

fixes $n :: \text{nat}$ **and** $cs :: \text{nat list}$

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

lemma *reset'''-reset'-upd'*:

fixes $n :: \text{nat}$ **and** $cs :: \text{nat list}$ **and** $M :: ('a :: \{\text{linordered-cancel-ab-monoid-add, uminus}\})$
 DBM'

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

lemma *reset'-upd-out-of-bounds1*:

fixes $i \ j \ k \ n :: \text{nat}$
assumes $\forall c \in \text{set } cs. c \leq n \ i > n$
shows $(\text{reset}'\text{-upd } M \ n \ cs \ d) \ (i, j) = M \ (i, j)$
 $\langle \text{proof} \rangle$

lemma *reset'-upd-out-of-bounds2*:

fixes $i \ j \ k \ n :: \text{nat}$
assumes $\forall c \in \text{set } cs. c \leq n \ j > n$
shows $(\text{reset}'\text{-upd } M \ n \ cs \ d) \ (i, j) = M \ (i, j)$
 $\langle \text{proof} \rangle$

lemma *reset-canonical-int-preservation*:

fixes $n :: \text{nat}$
assumes $\text{dbm-int } M \ n \ d \in \mathbb{Z}$
shows $\text{dbm-int } (\text{reset-canonical } M \ k \ d) \ n$
 $\langle \text{proof} \rangle$

lemma *reset-canonical-upd-int-preservation*:

assumes $\text{dbm-int } (\text{curry } M) \ n \ d \in \mathbb{Z} \ k > 0$
shows $\text{dbm-int } (\text{curry } (\text{reset-canonical-upd } M \ n \ k \ d)) \ n$
 $\langle \text{proof} \rangle$

lemma *reset'-upd-int-preservation*:

assumes $\text{dbm-int } (\text{curry } M) \ n \ d \in \mathbb{Z} \ \forall c \in \text{set } cs. c \neq 0$
shows $\text{dbm-int } (\text{curry } (\text{reset}'\text{-upd } M \ n \ cs \ d)) \ n$
 $\langle \text{proof} \rangle$

lemma *reset-canonical-upd-diag-preservation*:

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

<proof>

lemma *reset'-upd-diag-preservation:*

assumes $\forall c \in \text{set } cs. c > 0 \ i \leq n$

shows $(\text{reset}'\text{-upd } M \ n \ cs \ d) \ (i, i) = M \ (i, i)$

<proof>

lemma *upto-from-1-upt:*

fixes $n :: \text{nat}$

shows $\text{map } \text{nat } [1..\text{int } n] = [1..\<n+1]$

<proof>

lemma *reset-canonical-upd-alt-def:*

reset-canonical-upd ($M :: ('a :: \{\text{linordered-cancel-ab-monoid-add, uminus}\})$
 DBM') ($n :: \text{nat}$) ($k :: \text{nat}$) $d =$

fold

$(\lambda \ i \ M.$

if $i = k$ *then*

M

else do {

let $m0i = \text{op-mtx-get } M(0, i);$

let $mi0 = \text{op-mtx-get } M(i, 0);$

$M((k, i) := \text{Le } d + m0i, (i, k) := \text{Le } (-d) + mi0)$

}

)

$[1..\<n+1]$

$(M((k, 0) := \text{Le } d, (0, k) := \text{Le } (-d)))$

<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* $(up\ M)\ n$

$\langle proof \rangle$

definition *up-canonical* $:: 't\ DBM \Rightarrow 't\ DBM$ **where**

up-canonical $M = (\lambda\ i\ j. \text{if } i > 0 \wedge j = 0 \text{ then } \infty \text{ else } M\ i\ j)$

lemma *up-canonical-eq-up*:

assumes *canonical* $M\ n\ i \leq n\ j \leq n$

shows *up-canonical* $M\ i\ j = up\ M\ i\ j$

$\langle proof \rangle$

lemma *DBM-up-to-equiv*:

assumes $\forall\ i \leq n. \forall\ j \leq n. M\ i\ j = M'\ i\ j$

shows $[M]_{v,n} = [M']_{v,n}$

$\langle proof \rangle$

lemma *up-canonical-equiv-up*:

assumes *canonical* $M\ n$

shows $[up\ canonical\ M]_{v,n} = [up\ M]_{v,n}$

$\langle proof \rangle$

lemma *up-canonical-diag-preservation*:

assumes $\forall\ i \leq n. M\ i\ i = 0$

shows $\forall\ i \leq n. (up\ canonical\ M)\ i\ i = 0$

$\langle proof \rangle$

no-notation *Ref.update* $(\langle - := - \rangle\ 62)$

definition *up-canonical-upd* $:: 't\ DBM' \Rightarrow nat \Rightarrow 't\ DBM'$ **where**

up-canonical-upd $M\ n = fold\ (\lambda\ i\ M. M((i,0) := \infty))\ [1..<n+1]\ M$

lemma *up-canonical-upd-rec*:

up-canonical-upd $M\ (Suc\ n) = (up\ canonical\ upd\ M\ n)\ ((Suc\ n, 0) := \infty)$

$\langle proof \rangle$

lemma *up-canonical-out-of-bounds1*:

fixes $i :: nat$

assumes $i > n$

shows *up-canonical-upd* $M\ n\ (i, j) = M(i, j)$

$\langle proof \rangle$

lemma *up-canonical-out-of-bounds2*:
fixes $j :: \text{nat}$
assumes $j > 0$
shows $\text{up-canonical-upd } M \ n \ (i, j) = M(i,j)$
 $\langle \text{proof} \rangle$

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 $(\text{up-canonical-upd } M \ n) \ (i, j) = (\text{up-canonical } M') \ i \ j$
 $\langle \text{proof} \rangle$

lemma *up-canonical-int-preservation*:
assumes $\text{dbm-int } M \ n$
shows $\text{dbm-int } (\text{up-canonical } M) \ n$
 $\langle \text{proof} \rangle$

lemma *up-canonical-upd-int-preservation*:
assumes $\text{dbm-int } (\text{curry } M) \ n$
shows $\text{dbm-int } (\text{curry } (\text{up-canonical-upd } M \ n)) \ n$
 $\langle \text{proof} \rangle$

lemma *up-canonical-diag-preservation'*:
 $(\text{up-canonical } M) \ i \ i = M \ i \ i$
 $\langle \text{proof} \rangle$

lemma *up-canonical-upd-diag-preservation*:
 $(\text{up-canonical-upd } M \ n) \ (i, i) = M \ (i, i)$
 $\langle \text{proof} \rangle$

