A formal proof of the max-flow min-cut theorem for countable networks

Andreas Lochbihler

February 23, 2021

Abstract

This article formalises a proof of the maximum-flow minimal-cut theorem for networks with countably many edges. A network is a directed graph with non-negative real-valued edge labels and two dedicated vertices, the source and the sink. A flow in a network assigns non-negative real numbers to the edges such that for all vertices except for the source and the sink, the sum of values on incoming edges equals the sum of values on outgoing edges. A cut is a subset of the vertices which contains the source, but not the sink. Our theorem states that in every network, there is a flow and a cut such that the flow saturates all the edges going out of the cut and is zero on all the incoming edges. The proof is based on the paper "The Max-Flow Min-Cut theorem for countable networks" by Aharoni et al. [2].

Additionally, we prove a characterisation of the lifting operation for relations on discrete probability distributions, which leads to a concise proof of its distributivity over relation composition.

Contents

1 Preliminaries

theory *MFMC-Misc* **imports**
HOL−Probability.Probability
HOL−Library.Transitive-Closure-Table
HOL−Library.Complete-Partial-Order2
HOL−Library.Bourbaki-Witt-Fixpoint
begin

hide-const (**open**) *cycle*
hide-const (**open**) *path*
hide-const (**open**) *cut*
hide-const (**open**) *orthogonal*

lemmas *disjE* [*consumes 1*, *case-names left right*, *cases pred*] = *disjE*

lemma *inj-on-Pair2* [*simp*]: *inj-on* (*Pair x*) *A*
⟨*proof*⟩

lemma *inj-on-Pair1* [*simp*]: *inj-on* (λ*x*. (*x*, *y*)) *A*
⟨*proof*⟩

lemma *inj-map-prod′*: ⟦ *inj f*; *inj g* ⟧ ⟹ *inj-on* (*map-prod f g*) *X*
⟨*proof*⟩

lemma *not-range-Inr*: *x* ∉ *range Inr* ⟷ *x* ∈ *range Inl*
⟨*proof*⟩

lemma *not-range-Inl*: *x* ∉ *range Inl* ⟷ *x* ∈ *range Inr*
⟨*proof*⟩

lemma *Chains-into-chain*: *M* ∈ *Chains* {(*x*, *y*). *R x y*} ⟹ *Complete-Partial-Order.chain R M*
⟨*proof*⟩

lemma *chain-dual*: *Complete-Partial-Order.chain* (≥) = *Complete-Partial-Order.chain* (≤)
⟨*proof*⟩

lemma *Cauchy-real-Suc-diff*:
fixes *X* :: *nat* ⇒ *real* **and** *x* :: *real*
assumes *bounded*: ⋀*n*. |*f* (*Suc n*) − *f n*| ≤ (*c* / *x* ^ *n*)
and *x*: *1* < *x*
shows *Cauchy f*
⟨*proof*⟩

lemma *complete-lattice-ccpo-dual*:
class.ccpo Inf (≥) ((>) :: - :: *complete-lattice* ⇒ -)
⟨*proof*⟩

lemma *card-eq-1-iff*: $card\ A = Suc\ 0 \longleftrightarrow (\exists x.\ A = \{x\})$
⟨*proof*⟩

lemma *nth-rotate1*: $n < length\ xs \Longrightarrow rotate1\ xs\ !\ n = xs\ !\ (Suc\ n\ mod\ length\ xs)$
⟨*proof*⟩

lemma *set-zip-rightI*: $[\![\ x \in set\ ys;\ length\ xs \geq length\ ys\ ]\!] \Longrightarrow \exists z.\ (z,\ x) \in set\ (zip\ xs\ ys)$
⟨*proof*⟩

lemma *map-eq-append-conv*:
$map\ f\ xs = ys\ @\ zs \longleftrightarrow (\exists ys'\ zs'.\ xs = ys'\ @\ zs' \wedge ys = map\ f\ ys' \wedge zs = map\ f\ zs')$
⟨*proof*⟩

lemma *rotate1-append*:
$rotate1\ (xs\ @\ ys) = (if\ xs = []\ then\ rotate1\ ys\ else\ tl\ xs\ @\ ys\ @\ [hd\ xs])$
⟨*proof*⟩

lemma *in-set-tlD*: $x \in set\ (tl\ xs) \Longrightarrow x \in set\ xs$
⟨*proof*⟩

lemma *countable-converseI*:
assumes *countable A*
shows *countable* (*converse A*)
⟨*proof*⟩

lemma *countable-converse* [*simp*]: *countable* (*converse A*) ⟷ *countable A*
⟨*proof*⟩

lemma *nn-integral-count-space-reindex*:
inj-on f A ⟹$(\int^+ y.\ g\ y\ \partial count\text{-}space\ (f\ `\ A)) = (\int^+ x.\ g\ (f\ x)\ \partial count\text{-}space\ A)$
⟨*proof*⟩

syntax
-nn-sum :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b*::*comm-monoid-add* ((2$\sum^+$ -∈-./ -) [0, 51, 10] 10)
-nn-sum-UNIV :: *pttrn* ⇒ *'b* ⇒ *'b*::*comm-monoid-add* ((2$\sum^+$ -./ -) [0, 10] 10)
translations
$\sum^+ i{\in}A.\ b \rightleftharpoons CONST\ nn\text{-}integral\ (CONST\ count\text{-}space\ A)\ (\lambda i.\ b)$
$\sum^+ i.\ b \rightleftharpoons \sum^+ i{\in}CONST\ UNIV.\ b$

inductive-simps *rtrancl-path-simps*:
rtrancl-path R x [] *y*
rtrancl-path R x (*a* # *bs*) *y*

definition *restrict-rel* :: *'a set* ⇒ (*'a* × *'a*) *set* ⇒ (*'a* × *'a*) *set*
where *restrict-rel A R* = $\{(x,\ y){\in}R.\ x \in A \wedge y \in A\}$

lemma *in-restrict-rel-iff*: $(x, y) \in \mathit{restrict\text{-}rel}\ A\ R \longleftrightarrow (x, y) \in R \wedge x \in A \wedge y \in A$
⟨*proof*⟩

lemma *restrict-relE*: $[\![\ (x, y) \in \mathit{restrict\text{-}rel}\ A\ R;\ [\![\ (x, y) \in R;\ x \in A;\ y \in A\ ]\!] \Longrightarrow \mathit{thesis}\ ]\!] \Longrightarrow \mathit{thesis}$
⟨*proof*⟩

lemma *restrict-relI* [*intro*!]: $[\![\ (x, y) \in R;\ x \in A;\ y \in A\ ]\!] \Longrightarrow (x, y) \in \mathit{restrict\text{-}rel}\ A\ R$
⟨*proof*⟩

lemma *Field-restrict-rel-subset*: $\mathit{Field}\ (\mathit{restrict\text{-}rel}\ A\ R) \subseteq A \cap \mathit{Field}\ R$
⟨*proof*⟩

lemma *Field-restrict-rel* [*simp*]: $\mathit{Refl}\ R \Longrightarrow \mathit{Field}\ (\mathit{restrict\text{-}rel}\ A\ R) = A \cap \mathit{Field}\ R$
⟨*proof*⟩

lemma *Partial-order-restrict-rel*:
assumes *Partial-order R*
shows *Partial-order* (*restrict-rel A R*)
⟨*proof*⟩

lemma *Chains-restrict-relD*: $M \in \mathit{Chains}\ (\mathit{restrict\text{-}rel}\ A\ \mathit{leq}) \Longrightarrow M \in \mathit{Chains}\ \mathit{leq}$
⟨*proof*⟩

lemma *bourbaki-witt-fixpoint-restrict-rel*:
assumes *leq*: *Partial-order leq*
and *chain-Field*: $\bigwedge M.\ [\![\ M \in \mathit{Chains}\ (\mathit{restrict\text{-}rel}\ A\ \mathit{leq});\ M \neq \{\}\ ]\!] \Longrightarrow \mathit{lub}\ M \in A$
and *lub-least*: $\bigwedge M\ z.\ [\![\ M \in \mathit{Chains}\ \mathit{leq};\ M \neq \{\};\ \bigwedge x.\ x \in M \Longrightarrow (x, z) \in \mathit{leq}\ ]\!] \Longrightarrow (\mathit{lub}\ M, z) \in \mathit{leq}$
and *lub-upper*: $\bigwedge M\ z.\ [\![\ M \in \mathit{Chains}\ \mathit{leq};\ z \in M\ ]\!] \Longrightarrow (z, \mathit{lub}\ M) \in \mathit{leq}$
and *increasing*: $\bigwedge x.\ [\![\ x \in A;\ x \in \mathit{Field}\ \mathit{leq}\ ]\!] \Longrightarrow (x, f\ x) \in \mathit{leq} \wedge f\ x \in A$
shows *bourbaki-witt-fixpoint lub* (*restrict-rel A leq*) *f*
⟨*proof*⟩

lemma *Field-le* [*simp*]: $\mathit{Field}\ \{(x :: - :: \mathit{preorder}, y).\ x \leq y\} = \mathit{UNIV}$
⟨*proof*⟩

lemma *Field-ge* [*simp*]: $\mathit{Field}\ \{(x :: - :: \mathit{preorder}, y).\ y \leq x\} = \mathit{UNIV}$
⟨*proof*⟩

lemma *refl-le* [*simp*]: $\mathit{refl}\ \{(x :: - :: \mathit{preorder}, y).\ x \leq y\}$
⟨*proof*⟩

lemma *refl-ge* [*simp*]: $\mathit{refl}\ \{(x :: - :: \mathit{preorder}, y).\ y \leq x\}$

⟨*proof*⟩

lemma *partial-order-le* [*simp*]: *partial-order-on UNIV* {(*x* :: - :: *order*, *x'*). *x* ≤ *x'*}
⟨*proof*⟩

lemma *partial-order-ge* [*simp*]: *partial-order-on UNIV* {(*x* :: - :: *order*, *x'*). *x'* ≤ *x*}
⟨*proof*⟩

lemma *incseq-chain-range*: *incseq f* ⟹ *Complete-Partial-Order.chain* (≤) (*range f*)
⟨*proof*⟩

end

theory *MFMC-Finite* **imports**
EdmondsKarp-Maxflow.EdmondsKarp-Termination-Abstract
HOL−Library.While-Combinator
begin

2 Existence of maximum flows and minimal cuts in finite graphs

This theory derives the existencs of a maximal flow or a minimal cut for finite graphs from the termination proof of the Edmonds-Karp algorithm.

context *Graph* **begin**

lemma *outgoing-outside*: *x* ∉ *V* ⟹ *outgoing x* = {}
⟨*proof*⟩

lemma *incoming-outside*: *x* ∉ *V* ⟹ *incoming x* = {}
⟨*proof*⟩

end

context *NFlow* **begin**

lemma *conservation*: ⟦ *x* ≠ *s*; *x* ≠ *t* ⟧ ⟹ *sum f* (*incoming x*) = *sum f* (*outgoing x*)
⟨*proof*⟩

lemma *augmenting-path-imp-shortest*:
isAugmentingPath p ⟹ ∃ *p*. *Graph.isShortestPath cf s p t*
⟨*proof*⟩

lemma *shortest-is-augmenting*:

Graph.isShortestPath cf s p t $\Longrightarrow$ *isAugmentingPath p*
⟨*proof*⟩

definition *augment-with-path p* ≡ *augment* (*augmentingFlow p*)

end

context *Network* **begin**

definition *shortest-augmenting-path f* = (*SOME p. Graph.isShortestPath* (*residualGraph c f*) *s p t*)

lemma *shortest-augmenting-path*:
assumes *NFlow c s t f*
and ∃ *p. NPreflow.isAugmentingPath c s t f p*
shows *Graph.isShortestPath* (*residualGraph c f*) *s* (*shortest-augmenting-path f*) *t*
⟨*proof*⟩

definition *max-flow* **where**
max-flow = *while*
(λ*f*. ∃ *p. NPreflow.isAugmentingPath c s t f p*)
(λ*f*. *NFlow.augment-with-path c f* (*shortest-augmenting-path f*)) (λ-. *0*)

lemma *max-flow*:
NFlow c s t max-flow (**is** *?thesis1*)
¬ (∃ *p. NPreflow.isAugmentingPath c s t max-flow p*) (**is** *?thesis2*)
⟨*proof*⟩

end

end

3 Matrices for given marginals

This theory derives from the finite max-flow min-cut theorem the existence of matrices with given marginals based on a proof by Georg Kellerer [4].