5.4 Intersection

definition

$\text{unbounded-dbm } n = (\lambda \ (i, j). \ (\text{if } i = j \vee i > n \vee j > n \text{ then } Le \ 0 \ \text{else } \infty))$

definition $\text{And-upd} :: \text{nat} \Rightarrow ('t::\{\text{linorder}, \text{zero}\}) \ \text{DBM}' \Rightarrow 't \ \text{DBM}' \Rightarrow 't \ \text{DBM}'$ **where**

$\text{And-upd } n \ A \ B =$
 $\text{fold } (\lambda i \ M.$
 $\text{fold } (\lambda j \ M. \ M((i,j) := \min \ (A(i,j)) \ (B(i,j)))) \ [0..<n+1] \ M)$
 $[0..<n+1]$
 $(\text{unbounded-dbm } n)$

lemma *fold-loop-inv-rule*:

assumes $I\ 0\ x$
assumes $\bigwedge i\ x. I\ i\ x \implies i \leq n \implies I\ (\text{Suc } i)\ (f\ i\ x)$
assumes $\bigwedge x. I\ n\ x \implies Q\ x$
shows $Q\ (\text{fold } f\ [0..<n]\ x)$
 $\langle \text{proof} \rangle$

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

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 $\text{And-upd } n\ A\ B\ (i, j) = \text{And } A'\ B'\ i\ j$
 $\langle \text{proof} \rangle$

5.5 Inclusion

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

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

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

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

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

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

\Rightarrow *bool* **where**

$dbm\text{-subset } n \ M \ M' \equiv check\text{-diag } n \ M \vee pointwise\text{-cmp } (\leq) \ n \ (curry \ M)$
 $(curry \ M')$

lemma *dbm-subset-refl*:

$dbm\text{-subset } n \ M \ M$
 $\langle proof \rangle$

lemma *dbm-subset-trans*:

assumes $dbm\text{-subset } n \ M1 \ M2 \ dbm\text{-subset } n \ M2 \ M3$
shows $dbm\text{-subset } n \ M1 \ M3$
 $\langle proof \rangle$

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 $[M]_{v,n} \neq \{\}$
 $\langle proof \rangle$

The type constraint in this lemma is due to $[[canonical \ ?M \ ?n; [?M]_{?v,?n} \subseteq [?M']_{?v,?n}; [?M]_{?v,?n} \neq \{\}; ?i \leq ?n; ?j \leq ?n; ?i \neq ?j; \forall c. \ 0 < ?v \ c \wedge (\forall x \ y. \ ?v \ x \leq ?n \wedge ?v \ y \leq ?n \wedge ?v \ x = ?v \ y \longrightarrow x = y); \forall k \leq ?n. \ 0 < k \longrightarrow (\exists c. \ ?v \ c = k)]] \implies ?M \ ?i \ ?j \leq ?M' \ ?i \ ?j$. 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 $canonical \ (curry \ M) \ n \ \vee \ check\text{-diag } n \ M \ \forall i \leq n. \ M \ (i, i) \leq 0 \ \forall i \leq n. \ M' \ (i, i) \leq 0$
and $cn: \ clock\text{-numbering}' \ v \ n \ \mathbf{and} \ surj: \ \forall k \leq n. \ 0 < k \longrightarrow (\exists c. \ v \ c = k)$
shows $[curry \ M]_{v,n} \subseteq [curry \ M']_{v,n} \longleftrightarrow dbm\text{-subset } n \ M \ M'$
 $\langle proof \rangle$

lemma *pointwise-cmp-alt-def*:

$pointwise\text{-cmp } P \ n \ M \ M' =$
 $list\text{-all } (\lambda i. \ list\text{-all } (\lambda j. \ P \ (M \ i \ j) \ (M' \ i \ j)) \ [0..<Suc \ n]) \ [0..<Suc \ n]$
 $\langle proof \rangle$

lemma *dbm-subset-alt-def[code]*:

$dbm\text{-subset } n \ M \ M' =$
 $(list\text{-ex } (\lambda i. \ op\text{-mtx-get } M \ (i, i) < Le \ 0) \ [0..<Suc \ n] \ \vee$
 $list\text{-all } (\lambda i. \ list\text{-all } (\lambda j. \ (op\text{-mtx-get } M \ (i, j) \leq op\text{-mtx-get } M' \ (i, j)))$
 $[0..<Suc \ n]) \ [0..<Suc \ n])$
 $\langle proof \rangle$

definition *pointwise-cmp-alt-def* **where**

pointwise-cmp-alt-def $P\ n\ M\ M' = \text{fold } (\lambda\ i\ b.\ \text{fold } (\lambda\ j\ b.\ P\ (M\ i\ j)\ (M'\ i\ j) \wedge b))\ [1..\text{Suc } n]\ b)\ [1..\text{Suc } n]\ \text{True}$

lemma *list-all-foldli*:

list-all $P\ xs = \text{foldli } xs\ (\lambda x.\ x = \text{True})\ (\lambda x\ -. P\ x)\ \text{True}$
 $\langle \text{proof} \rangle$

lemma *list-ex-foldli*:

list-ex $P\ xs = \text{foldli } xs\ \text{Not } (\lambda x\ y.\ P\ x \vee y)\ \text{False}$
 $\langle \text{proof} \rangle$