theory *Matrix-For-Marginals*
imports *MFMC-Misc HOL−Library.Diagonal-Subsequence MFMC-Finite*
begin

lemma *bounded-matrix-for-marginals-finite*:
fixes *f g* :: *nat* ⇒ *real*
and *n* :: *nat*
and *R* :: (*nat* × *nat*) *set*
assumes *eq-sum*: *sum f* {..*n*} = *sum g* {..*n*}
and *le*: ⋀*X*. *X* ⊆ {..*n*} $\Longrightarrow$ *sum f X* ≤ *sum g* (*R* `` *X*)
and *f-nonneg*: ⋀*x*. *0* ≤ *f x*

and *g-nonneg*: $\bigwedge y.\ 0 \le g\ y$
and *R*: $R \subseteq \{..n\} \times \{..n\}$
obtains $h :: nat \Rightarrow nat \Rightarrow real$
where $\bigwedge x\ y.\ [\![\ x \le n;\ y \le n\ ]\!] \Longrightarrow 0 \le h\ x\ y$
and $\bigwedge x\ y.\ [\![\ 0 < h\ x\ y;\ x \le n;\ y \le n\ ]\!] \Longrightarrow (x,\ y) \in R$
and $\bigwedge x.\ x \le n \Longrightarrow f\ x = sum\ (h\ x)\ \{..n\}$
and $\bigwedge y.\ y \le n \Longrightarrow g\ y = sum\ (\lambda x.\ h\ x\ y)\ \{..n\}$
⟨*proof*⟩

lemma *convergent-bounded-family-nat*:
fixes $f :: nat \Rightarrow nat \Rightarrow real$
assumes *bounded*: $\bigwedge x.\ bounded\ (range\ (\lambda n.\ f\ n\ x))$
obtains k **where** *strict-mono* k $\bigwedge x.\ convergent\ (\lambda n.\ f\ (k\ n)\ x)$
⟨*proof*⟩

lemma *convergent-bounded-family*:
fixes $f :: nat \Rightarrow 'a \Rightarrow real$
assumes *bounded*: $\bigwedge x.\ x \in A \Longrightarrow bounded\ (range\ (\lambda n.\ f\ n\ x))$
and A: *countable A*
obtains k **where** *strict-mono* k $\bigwedge x.\ x \in A \Longrightarrow convergent\ (\lambda n.\ f\ (k\ n)\ x)$
⟨*proof*⟩

abbreviation *zero-on* :: $('a \Rightarrow 'b :: zero) \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow 'b$
where *zero-on* $f \equiv override\text{-}on\ f\ (\lambda\text{-}.\ 0)$

lemma *zero-on-le* [*simp*]: **fixes** $f :: 'a \Rightarrow 'b :: \{preorder,\ zero\}$ **shows**
$zero\text{-}on\ f\ X\ x \le f\ x \longleftrightarrow (x \in X \longrightarrow 0 \le f\ x)$
⟨*proof*⟩

lemma *zero-on-nonneg*: **fixes** $f :: 'a \Rightarrow 'b :: \{preorder,\ zero\}$ **shows**
$0 \le zero\text{-}on\ f\ X\ x \longleftrightarrow (x \notin X \longrightarrow 0 \le f\ x)$
⟨*proof*⟩

lemma *sums-zero-on*:
fixes $f :: nat \Rightarrow 'a :: real\text{-}normed\text{-}vector$
assumes f: *f sums s*
and X: *finite X*
shows *zero-on f X sums* $(s - sum\ f\ X)$
⟨*proof*⟩

lemma
fixes $f :: nat \Rightarrow 'a :: real\text{-}normed\text{-}vector$
assumes f: *summable f*
and X: *finite X*
shows *summable-zero-on* [*simp*]: *summable* (*zero-on f X*) (**is** *?thesis1*)
and *suminf-zero-on*: *suminf* (*zero-on f X*) = *suminf f* − *sum f X* (**is** *?thesis2*)
⟨*proof*⟩

lemma *summable-zero-on-nonneg*:

fixes $f :: nat \Rightarrow 'a :: \{ordered\text{-}comm\text{-}monoid\text{-}add, linorder\text{-}topology, conditionally\text{-}complete\text{-}linorder\}$
assumes *f*: *summable f*
and *nonneg*: $\bigwedge x.\ 0 \leq f\ x$
shows *summable* (*zero-on f X*)
⟨*proof*⟩

lemma *zero-on-ennreal* [*simp*]: *zero-on* ($\lambda x.$ *ennreal* (*f x*)) *A* = ($\lambda x.$ *ennreal* (*zero-on f A x*))
⟨*proof*⟩

lemma *sum-lessThan-conv-atMost-nat*:
fixes *f* :: *nat* ⇒ *'b* :: *ab-group-add*
shows *sum f* {..<*n*} = *sum f* {..*n*} − *f n*
⟨*proof*⟩

lemma *Collect-disjoint-atLeast*:
Collect P ∩ {*x*..} = {} ⟷ (∀ *y*≥*x*. ¬ *P y*)
⟨*proof*⟩

lemma *bounded-matrix-for-marginals-nat*:
fixes *f g* :: *nat* ⇒ *real*
and *R* :: (*nat* × *nat*) *set*
and *s* :: *real*
assumes *sum-f*: *f sums s* **and** *sum-g*: *g sums s*
and *f-nonneg*: $\bigwedge x.\ 0 \leq f\ x$ **and** *g-nonneg*: $\bigwedge y.\ 0 \leq g\ y$
and *f-le-g*: $\bigwedge X.$ *suminf* (*zero-on f* (− *X*)) ≤ *suminf* (*zero-on g* (− *R* `` *X*))
obtains *h* :: *nat* ⇒ *nat* ⇒ *real*
where $\bigwedge x\ y.\ 0 \leq h\ x\ y$
and $\bigwedge x\ y.\ 0 < h\ x\ y \Longrightarrow (x, y) \in R$
and $\bigwedge x.$ *h x sums f x*
and $\bigwedge y.$ ($\lambda x.$ *h x y*) *sums g y*
⟨*proof*⟩