5.6 Extrapolations

context

fixes

upd-entry $::\ \text{nat} \Rightarrow \text{nat} \Rightarrow 't \Rightarrow 't \Rightarrow ('t::\{\text{linordered-ab-group-add}\})$
 $\text{DBMEntry} \Rightarrow 't\ \text{DBMEntry}$

and *upd-entry-0* $::\ \text{nat} \Rightarrow 't \Rightarrow 't\ \text{DBMEntry} \Rightarrow 't\ \text{DBMEntry}$

begin

definition *extra* $::$

$'t\ \text{DBM} \Rightarrow (\text{nat} \Rightarrow 't) \Rightarrow (\text{nat} \Rightarrow 't) \Rightarrow \text{nat} \Rightarrow 't\ \text{DBM}$

where

extra $M\ l\ u\ n \equiv \lambda i\ j.$

let $ub = \text{if } i > 0 \text{ then } l\ i \text{ else } 0 \text{ in}$

let $lb = \text{if } j > 0 \text{ then } u\ j \text{ else } 0 \text{ in}$

if $i \leq n \wedge j \leq n \text{ then}$

if $i \neq j \text{ then}$

if $i > 0 \text{ then } \text{upd-entry } i\ j\ lb\ ub\ (M\ i\ j) \text{ else } \text{upd-entry-0 } j\ lb\ (M\ i\ j)$

else $\text{norm-diag } (M\ i\ j)$

else $M\ i\ j$

definition *upd-line-0* $::$

$'t\ \text{DBM}' \Rightarrow 't\ \text{list} \Rightarrow \text{nat} \Rightarrow 't\ \text{DBM}'$

where

upd-line-0 $M\ k\ n =$

fold

$(\lambda j\ M.$

$M((0, j) := \text{upd-entry-0 } j\ (\text{op-list-get } k\ j)\ (M(0, j))))$

$[1..\text{Suc } n]$

$(M((0, 0) := \text{norm-diag } (M\ (0, 0))))$

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 $extra\text{-}upd :: 't\ DBM' \Rightarrow 't\ list \Rightarrow 't\ list \Rightarrow nat \Rightarrow 't\ DBM'$

where

$extra\text{-}upd\ M\ l\ u\ n \equiv$
 $fold\ (\lambda i\ M.\ upd\text{-}line\ M\ u\ (op\text{-}list\text{-}get\ l\ i)\ i\ n)\ [1..<Suc\ n]\ (upd\text{-}line\text{-}0\ M\ u\ n)$

lemma $upd\text{-}line\text{-}0\text{-}out\text{-}ouf\text{-}bounds1:$

assumes $i > 0$

shows $upd\text{-}line\text{-}0\ M\ k\ n\ (i, j) = M\ (i, j)$

<proof>

lemma $upd\text{-}line\text{-}0\text{-}out\text{-}ouf\text{-}bounds2:$

assumes $j > n$

shows $upd\text{-}line\text{-}0\ M\ k\ n\ (i, j) = M\ (i, j)$

<proof>

lemma $upd\text{-}out\text{-}of\text{-}bounds\text{-}aux1:$

assumes $i > n$

shows $fold\ (\lambda i\ M.\ upd\text{-}line\ M\ k\ (op\text{-}list\text{-}get\ l\ i)\ i\ m)\ [1..<Suc\ n]\ M\ (i, j)$
 $= M\ (i, j)$

<proof>

lemma $upd\text{-}out\text{-}of\text{-}bounds\text{-}aux2:$

assumes $j > m$

shows $fold\ (\lambda i\ M.\ upd\text{-}line\ M\ k\ (op\text{-}list\text{-}get\ l\ i)\ i\ m)\ [1..<Suc\ n]\ M\ (i, j)$
 $= M\ (i, j)$

<proof>

lemma $upd\text{-}out\text{-}of\text{-}bounds1:$

assumes $i > n$

shows $extra\text{-}upd\ M\ l\ u\ n\ (i, j) = M\ (i, j)$

<proof>

lemma $upd\text{-}out\text{-}of\text{-}bounds2:$

assumes $j > n$

shows $extra\text{-}upd\ M\ l\ u\ n\ (i, j) = M\ (i, j)$

<proof>

definition $norm\text{-}entry$ **where**

$norm\text{-}entry\ x\ l\ u\ i\ j = ($

let $ub = \text{if } i > 0 \text{ then } (l ! i) \text{ else } 0 \text{ in}$
 let $lb = \text{if } j > 0 \text{ then } (u ! j) \text{ else } 0 \text{ in}$
 if $i \neq j$ then if $i = 0$ then $\text{upd-entry-0 } j \text{ lb } x$ else $\text{upd-entry } i \text{ j lb ub } x$ else
 $\text{norm-diag } x$

lemma *upd-extra-aux*:

assumes $i \leq n \ j \leq m$

shows

$\text{fold } (\lambda i \ M. \ \text{upd-line } M \ u \ (\text{op-list-get } l \ i) \ i \ m) \ [1..<\text{Suc } n] \ (\text{upd-line-0 } M$
 $u \ m) \ (i, \ j)$
 $= \text{norm-entry } (M \ (i, \ j)) \ l \ u \ i \ j$
 <proof>

lemma *upd-extra-aux'*:

assumes $i \leq n \ j \leq n$

shows $\text{extra-upd } M \ l \ u \ n \ (i, \ j) = \text{extra } (\text{curry } M) \ (\lambda i. \ l ! i) \ (\lambda i. \ u ! i) \ n \ i \ j$
 <proof>

lemma *extra-upd-extra''*:

$\text{extra-upd } M \ l \ u \ n \ (i, \ j) = \text{extra } (\text{curry } M) \ (\lambda i. \ l ! i) \ (\lambda i. \ u ! i) \ n \ i \ j$
 <proof>

lemma *extra-upd-extra'*:

$\text{curry } (\text{extra-upd } M \ l \ u \ n) = \text{extra } (\text{curry } M) \ (\lambda i. \ l ! i) \ (\lambda i. \ u ! i) \ n$
 <proof>

lemma *extra-upd-extra*:

$\text{extra-upd} = (\lambda M \ l \ u \ n \ (i, \ j). \ \text{extra } (\text{curry } M) \ (\lambda i. \ l ! i) \ (\lambda i. \ u ! i) \ n \ i \ j)$
 <proof>

end

lemma *norm-is-extra*:

$\text{norm } M \ k \ n =$

extra

$(\lambda - \ lb \ ub \ e. \ \text{norm-lower } (\text{norm-upper } e \ ub) \ (-lb))$

$(\lambda - \ lb \ e. \ \text{norm-lower } (\text{norm-upper } e \ 0) \ (-lb)) \ M \ k \ k \ n$

<proof>

lemma *extra-lu-is-extra*:

$\text{extra-lu } M \ l \ u \ n =$

extra

$(\lambda - \ lb \ ub \ e. \ \text{norm-lower } (\text{norm-upper } e \ ub) \ (-lb))$

(λ - lb e. norm-lower (norm-upper e 0) (-lb)) M l u n
 <proof>

lemma extra-lup-is-extra:

extra-lup M l u n =
 extra
 (λ i j lb ub e. if Lt ub < e then ∞
 else if M 0 i < Lt (- ub) then ∞
 else if M 0 j < (if j > 0 then Lt (- lb) else Lt 0) then ∞
 else e)
 (λ j lb e. if Le 0 < M 0 j then ∞
 else if M 0 j < (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
 (λ - lb ub e. norm-lower (norm-upper e ub) (-lb))
 (λ - lb e. norm-lower (norm-upper e 0) (-lb)) M k k

definition

extra-lu-upd =
 extra-upd
 (λ - lb ub e. norm-lower (norm-upper e ub) (-lb))
 (λ - lb e. norm-lower (norm-upper e 0) (-lb))

definition

extra-lup-upd M =
 extra-upd
 (λ i j lb ub e. if Lt ub < e then ∞
 else if M (0, i) < Lt (- ub) then ∞
 else if M (0, j) < (if j > 0 then Lt (- lb) else Lt 0) then ∞
 else e)
 (λ j lb e. if Le 0 < M (0, j) then ∞
 else if M (0, j) < (if j > 0 then Lt (- lb) else Lt 0) then Lt (- lb)
 else M (0, j)) M

lemma extra-upd-cong:

assumes $\bigwedge i j x y e. i \leq n \implies j \leq n \implies \text{upd-entry } i j x y e = \text{upd-entry}'$
 $i j x y e$

$\bigwedge i x e. i \leq n \implies \text{upd-entry-0 } i x e = \text{upd-entry-0}' i x e$

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 $xs = IArray\ (map\ (\lambda i. M\ (0, i))\ [0..<Suc\ n])$ in
 extra-upd
 $(\lambda i\ j\ lb\ ub\ e. if\ Lt\ ub\ <\ e\ then\ \infty$
 else if $(xs\ !!\ i) <\ Lt\ (-\ ub)$ then ∞
 else if $(xs\ !!\ j) <\ (if\ j > 0\ then\ Lt\ (-\ lb)\ else\ Lt\ 0)$ then ∞
 else e)
 $(\lambda j\ lb\ e. if\ Le\ 0 <\ (xs\ !!\ j)$ 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 $xs = map\ (\lambda i. M\ (0, i))\ [0..<Suc\ n]$ in
 extra-upd
 $(\lambda i\ j\ lb\ ub\ e. if\ Lt\ ub <\ e\ then\ \infty$
 else if $(xs\ !\ i) <\ Lt\ (-\ ub)$ then ∞
 else if $(xs\ !\ j) <\ (if\ j > 0\ then\ Lt\ (-\ lb)\ else\ Lt\ 0)$ then ∞
 else e)
 $(\lambda j\ lb\ e. if\ Le\ 0 <\ (xs\ !\ j)$ 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:* $norm\text{-}upd = (\lambda M\ k\ n\ (i, j). norm\ (curry\ M)\ (\lambda i. k\ !\ i)\ n\ i\ j)$

and *extra-lu-upd-extra-lu:*

$extra\text{-}lu\text{-}upd = (\lambda M\ l\ u\ n\ (i, j). extra\text{-}lu\ (curry\ M)\ (\lambda i. l\ !\ i)\ (\lambda i. u\ !\ i)\ n\ i\ j)$

and *extra-lup-upd-extra-lup:*

$extra\text{-}lup\text{-}upd = (\lambda M\ l\ u\ n\ (i, j). extra\text{-}lup\ (curry\ M)\ (\lambda i. l\ !\ i)\ (\lambda i. u\ !\ i)\ n\ i\ j)$

⟨proof⟩

lemma *norm-upd-norm':*

$curry\ (norm\text{-}upd\ M\ k\ n) = norm\ (curry\ M)\ (\lambda i. k\ !\ i)\ n$
⟨proof⟩

lemma *norm-int-preservation:*

assumes $dbm\text{-}int\ M\ n\ \forall\ c \leq n. k\ c \in \mathbb{Z}$

shows $dbm\text{-}int\ (norm\ M\ k\ n)\ n$

<proof>

lemma

assumes $dbm-int\ M\ n\ \forall\ c \leq n. l\ c \in \mathbb{Z}\ \forall\ c \leq n. u\ c \in \mathbb{Z}$
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 $dbm-int\ (curry\ M)\ n\ \forall\ c \in set\ k. c \in \mathbb{Z}\ length\ k = Suc\ n$
shows $dbm-int\ (curry\ (norm-upd\ M\ k\ n))\ n$
<proof>

lemma

fixes $M :: ('t :: \{linordered-ab-group-add, ring-1\})\ DBM'$
assumes $dbm-int\ (curry\ M)\ n$
 $\forall\ c \in set\ l. c \in \mathbb{Z}\ length\ l = Suc\ n\ \forall\ c \in set\ u. c \in \mathbb{Z}\ length\ u = Suc\ n$
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 $list-all\ (\lambda x. \langle A * true \rangle\ f\ x\ \langle \lambda r. \uparrow(Q\ x\ r) * A \rangle_t)\ xs$
shows $\langle A * true \rangle\ Heap-Monad.fold-map\ f\ xs\ \langle \lambda rs. \uparrow(list-all2\ (\lambda x\ r. Q\ x\ r)\ xs\ rs) * A \rangle_t$

$\langle proof \rangle$

lemma *fold-map-ht'*:

assumes $list\text{-}all (\lambda x. \langle true \rangle f x \langle \lambda r. \uparrow(Q x r) \rangle_t) xs$

shows $\langle true \rangle Heap\text{-}Monad.fold\text{-}map f xs \langle \lambda rs. \uparrow(list\text{-}all2 (\lambda x r. Q x r) xs rs) \rangle_t$

$\langle proof \rangle$

lemma *fold-map-ht1*:

assumes $\bigwedge x xi. \langle A * R x xi * true \rangle f xi \langle \lambda r. A * \uparrow(Q x r) \rangle_t$

shows

$\langle A * list\text{-}assn R xs xsi * true \rangle$

$Heap\text{-}Monad.fold\text{-}map f xsi$

$\langle \lambda rs. A * \uparrow(list\text{-}all2 (\lambda x r. Q x r) xs rs) \rangle_t$