lemma *bounded-matrix-for-marginals-ennreal*:
assumes *sum-eq*: ($\sum^+$ *x*∈*A*. *f x*) = ($\sum^+$ *y*∈*B*. *g y*)
and *finite*: ($\sum^+$ *x*∈*B*. *g x*) ≠ ⊤
and *le*: $\bigwedge X.$ *X* ⊆ *A* ⟹ ($\sum^+$ *x*∈*X*. *f x*) ≤ ($\sum^+$ *y*∈*R* `` *X*. *g y*)
and *countable* [*simp*]: *countable A countable B*
and *R*: *R* ⊆ *A* × *B*
obtains *h* **where** $\bigwedge x\ y.\ 0 < h\ x\ y \Longrightarrow (x, y) \in R$
and $\bigwedge x\ y.\ h\ x\ y \neq \top$
and $\bigwedge x.\ x \in A \Longrightarrow (\sum^+ y \in B.\ h\ x\ y) = f\ x$
and $\bigwedge y.\ y \in B \Longrightarrow (\sum^+ x \in A.\ h\ x\ y) = g\ y$
⟨*proof*⟩

end
theory *MFMC-Network* **imports**
MFMC-Misc
begin

4 Graphs

type-synonym *'v edge* = *'v* × *'v*

record *'v graph* =
  edge :: *'v* ⇒ *'v* ⇒ *bool*

abbreviation *edges* :: (*'v*, *'more*) *graph-scheme* ⇒ *'v edge set* (**E**ı)
where $\mathbf{E}_G \equiv \{(x,\ y).\ edge\ G\ x\ y\}$

definition *outgoing* :: (*'v*, *'more*) *graph-scheme* ⇒ *'v* ⇒ *'v set* (**OUT**ı)
where $\mathbf{OUT}_G\ x = \{y.\ (x,\ y) \in \mathbf{E}_G\}$

definition *incoming* :: (*'v*, *'more*) *graph-scheme* ⇒ *'v* ⇒ *'v set* (**IN**ı)
where $\mathbf{IN}_G\ y = \{x.\ (x,\ y) \in \mathbf{E}_G\}$

Vertices are implicitly defined as the endpoints of edges, so we do not allow isolated vertices. For the purpose of flows, this does not matter as isolated vertices cannot contribute to a flow. The advantage is that we do not need any invariant on graphs that the endpoints of edges are a subset of the vertices. Conversely, this design choice makes a few proofs about reductions on webs harder, because we have to adjust other sets which are supposed to be part of the vertices.

definition *vertex* :: (*'v*, *'more*) *graph-scheme* ⇒ *'v* ⇒ *bool*
where *vertex G x* ⟷ *Domainp* (*edge G*) *x* ∨ *Rangep* (*edge G*) *x*

lemma *vertexI*:
  shows *vertexI1*: *edge* Γ *x y* ⟹ *vertex* Γ *x*
  and *vertexI2*: *edge* Γ *x y* ⟹ *vertex* Γ *y*
⟨*proof*⟩

abbreviation *vertices* :: (*'v*, *'more*) *graph-scheme* ⇒ *'v set* (**V**ı)
where $\mathbf{V}_G \equiv Collect\ (vertex\ G)$

lemma V-*def*: $\mathbf{V}_G = fst\ `\ \mathbf{E}_G \cup snd\ `\ \mathbf{E}_G$
⟨*proof*⟩

type-synonym *'v path* = *'v list*

abbreviation *path* :: (*'v*, *'more*) *graph-scheme* ⇒ *'v* ⇒ *'v path* ⇒ *'v* ⇒ *bool*
where *path G* ≡ *rtrancl-path* (*edge G*)

inductive *cycle* :: (*'v*, *'more*) *graph-scheme* ⇒ *'v path* ⇒ *bool*
  for *G*
where — Cycles must not pass through the same node multiple times. Otherwise, the cycle might enter a node via two different edges and leave it via just one edge. Thus, the clean-up lemma would not hold any more.
  cycle: ⟦ *path G v p v*; *p* ≠ []; *distinct p* ⟧ ⟹ *cycle G p*

inductive-simps *cycle-Nil* [*simp*]: *cycle G Nil*

abbreviation *cycles* :: (*'v*, *'more*) *graph-scheme* ⇒ *'v path set*
where *cycles G* ≡ *Collect* (*cycle G*)

lemma *countable-cycles* [*simp*]:
assumes *countable* ($\mathbf{V}_G$)
shows *countable* (*cycles G*)
⟨*proof*⟩

definition *cycle-edges* :: *'v path* ⇒ *'v edge list*
where *cycle-edges p* = *zip p* (*rotate1 p*)

lemma *cycle-edges-not-Nil*: *cycle G p* ⟹ *cycle-edges p* ≠ []
⟨*proof*⟩

lemma *distinct-cycle-edges*:
cycle G p ⟹ *distinct* (*cycle-edges p*)
⟨*proof*⟩

lemma *cycle-enter-leave-same*:
assumes *cycle G p*
shows *card* (*set* [(*x'*, *y*) ← *cycle-edges p*. *x'* = *x*]) = *card* (*set* [(*x'*, *y*) ← *cycle-edges p*. *y* = *x*])
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *cycle-leave-ex-enter*:
assumes *cycle G p* **and** (*x*, *y*) ∈ *set* (*cycle-edges p*)
shows ∃ *z*. (*z*, *x*) ∈ *set* (*cycle-edges p*)
⟨*proof*⟩

lemma *cycle-edges-edges*:
assumes *cycle G p*
shows *set* (*cycle-edges p*) ⊆ $\mathbf{E}_G$
⟨*proof*⟩