$\langle proof \rangle$

lemma *fold-map-ht2*:

assumes $\bigwedge x xi. \langle A * R x xi * true \rangle f xi \langle \lambda r. A * R x xi * \uparrow(Q x r) \rangle_t$

shows

$\langle A * list\text{-}assn R xs xsi * true \rangle$

$Heap\text{-}Monad.fold\text{-}map f xsi$

$\langle \lambda rs. A * list\text{-}assn R xs xsi * \uparrow(list\text{-}all2 (\lambda x r. Q x r) xs rs) \rangle_t$

$\langle proof \rangle$

lemma *fold-map-ht3*:

assumes $\bigwedge x xi. \langle A * R x xi * true \rangle f xi \langle \lambda r. A * Q x r \rangle_t$

shows $\langle A * list\text{-}assn R xs xsi * true \rangle Heap\text{-}Monad.fold\text{-}map f xsi \langle \lambda rs. A * list\text{-}assn Q xs rs \rangle_t$

$\langle proof \rangle$

imp-for' and *imp-for* **lemma** *imp-for-rule2*:

assumes

$emp \implies_A I i a$

$\bigwedge i a. \langle A * I i a * true \rangle ci a \langle \lambda r. A * I i a * \uparrow(r \longleftrightarrow c a) \rangle_t$

$\bigwedge i a. i < j \implies c a \implies \langle A * I i a * true \rangle f i a \langle \lambda r. A * I (i + 1) r \rangle_t$

$\bigwedge a. I j a \implies_A Q a \bigwedge i a. i < j \implies \neg c a \implies I i a \implies_A Q a$
 $i \leq j$

shows $\langle A * true \rangle imp\text{-}for i j ci f a \langle \lambda r. A * Q r \rangle_t$

$\langle proof \rangle$

lemma *imp-for-rule*:

assumes

$emp \Longrightarrow_A I i a$
 $\wedge i a. \langle I i a * true \rangle ci a \langle \lambda r. I i a * \uparrow(r \longleftrightarrow c a) \rangle_t$
 $\wedge i a. i < j \Longrightarrow c a \Longrightarrow \langle I i a * true \rangle f i a \langle \lambda r. I (i + 1) r \rangle_t$
 $\wedge a. I j a \Longrightarrow_A Q a \wedge i a. i < j \Longrightarrow \neg c a \Longrightarrow I i a \Longrightarrow_A Q a$
 $i \leq j$
shows $\langle true \rangle imp\text{-for } i j ci f a \langle \lambda r. Q r \rangle_t$
 $\langle proof \rangle$

lemma *imp-for'-rule2*:

assumes
 $emp \Longrightarrow_A I i a$
 $\wedge i a. i < j \Longrightarrow \langle A * I i a * true \rangle f i a \langle \lambda r. A * I (i + 1) r \rangle_t$
 $\wedge a. I j a \Longrightarrow_A Q a$
 $i \leq j$
shows $\langle A * true \rangle imp\text{-for}' i j f a \langle \lambda r. A * Q r \rangle_t$
 $\langle proof \rangle$

lemma *imp-for'-rule*:

assumes
 $emp \Longrightarrow_A I i a$
 $\wedge i a. i < j \Longrightarrow \langle I i a * true \rangle f i a \langle \lambda r. I (i + 1) r \rangle_t$
 $\wedge a. I j a \Longrightarrow_A Q a$
 $i \leq j$
shows $\langle true \rangle imp\text{-for}' i j f a \langle \lambda r. Q r \rangle_t$
 $\langle proof \rangle$

lemma *nth-rule*:

assumes *is-pure S*
and $b < length a$
shows
 $\langle nat\text{-assn } b bi * array\text{-assn } S a ai \rangle Array.nth ai bi$
 $\langle \lambda r. \exists_{A x}. nat\text{-assn } b bi * array\text{-assn } S a ai * S x r * true * \uparrow(x = a$
 $! b) \rangle$
 $\langle proof \rangle$

lemma *imp-for-list-all*:

assumes
 $is\text{-pure } R n \leq length xs$
 $\wedge x xi. \langle A * R x xi * true \rangle Pi xi \langle \lambda r. A * \uparrow(r \longleftrightarrow P x) \rangle_t$
shows
 $\langle A * array\text{-assn } R xs a * true \rangle$
 $imp\text{-for } 0 n Heap\text{-Monad.return}$
 $(\lambda i -. do \{$
 $x \leftarrow Array.nth a i; Pi x$

$\}$
 $True$
 $\langle \lambda r. A * \text{array-assn } R \text{ } xs \text{ } a * \uparrow(r \longleftrightarrow \text{list-all } P \text{ } (\text{take } n \text{ } xs)) \rangle_t$
 $\langle \text{proof} \rangle$

lemma *imp-for-list-ex:*

assumes

$is\text{-pure } R \ n \leq \text{length } xs$

$\bigwedge x \ xi. \langle A * R \ x \ xi * true \rangle \ Pi \ xi \ \langle \lambda r. A * \uparrow(r \longleftrightarrow P \ x) \rangle_t$

shows

$\langle A * \text{array-assn } R \text{ } xs \text{ } a * true \rangle$

$imp\text{-for } 0 \ n \ (\lambda x. \text{Heap-Monad.return } (\neg x))$

$(\lambda i \ -. \ do \ \{$

$\ x \leftarrow \text{Array.nth } a \ i; \ Pi \ x$

$\})$

$False$

$\langle \lambda r. A * \text{array-assn } R \text{ } xs \text{ } a * \uparrow(r \longleftrightarrow \text{list-ex } P \text{ } (\text{take } n \text{ } xs)) \rangle_t$

$\langle \text{proof} \rangle$

lemma *imp-for-list-all2:*

assumes

$is\text{-pure } R \ is\text{-pure } S \ n \leq \text{length } xs \ n \leq \text{length } ys$