5 Network and Flow

record *'v network* = *'v graph* +
capacity :: *'v edge* ⇒ *ennreal*
source :: *'v*
sink :: *'v*

type-synonym *'v flow* = *'v edge* ⇒ *ennreal*

inductive-set *support-flow* :: *'v flow* ⇒ *'v edge set*
for *f*
where *f e* > *0* ⟹ *e* ∈ *support-flow f*

lemma *support-flow-conv*: *support-flow f* = {*e*. *f e* > *0*}
⟨*proof*⟩

lemma *not-in-support-flowD*: *x* ∉ *support-flow f* ⟹ *f x* = *0*
⟨*proof*⟩

definition *d-OUT* :: *'v flow* ⇒ *'v* ⇒ *ennreal*
where *d-OUT g x* = (∑⁺ *y*. *g* (*x*, *y*))

definition *d-IN* :: *'v flow* ⇒ *'v* ⇒ *ennreal*
where *d-IN g y* = (∑⁺ *x*. *g* (*x*, *y*))

lemma *d-OUT-mono*: (⋀*y*. *f* (*x*, *y*) ≤ *g* (*x*, *y*)) ⟹ *d-OUT f x* ≤ *d-OUT g x*
⟨*proof*⟩

lemma *d-IN-mono*: (⋀*x*. *f* (*x*, *y*) ≤ *g* (*x*, *y*)) ⟹ *d-IN f y* ≤ *d-IN g y*
⟨*proof*⟩

lemma *d-OUT-0* [*simp*]: *d-OUT* (λ-. *0*) *x* = *0*
⟨*proof*⟩

lemma *d-IN-0* [*simp*]: *d-IN* (λ-. *0*) *x* = *0*
⟨*proof*⟩

lemma *d-OUT-add*: *d-OUT* (λ*e*. *f e* + *g e*) *x* = *d-OUT f x* + *d-OUT g x*
⟨*proof*⟩

lemma *d-IN-add*: *d-IN* (λ*e*. *f e* + *g e*) *x* = *d-IN f x* + *d-IN g x*
⟨*proof*⟩

lemma *d-OUT-cmult*: *d-OUT* (λ*e*. *c* ∗ *f e*) *x* = *c* ∗ *d-OUT f x*
⟨*proof*⟩

lemma *d-IN-cmult*: *d-IN* (λ*e*. *c* ∗ *f e*) *x* = *c* ∗ *d-IN f x*
⟨*proof*⟩

lemma *d-OUT-ge-point*: *f* (*x*, *y*) ≤ *d-OUT f x*
⟨*proof*⟩

lemma *d-IN-ge-point*: *f* (*y*, *x*) ≤ *d-IN f x*
⟨*proof*⟩

lemma *d-OUT-monotone-convergence-SUP*:
assumes *incseq* (λ*n y*. *f n* (*x*, *y*))
shows *d-OUT* (λ*e*. *SUP n*. *f n e*) *x* = (*SUP n*. *d-OUT* (*f n*) *x*)
⟨*proof*⟩

lemma *d-IN-monotone-convergence-SUP*:

assumes *incseq* $(\lambda n\ x.\ f\ n\ (x,\ y))$
shows *d-IN* $(\lambda e.\ SUP\ n.\ f\ n\ e)\ y = (SUP\ n.\ d\text{-}IN\ (f\ n)\ y)$
⟨*proof*⟩

lemma *d-OUT-diff*:
assumes $\bigwedge y.\ g\ (x,\ y) \leq f\ (x,\ y)$ *d-OUT g x* $\neq \top$
shows *d-OUT* $(\lambda e.\ f\ e - g\ e)\ x =$ *d-OUT f x* $-$ *d-OUT g x*
⟨*proof*⟩

lemma *d-IN-diff*:
assumes $\bigwedge x.\ g\ (x,\ y) \leq f\ (x,\ y)$ *d-IN g y* $\neq \top$
shows *d-IN* $(\lambda e.\ f\ e - g\ e)\ y =$ *d-IN f y* $-$ *d-IN g y*
⟨*proof*⟩

lemma fixes *G* (**structure**)
shows *d-OUT-alt-def*: $(\bigwedge y.\ (x,\ y) \notin \mathbf{E} \Longrightarrow g\ (x,\ y) = 0) \Longrightarrow$ *d-OUT g x* $= (\sum^+ y{\in}\mathbf{OUT}\ x.\ g\ (x,\ y))$
and *d-IN-alt-def*: $(\bigwedge x.\ (x,\ y) \notin \mathbf{E} \Longrightarrow g\ (x,\ y) = 0) \Longrightarrow$ *d-IN g y* $= (\sum^+ x{\in}\mathbf{IN}\ y.\ g\ (x,\ y))$
⟨*proof*⟩

lemma *d-OUT-alt-def2*: *d-OUT g x* $= (\sum^+ y{\in}\{y.\ (x,\ y) \in$ *support-flow g*$\}.\ g\ (x,\ y))$
and *d-IN-alt-def2*: *d-IN g y* $= (\sum^+ x{\in}\{x.\ (x,\ y) \in$ *support-flow g*$\}.\ g\ (x,\ y))$
⟨*proof*⟩

definition *d-diff* :: $('v$ *edge* $\Rightarrow$ *ennreal*$) \Rightarrow 'v \Rightarrow$ *ennreal*
where *d-diff g x* $=$ *d-OUT g x* $-$ *d-IN g x*

abbreviation *KIR* :: $('v$ *edge* $\Rightarrow$ *ennreal*$) \Rightarrow 'v \Rightarrow$ *bool*
where *KIR f x* $\equiv$ *d-OUT f x* $=$ *d-IN f x*

inductive-set *SINK* :: $('v$ *edge* $\Rightarrow$ *ennreal*$) \Rightarrow 'v$ *set*
for *f*
where *SINK*: *d-OUT f x* $= 0 \Longrightarrow x \in$ *SINK f*

lemma *SINK-mono*:
assumes $\bigwedge e.\ f\ e \leq g\ e$
shows *SINK g* $\subseteq$ *SINK f*
⟨*proof*⟩

lemma *SINK-mono'*: $f \leq g \Longrightarrow$ *SINK g* $\subseteq$ *SINK f*
⟨*proof*⟩

lemma *support-flow-Sup*: *support-flow* $(Sup\ Y) = (\bigcup f{\in}Y.$ *support-flow f*$)$
⟨*proof*⟩

lemma
assumes *chain*: *Complete-Partial-Order.chain* $(\leq)\ Y$

and Y: $Y \neq \{\}$
and *countable*: *countable* (*support-flow* (*Sup* Y))
shows *d-OUT-Sup*: *d-OUT* (*Sup* Y) x = (*SUP* $f \in Y$. *d-OUT* f x) (**is** *?OUT* x **is** *?lhs1* x = *?rhs1* x)
and *d-IN-Sup*: *d-IN* (*Sup* Y) y = (*SUP* $f \in Y$. *d-IN* f y) (**is** *?IN* **is** *?lhs2* = *?rhs2*)
and *SINK-Sup*: *SINK* (*Sup* Y) = ($\bigcap f \in Y$. *SINK* f) (**is** *?SINK*)
⟨*proof*⟩

lemma
assumes *chain*: *Complete-Partial-Order.chain* ($\leq$) Y
and Y: $Y \neq \{\}$
and *countable*: *countable* (*support-flow* f)
and *bounded*: $\bigwedge g\ e.\ g \in Y \Longrightarrow g\ e \leq f\ e$
shows *d-OUT-Inf*: *d-OUT* f $x \neq$ *top* $\Longrightarrow$ *d-OUT* (*Inf* Y) x = (*INF* $g \in Y$. *d-OUT* g x) (**is** - $\Longrightarrow$ *?OUT* **is** - $\Longrightarrow$ *?lhs1* = *?rhs1*)
and *d-IN-Inf*: *d-IN* f $x \neq$ *top* $\Longrightarrow$ *d-IN* (*Inf* Y) x = (*INF* $g \in Y$. *d-IN* g x) (**is** - $\Longrightarrow$ *?IN* **is** - $\Longrightarrow$ *?lhs2* = *?rhs2*)
⟨*proof*⟩

inductive *flow* :: ('*v*, '*more*) *network-scheme* $\Rightarrow$ '*v flow* $\Rightarrow$ *bool*
for Δ (**structure**) **and** f
where
flow: $[\![\bigwedge e.\ f\ e \leq$ *capacity* $\Delta\ e$;
$\bigwedge x.\ [\![x \neq$ *source* Δ; $x \neq$ *sink* $\Delta\ ]\!] \Longrightarrow$ *KIR* $f\ x\ ]\!]$
$\Longrightarrow$ *flow* $\Delta\ f$

lemma *flowD-capacity*: *flow* $\Delta\ f \Longrightarrow f\ e \leq$ *capacity* $\Delta\ e$
⟨*proof*⟩

lemma *flowD-KIR*: $[\![$ *flow* $\Delta\ f$; $x \neq$ *source* Δ; $x \neq$ *sink* $\Delta\ ]\!] \Longrightarrow$ *KIR* $f\ x$
⟨*proof*⟩

lemma *flowD-capacity-OUT*: *flow* $\Delta\ f \Longrightarrow$ *d-OUT* $f\ x \leq$ *d-OUT* (*capacity* Δ) x
⟨*proof*⟩

lemma *flowD-capacity-IN*: *flow* $\Delta\ f \Longrightarrow$ *d-IN* $f\ x \leq$ *d-IN* (*capacity* Δ) x
⟨*proof*⟩

abbreviation *value-flow* :: ('*v*, '*more*) *network-scheme* $\Rightarrow$ ('*v edge* $\Rightarrow$ *ennreal*) $\Rightarrow$ *ennreal*
where *value-flow* $\Delta\ f \equiv$ *d-OUT* f (*source* Δ)

5.1 Cut

type-synonym '*v cut* = '*v set*

inductive *cut* :: ('*v*, '*more*) *network-scheme* $\Rightarrow$ '*v cut* $\Rightarrow$ *bool*
for Δ **and** S
where *cut*: $[\![$ *source* $\Delta \in S$; *sink* $\Delta \notin S\ ]\!] \Longrightarrow$ *cut* $\Delta\ S$

inductive *orthogonal* :: (*'v*, *'more*) *network-scheme* ⇒ *'v flow* ⇒ *'v cut* ⇒ *bool*
  for Δ *f S*
where
  ⟦ ⋀*x y*. ⟦ *edge* Δ *x y*; *x* ∈ *S*; *y* ∉ *S* ⟧ ⟹ *f* (*x*, *y*) = *capacity* Δ (*x*, *y*);
    ⋀*x y*. ⟦ *edge* Δ *x y*; *x* ∉ *S*; *y* ∈ *S* ⟧ ⟹ *f* (*x*, *y*) = *0* ⟧
  ⟹ *orthogonal* Δ *f S*

lemma *orthogonalD-out*:
  ⟦ *orthogonal* Δ *f S*; *edge* Δ *x y*; *x* ∈ *S*; *y* ∉ *S* ⟧ ⟹ *f* (*x*, *y*) = *capacity* Δ (*x*, *y*)
⟨*proof*⟩

lemma *orthogonalD-in*:
  ⟦ *orthogonal* Δ *f S*; *edge* Δ *x y*; *x* ∉ *S*; *y* ∈ *S* ⟧ ⟹ *f* (*x*, *y*) = *0*
⟨*proof*⟩

5.2 Countable network

locale *countable-network* =
  fixes Δ :: (*'v*, *'more*) *network-scheme* (**structure**)
  assumes *countable-E* [*simp*]: *countable* **E**
  and *source-neq-sink* [*simp*]: *source* Δ ≠ *sink* Δ
  and *capacity-outside*: *e* ∉ **E** ⟹ *capacity* Δ *e* = *0*
  and *capacity-finite* [*simp*]: *capacity* Δ *e* ≠ ⊤
begin

lemma *sink-neq-source* [*simp*]: *sink* Δ ≠ *source* Δ
⟨*proof*⟩

lemma *countable-V* [*simp*]: *countable* **V**
⟨*proof*⟩

lemma *flowD-outside*:
  assumes *g*: *flow* Δ *g*
  shows *e* ∉ **E** ⟹ *g e* = *0*
⟨*proof*⟩

lemma *flowD-finite*:
  assumes *flow* Δ *g*
  shows *g e* ≠ ⊤
⟨*proof*⟩

lemma *zero-flow* [*simp*]: *flow* Δ (λ-. *0*)
⟨*proof*⟩

end

5.3 Reduction for avoiding antiparallel edges

locale *antiparallel-edges* = *countable-network* Δ

for Δ :: ('*v*, '*more*) *network-scheme* (**structure**)
begin

We eliminate the assumption of antiparallel edges by adding a vertex for every edge. Thus, antiparallel edges are split up into a cycle of 4 edges. This idea already appears in [1].