$\bigwedge x \ xi \ y \ yi. \langle A * R \ x \ xi * S \ y \ yi * true \rangle \ Pi \ xi \ yi \ \langle \lambda r. A * \uparrow(r \longleftrightarrow P \ x \ y) \rangle_t$

shows

$\langle A * \text{array-assn } R \text{ } xs \text{ } a * \text{array-assn } S \text{ } ys \text{ } b * true \rangle$

$imp\text{-for } 0 \ n \ \text{Heap-Monad.return}$

$(\lambda i \ -. \ do \ \{$

$\ x \leftarrow \text{Array.nth } a \ i; \ y \leftarrow \text{Array.nth } b \ i; \ Pi \ x \ y$

$\})$

$True$

$\langle \lambda r. A * \text{array-assn } R \text{ } xs \text{ } a * \text{array-assn } S \text{ } ys \text{ } b * \uparrow(r \longleftrightarrow \text{list-all2 } P \text{ } (\text{take } n \text{ } xs) \text{ } (\text{take } n \text{ } ys)) \rangle_t$

$\langle \text{proof} \rangle$

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

assumes

$is\text{-pure } R \ is\text{-pure } S \ n \leq \text{length } xs \ n \leq \text{length } ys$

$\bigwedge x \ xi \ y \ yi. \langle R \ x \ xi * S \ y \ yi \rangle \ Pi \ xi \ yi \ \langle \lambda r. \uparrow(r \longleftrightarrow P \ x \ y) \rangle_t$

shows

$\langle \text{array-assn } R \text{ } xs \text{ } a * \text{array-assn } S \text{ } ys \text{ } b \rangle$

$imp\text{-for } 0 \ n \ \text{Heap-Monad.return}$

$(\lambda i \ -. \ do \ \{$

$\ x \leftarrow \text{Array.nth } a \ i; \ y \leftarrow \text{Array.nth } b \ i; \ Pi \ x \ y$

```

    })
    True
    <λr. array-assn R xs a * array-assn S ys b * ↑(r ↔ list-all2 P (take n
xs) (take n ys))>t
    <proof>

```

end

theory *DBM-Operations-Impl-Refine*

imports

DBM-Operations-Impl

HOL-Library.IArray

DBM-Imperative-Loops

begin

lemma *rev-map-fold-append-aux*:

```

fold (λ x xs. f x # xs) xs zs @ ys = fold (λ x xs. f x # xs) xs (zs@ys)
<proof>

```

lemma *rev-map-fold*:

```

rev (map f xs) = fold (λ x xs. f x # xs) xs []
<proof>

```

lemma *map-rev-fold*:

```

map f xs = rev (fold (λ x xs. f x # xs) xs [])
<proof>

```

lemma *pointwise-cmp-iff*:

```

pointwise-cmp P n M M' ↔ list-all2 P (take ((n + 1) * (n + 1)) xs)
(take ((n + 1) * (n + 1)) ys)
if ∀ i ≤ n. ∀ j ≤ n. xs ! (i + i * n + j) = M i j
    ∀ i ≤ n. ∀ j ≤ n. ys ! (i + i * n + j) = M' i j
    (n + 1) * (n + 1) ≤ length xs (n + 1) * (n + 1) ≤ length ys
<proof>

```

fun *intersperse* :: 'a ⇒ 'a list ⇒ '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-pure (λa c. ↑ (c = a)) = Id
<proof>

```

lemma *pure-eq-conv*:

```

(λa c. ↑ (c = a)) = id-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 (λi. list-all (λj. (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 (λi. op-mtx-get M (i, i) < 0) [0..*Suc* n] ∨*
list-all (λi. list-all (λj. (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 (λi. M (0, i)) [0..*Suc* m])*

definition

*mtx-line m (M :: - DBM') = map (λi. 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* \equiv *nbn-assn* (*Suc n*)

lemmas *Relation.IdI* [**where** $a = \infty$, *sepref-import-param*]

lemma [*sepref-import-param*]: $((+), (+)) \in Id \rightarrow Id \rightarrow Id$ \langle proof \rangle

lemma [*sepref-import-param*]: $(uminus, uminus) \in (Id :: (-*) set) \rightarrow Id$ \langle proof \rangle

lemma [*sepref-import-param*]: $(Lt, Lt) \in Id \rightarrow Id$ \langle proof \rangle

lemma [*sepref-import-param*]: $(Le, Le) \in Id \rightarrow Id$ \langle proof \rangle

lemma [*sepref-import-param*]: $(\infty, \infty) \in Id$ \langle proof \rangle

lemma [*sepref-import-param*]: $(min :: - DBMEntry \Rightarrow -, min) \in Id \rightarrow Id \rightarrow Id$ \langle proof \rangle

lemma [*sepref-import-param*]: $(Suc, Suc) \in Id \rightarrow Id$ \langle proof \rangle

lemma [*sepref-import-param*]: $(norm-lower, norm-lower) \in Id \rightarrow Id \rightarrow Id$ \langle proof \rangle

lemma [*sepref-import-param*]: $(norm-upper, norm-upper) \in Id \rightarrow Id \rightarrow Id$ \langle proof \rangle

lemma [*sepref-import-param*]: $(norm-diag, norm-diag) \in Id \rightarrow Id$ \langle proof \rangle

end

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

zero-clock = 0

sepref-register *zero-clock*

lemma [*sepref-import-param*]: $(zero-clock, zero-clock) \in Id$ \langle proof \rangle

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

context

fixes $n :: nat$

begin

interpretation *DBM-Impl n* \langle proof \rangle

sepref-definition *reset-canonical-upd-impl'* **is**

uncurry2 (*uncurry* ($\lambda x. RETURN$ ooo *reset-canonical-upd x*)) ::
 $[\lambda((- , i), j), -]. i \leq n \wedge j \leq n]_a mtx-assn^d *_a nat-assn^k *_a nat-assn^k *_a$
id-assn^k \rightarrow *mtx-assn*
 \langle proof \rangle

sepref-definition *reset-canonical-upd-impl* **is**

uncurry2 (*uncurry* ($\lambda x. RETURN$ ooo *reset-canonical-upd x*)) ::
 $[\lambda((- , i), j), -]. i \leq n \wedge j \leq n]_a mtx-assn^d *_a nat-assn^k *_a nat-assn^k *_a$