datatype (*plugins del*: *transfer size*) '*v*' *vertex* = *Vertex* '*v*' | *Edge* '*v*' '*v*'

inductive *edg* :: '*v vertex* ⇒ '*v vertex* ⇒ *bool*
where
  OUT: *edge* Δ *x y* ⟹ *edg* (*Vertex x*) (*Edge x y*)
| *IN*: *edge* Δ *x y* ⟹ *edg* (*Edge x y*) (*Vertex y*)

inductive-simps *edg-simps* [*simp*]:
  edg (*Vertex x*) *v*
  edg (*Edge x y*) *v*
  edg v (*Vertex x*)
  edg v (*Edge x y*)

fun *split* :: '*v flow* ⇒ '*v vertex flow*
where
  split f (*Vertex x*, *Edge x*' *y*) = (*if x*' = *x then f* (*x*, *y*) *else 0*)
| *split f* (*Edge x y*', *Vertex y*) = (*if y*' = *y then f* (*x*, *y*) *else 0*)
| *split f* - = *0*

lemma *split-Vertex1-eq-0I*: (⋀*z*. *y* ≠ *Edge x z*) ⟹ *split f* (*Vertex x*, *y*) = *0*
⟨*proof*⟩

lemma *split-Vertex2-eq-0I*: (⋀*z*. *y* ≠ *Edge z x*) ⟹ *split f* (*y*, *Vertex x*) = *0*
⟨*proof*⟩

lemma *split-Edge1-eq-0I*: (⋀*z*. *y* ≠ *Vertex x*) ⟹ *split f* (*Edge z x*, *y*) = *0*
⟨*proof*⟩

lemma *split-Edge2-eq-0I*: (⋀*z*. *y* ≠ *Vertex x*) ⟹ *split f* (*y*, *Edge x z*) = *0*
⟨*proof*⟩

definition Δ'' :: '*v vertex network*
where Δ'' = ⦇*edge* = *edg*, *capacity* = *split* (*capacity* Δ), *source* = *Vertex* (*source* Δ), *sink* = *Vertex* (*sink* Δ)⦈

lemma Δ''-*sel* [*simp*]:
  edge Δ'' = *edg*
  capacity Δ'' = *split* (*capacity* Δ)
  source Δ'' = *Vertex* (*source* Δ)
  sink Δ'' = *Vertex* (*sink* Δ)
⟨*proof*⟩

lemma **E**-Δ'': $\mathbf{E}_{\Delta''}$ = (λ(*x*, *y*). (*Vertex x*, *Edge x y*)) ‘ **E** ∪ (λ(*x*, *y*). (*Edge x y*,

Vertex y)) ‘ **E**
⟨*proof*⟩

lemma **V**-Δ′′: $\mathbf{V}_{\Delta''}$ = *Vertex* ‘ **V** ∪ *case-prod Edge* ‘ **E**
⟨*proof*⟩

lemma *inj-on-Edge1* [*simp*]: *inj-on* (λ*x*. *Edge x y*) *A*
⟨*proof*⟩

lemma *inj-on-Edge2* [*simp*]: *inj-on* (*Edge x*) *A*
⟨*proof*⟩

lemma *d-IN-split-Vertex* [*simp*]: *d-IN* (*split f*) (*Vertex x*) = *d-IN f x* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *d-OUT-split-Vertex* [*simp*]: *d-OUT* (*split f*) (*Vertex x*) = *d-OUT f x* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *d-IN-split-Edge* [*simp*]: *d-IN* (*split f*) (*Edge x y*) = *max 0* (*f* (*x*, *y*)) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *d-OUT-split-Edge* [*simp*]: *d-OUT* (*split f*) (*Edge x y*) = *max 0* (*f* (*x*, *y*)) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma Δ′′-*countable-network*: *countable-network* Δ′′
⟨*proof*⟩

interpretation Δ′′: *countable-network* Δ′′ ⟨*proof*⟩

lemma *flow-split* [*simp*]:
  assumes *flow* Δ *f*
  shows *flow* Δ′′ (*split f*)
⟨*proof*⟩

abbreviation (*input*) *collect* :: *'v vertex flow* ⇒ *'v flow*
where *collect f* ≡ (λ(*x*, *y*). *f* (*Edge x y*, *Vertex y*))

lemma *d-OUT-collect*:
  assumes *f*: *flow* Δ′′ *f*
  shows *d-OUT* (*collect f*) *x* = *d-OUT f* (*Vertex x*)
⟨*proof*⟩

lemma *flow-collect* [*simp*]:
  assumes *f*: *flow* Δ′′ *f*
  shows *flow* Δ (*collect f*)

⟨*proof*⟩

lemma *value-collect*: *flow* Δ'' $f \Longrightarrow$ *value-flow* Δ (*collect* f) = *value-flow* Δ'' f
⟨*proof*⟩

end

end
theory *MFMC-Web* **imports**
MFMC-Network
begin

6 Webs and currents

record $'v$ *web* = $'v$ *graph* +
weight :: $'v \Rightarrow$ *ennreal*
A :: $'v$ *set*
B :: $'v$ *set*

lemma *vertex-weight-update* [*simp*]: *vertex* (*weight-update* f Γ) = *vertex* Γ
⟨*proof*⟩

type-synonym $'v$ *current* = $'v$ *edge* $\Rightarrow$ *ennreal*

inductive *current* :: ($'v$, $'more$) *web-scheme* $\Rightarrow$ $'v$ *current* $\Rightarrow$ *bool*
for Γ f
where
current:
⟦ $\bigwedge x$. *d-OUT* f $x \leq$ *weight* Γ x;
$\bigwedge x$. *d-IN* f $x \leq$ *weight* Γ x;
$\bigwedge x$. $x \notin A$ $\Gamma \Longrightarrow$ *d-OUT* f $x \leq$ *d-IN* f x;
$\bigwedge a$. $a \in A$ $\Gamma \Longrightarrow$ *d-IN* f $a = 0$;
$\bigwedge b$. $b \in B$ $\Gamma \Longrightarrow$ *d-OUT* f $b = 0$;
$\bigwedge e$. $e \notin \mathbf{E}_\Gamma \Longrightarrow f$ $e = 0$ ⟧
$\Longrightarrow$ *current* Γ f