$id\text{-}assn^k \rightarrow mt\text{x-}assn$
 $\langle proof \rangle$

sepref-definition *up-canonical-upd-impl* is

$uncurry (RETURN \circ\circ up\text{-}canonical\text{-}upd) :: [\lambda(-,i). i \leq n]_a mt\text{x-}assn^d *_a$
 $nat\text{-}assn^k \rightarrow mt\text{x-}assn$
 $\langle proof \rangle$

lemma [*sepref-import-param*]:

$(Le\ 0, 0) \in Id$

$\langle proof \rangle$

sepref-register 0

sepref-definition *check-diag-impl'* is

$uncurry (RETURN \circ\circ check\text{-}diag) ::$
 $[\lambda(i, -). i \leq n]_a nat\text{-}assn^k *_a mt\text{x-}assn^k \rightarrow bool\text{-}assn$
 $\langle proof \rangle$

lemma [*sepref-opt-simps*]:

$(x = True) = x$

$\langle proof \rangle$

sepref-definition *dbm-subset'-impl2* is

$uncurry2 (RETURN \circ\circ\circ dbm\text{-}subset') ::$
 $[\lambda((i, -), -). i \leq n]_a nat\text{-}assn^k *_a mt\text{x-}assn^k *_a mt\text{x-}assn^k \rightarrow bool\text{-}assn$
 $\langle proof \rangle$

definition

$dbm\text{-}subset'\text{-}impl' \equiv \lambda m\ a\ b.$

do {

imp-for 0 ((m + 1) * (m + 1)) Heap-Monad.return

($\lambda i\ -. do \{$

$x \leftarrow Array.nth\ a\ i; y \leftarrow Array.nth\ b\ i; Heap\text{-}Monad.return\ (x \leq y)$

$\})$

True

}

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

$\langle a \mapsto_a xs * b \mapsto_a ys \rangle$

imp-for 0 n' Heap-Monad.return

($\lambda i\ -. do \{$

$x \leftarrow Array.nth\ a\ i; y \leftarrow Array.nth\ b\ i; Heap\text{-}Monad.return\ (P\ x\ y)$

$\})$

$True$
 $\langle \lambda r. \uparrow(r \longleftrightarrow list-all2 P (take\ n'\ xs)\ (take\ n'\ ys)) * a \mapsto_a xs * b \mapsto_a ys \rangle_t$
if $n' \leq length\ xs\ n' \leq length\ ys$
 $\langle proof \rangle$

lemma *dbm-subset'-impl'-refine*:

$(uncurry2\ dbm-subset'-impl',\ uncurry2\ (RETURN\ \circ\circ\circ\ dbm-subset'))$
 $\in [\lambda((i, -), -). i = n]_a\ nat-assn^k * _a\ local.mtx-assn^k * _a\ local.mtx-assn^k \rightarrow$
 $bool-assn$
 $\langle proof \rangle$

sempref-register *check-diag* ::

$nat \Rightarrow - :: \{linordered-cancel-ab-monoid-add, heap\}\ DBMEntry\ i-mtx \Rightarrow$
 $bool$

sempref-register *dbm-subset'* ::

$nat \Rightarrow 'a :: \{linordered-cancel-ab-monoid-add, heap\}\ DBMEntry\ i-mtx \Rightarrow$
 $'a\ DBMEntry\ i-mtx \Rightarrow bool$

lemmas [*sempref-fr-rules*] = *dbm-subset'-impl'-refine check-diag-impl'.refine*

sempref-definition *dbm-subset-impl'* **is**

$uncurry2\ (RETURN\ \circ\circ\circ\ dbm-subset) ::$
 $[\lambda((i, -), -). i = n]_a\ nat-assn^k * _a\ mtx-assn^k * _a\ mtx-assn^k \rightarrow bool-assn$
 $\langle proof \rangle$

context

notes [*id-rules*] = *itypeI*[of $n\ TYPE\ (nat)$]

and [*sempref-import-param*] = *IdI*[of n]

begin

sempref-definition *dbm-subset-impl* **is**

$uncurry\ (RETURN\ \circ\circ\ PR-CONST\ (dbm-subset\ n)) :: mtx-assn^k * _a\ mtx-assn^k$
 $\rightarrow_a\ bool-assn$
 $\langle proof \rangle$

sempref-definition *check-diag-impl* **is**

$RETURN\ \circ\ PR-CONST\ (check-diag\ n) :: mtx-assn^k \rightarrow_a\ bool-assn$
 $\langle proof \rangle$

sempref-definition *dbm-subset'-impl* **is**

$uncurry\ (RETURN\ \circ\circ\ PR-CONST\ (dbm-subset'\ n)) :: mtx-assn^k * _a\ mtx-assn^k$
 $\rightarrow_a\ bool-assn$
 $\langle proof \rangle$

end

abbreviation

$iarray-assn\ x\ y \equiv pure\ (br\ IArray\ (\lambda-. True))\ y\ x$

lemma [sepref-fr-rules]:

$(uncurry\ (return\ ooo\ IArray.sub),\ uncurry\ (RETURN\ ooo\ op-list-get))$
 $\in\ iarray-assn^k\ *_a\ id-assn^k\ \rightarrow_a\ id-assn$
 $\langle proof \rangle$

lemmas $extra-defs = extra-upd-def\ upd-line-def\ upd-line-0-def$

sepref-definition $norm-upd-impl$ **is**

$uncurry2\ (RETURN\ ooo\ norm-upd) ::$
 $[\lambda((-,\ xs),\ i). length\ xs > n \wedge i \leq n]_a\ mtx-assn^d\ *_a\ iarray-assn^k\ *_a$
 $nat-assn^k\ \rightarrow\ mtx-assn$
 $\langle proof \rangle$

sepref-definition $norm-upd-impl'$ **is**

$uncurry2\ (RETURN\ ooo\ norm-upd) ::$
 $[\lambda((-,\ xs),\ i). length\ xs > n \wedge i \leq n]_a\ mtx-assn^d\ *_a\ (list-assn\ id-assn)^k\ *_a$
 $nat-assn^k\ \rightarrow\ mtx-assn$
 $\langle proof \rangle$

sepref-definition $extra-lu-upd-impl$ **is**

$uncurry3\ (\lambda x. RETURN\ ooo\ (extra-lu-upd\ x)) ::$
 $[\lambda(((,\ ys),\ xs),\ i). length\ xs > n \wedge length\ ys > n \wedge i \leq n]_a$
 $mtx-assn^d\ *_a\ iarray-assn^k\ *_a\ iarray-assn^k\ *_a\ nat-assn^k\ \rightarrow\ mtx-assn$
 $\langle proof \rangle$

sepref-definition $mtx-line-to-list-impl$ **is**

$uncurry\ (RETURN\ ooo\ PR-CONST\ mtx-line) ::$
 $[\lambda(m,\ -). m \leq n]_a\ nat-assn^k\ *_a\ mtx-assn^k\ \rightarrow\ list-assn\ id-assn$
 $\langle proof \rangle$

context

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

begin

sepref-definition $mtx-line-to-list-impl2$ **is**

$RETURN\ o\ PR-CONST\ mtx-line\ m :: mtx-assn^k\ \rightarrow_a\ list-assn\ id-assn$

$\langle \text{proof} \rangle$

end

lemma *IArray-impl*:

$(\text{return } o \text{ IArray}, \text{ RETURN } o \text{ id}) \in (\text{list-assn id-assn})^k \rightarrow_a \text{iarray-assn}$
 $\langle \text{proof} \rangle$

definition

$\text{mtx-line-to-iarray-impl } m \ M = (\text{mtx-line-to-list-impl2 } m \ M \gg= \text{return } o \text{ IArray})$

lemmas *mtx-line-to-iarray-impl-ht* =

$\text{mtx-line-to-list-impl2.refine}[\text{to-hnr}, \text{unfolded hn-refine-def hn-ctxt-def}, \text{simplified}]$

lemmas *IArray-ht* = *IArray-impl* $[\text{to-hnr}, \text{unfolded hn-refine-def hn-ctxt-def}, \text{simplified}]$

lemma *mtx-line-to-iarray-impl-refine* $[\text{sepref-fr-rules}]$:

$(\text{uncurry } \text{mtx-line-to-iarray-impl}, \text{uncurry } (\text{RETURN } \circ o \text{ mtx-line}))$
 $\in [\lambda(m, -). m \leq n]_a \text{ nat-assn}^k *_a \text{ mtx-assn}^k \rightarrow \text{iarray-assn}$
 $\langle \text{proof} \rangle$

sepref-register *mtx-line* :: $\text{nat} \Rightarrow ('ef) \text{ DBMEntry } i\text{-mtx} \Rightarrow 'ef \text{ DBMEntry list}$

lemma $[\text{sepref-import-param}]$: $(\text{dbm-lt} :: - \text{ DBMEntry} \Rightarrow -, \text{dbm-lt}) \in \text{Id} \rightarrow \text{Id} \rightarrow \text{Id} \langle \text{proof} \rangle$

sepref-definition *extra-lup-upd-impl* is

$\text{uncurry3 } (\lambda x. \text{RETURN } o o o (\text{extra-lup-upd } x)) ::$
 $[\lambda((-, \text{ys}), \text{xs}), i]. \text{length } \text{xs} > n \wedge \text{length } \text{ys} > n \wedge i \leq n]_a$
 $\text{mtx-assn}^d *_a \text{iarray-assn}^k *_a \text{iarray-assn}^k *_a \text{nat-assn}^k \rightarrow \text{mtx-assn}$
 $\langle \text{proof} \rangle$

context

notes $[\text{id-rules}] = \text{itypeI}[\text{of } n \ \text{TYPE} \ (\text{nat})]$
and $[\text{sepref-import-param}] = \text{IdI}[\text{of } n]$

begin

definition

$\text{unbounded-dbm}' = \text{unbounded-dbm } n$

lemma *unbounded-dbm-alt-def*:

$unbounded-dbm\ n = op-amtx-new\ (Suc\ n)\ (Suc\ n)\ (unbounded-dbm')$
 $\langle proof \rangle$

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

lemma [*sepref-fr-rules*]:

$(uncurry0\ (return\ unbounded-dbm'),\ uncurry0\ (RETURN\ (PR-CONST\ (unbounded-dbm'))))$
 $\in\ unit-assn^k \rightarrow_a\ pure\ (nat-rel \times_r\ nat-rel \rightarrow Id)$
 $\langle proof \rangle$

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*:

$mtx-nonzero\ unbounded-dbm' \subseteq \{0..<Suc\ n\} \times \{0..<Suc\ n\}$
 $\langle proof \rangle$

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

lemma *unbounded-dbm'-bounded-1*:

$(a,\ b) \in\ mtx-nonzero\ unbounded-dbm' \implies a < Suc\ n$
 $\langle proof \rangle$

lemma *unbounded-dbm'-bounded-2*:

$(a,\ b) \in\ mtx-nonzero\ unbounded-dbm' \implies b < Suc\ n$
 $\langle proof \rangle$

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*]:

$dbm-subset\ \$\ n \equiv PR-CONST\ (dbm-subset\ n)$
 $\langle proof \rangle$

sepref-definition *unbounded-dbm-impl* **is**

$uncurry0\ (RETURN\ (PR-CONST\ (unbounded-dbm\ n))) :: unit-assn^k \rightarrow_a\ mtx-assn$
 $\langle proof \rangle$

DBM to List

definition *dbm-to-list* :: (nat × nat ⇒ 'a) ⇒ 'a list **where**

dbm-to-list M ≡

rev \$ fold (λi xs. fold (λj xs. M (i, j) # xs) [0..

sempref-definition *dbm-to-list-impl* **is**

RETURN o PR-CONST *dbm-to-list* :: mtx-assn^k →_a list-assn id-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 (λe (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-assnk →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* ($\langle \cdot := \cdot \rangle$ 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 = ((($\lambda(i, j). Le\ 0$)($(1, 2) := Le\ 2$))($(0, 2) := Le\ (-1)$))($(1, 0) := \infty$))($(2, 0) := \infty$)

— Pretty-printing

definition *show-test-dbm* **where**

show-test-dbm *M* = *String.implode* (
 show-dbm 2
 ($\lambda i. \text{if } i = 1 \text{ then } "x" \text{ else if } i = 2 \text{ then } "y" \text{ else } "f"$) *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 preserves 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 preserves 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 $\lambda(i, j). \infty$ 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.