lemma *currentD-weight-OUT*: *current* Γ $f \Longrightarrow$ *d-OUT* f $x \leq$ *weight* Γ x
⟨*proof*⟩

lemma *currentD-weight-IN*: *current* Γ $f \Longrightarrow$ *d-IN* f $x \leq$ *weight* Γ x
⟨*proof*⟩

lemma *currentD-OUT-IN*: ⟦ *current* Γ f; $x \notin A$ Γ ⟧ $\Longrightarrow$ *d-OUT* f $x \leq$ *d-IN* f x
⟨*proof*⟩

lemma *currentD-IN*: ⟦ *current* Γ f; $a \in A$ Γ ⟧ $\Longrightarrow$ *d-IN* f $a = 0$
⟨*proof*⟩

lemma *currentD-OUT*: ⟦ *current* Γ f; $b \in B$ Γ ⟧ $\Longrightarrow$ *d-OUT* f $b = 0$

⟨*proof*⟩

lemma *currentD-outside*: ⟦ *current* Γ *f*; ¬ *edge* Γ *x y* ⟧ ⟹ *f* (*x*, *y*) = *0*
⟨*proof*⟩

lemma *currentD-outside'*: ⟦ *current* Γ *f*; *e* ∉ $\mathbf{E}_\Gamma$ ⟧ ⟹ *f e* = *0*
⟨*proof*⟩

lemma *currentD-OUT-eq-0*:
assumes *current* Γ *f*
shows *d-OUT f x* = *0* ⟷ (∀ *y*. *f* (*x*, *y*) = *0*)
⟨*proof*⟩

lemma *currentD-IN-eq-0*:
assumes *current* Γ *f*
shows *d-IN f x* = *0* ⟷ (∀ *y*. *f* (*y*, *x*) = *0*)
⟨*proof*⟩

lemma *current-support-flow*:
fixes Γ (**structure**)
assumes *current* Γ *f*
shows *support-flow f* ⊆ **E**
⟨*proof*⟩

lemma *currentD-outside-IN*: ⟦ *current* Γ *f*; *x* ∉ $\mathbf{V}_\Gamma$ ⟧ ⟹ *d-IN f x* = *0*
⟨*proof*⟩

lemma *currentD-outside-OUT*: ⟦ *current* Γ *f*; *x* ∉ $\mathbf{V}_\Gamma$ ⟧ ⟹ *d-OUT f x* = *0*
⟨*proof*⟩

lemma *currentD-weight-in*: *current* Γ *h* ⟹ *h* (*x*, *y*) ≤ *weight* Γ *y*
⟨*proof*⟩

lemma *currentD-weight-out*: *current* Γ *h* ⟹ *h* (*x*, *y*) ≤ *weight* Γ *x*
⟨*proof*⟩

lemma *current-leI*:
fixes Γ (**structure**)
assumes *f*: *current* Γ *f*
and *le*: ⋀*e*. *g e* ≤ *f e*
and *OUT-IN*: ⋀*x*. *x* ∉ *A* Γ ⟹ *d-OUT g x* ≤ *d-IN g x*
shows *current* Γ *g*
⟨*proof*⟩

lemma *current-weight-mono*:
⟦ *current* Γ *f*; *edge* Γ = *edge* Γ′; *A* Γ = *A* Γ′; *B* Γ = *B* Γ′; ⋀*x*. *weight* Γ *x* ≤ *weight* Γ′ *x* ⟧
⟹ *current* Γ′ *f*
⟨*proof*⟩

abbreviation (*input*) *zero-current* :: *'v current*
where *zero-current* ≡ *λ-. 0*

lemma *SINK-0* [*simp*]: *SINK zero-current = UNIV*
⟨*proof*⟩

lemma *current-0* [*simp*]: *current Γ zero-current*
⟨*proof*⟩

inductive *web-flow* :: (*'v*, *'more*) *web-scheme* ⇒ *'v current* ⇒ *bool*
  for *Γ* (**structure**) **and** *f*
where
  web-flow: ⟦ *current Γ f*; ⋀*x*. ⟦ *x* ∈ **V**; *x* ∉ *A Γ*; *x* ∉ *B Γ* ⟧ ⟹ *KIR f x* ⟧ ⟹ *web-flow Γ f*

lemma *web-flowD-current*: *web-flow Γ f* ⟹ *current Γ f*
⟨*proof*⟩

lemma *web-flowD-KIR*: ⟦ *web-flow Γ f*; *x* ∉ *A Γ*; *x* ∉ *B Γ* ⟧ ⟹ *KIR f x*
⟨*proof*⟩

6.1 Saturated and terminal vertices

inductive-set *SAT* :: (*'v*, *'more*) *web-scheme* ⇒ *'v current* ⇒ *'v set*
  for *Γ f*
where
  A: *x* ∈ *A Γ* ⟹ *x* ∈ *SAT Γ f*
| *IN*: *d-IN f x* ≥ *weight Γ x* ⟹ *x* ∈ *SAT Γ f*
  — We use ≥ *weight* such that *SAT* is monotone w.r.t. increasing currents

lemma *SAT-0* [*simp*]: *SAT Γ zero-current = A Γ* ∪ {*x*. *weight Γ x* ≤ *0*}
⟨*proof*⟩

lemma *SAT-mono*:
  assumes ⋀*e*. *f e* ≤ *g e*
  shows *SAT Γ f* ⊆ *SAT Γ g*
⟨*proof*⟩

lemma *SAT-Sup-upper*: *f* ∈ *Y* ⟹ *SAT Γ f* ⊆ *SAT Γ* (*Sup Y*)
⟨*proof*⟩

lemma *currentD-SAT*:
  assumes *current Γ f*
  shows *x* ∈ *SAT Γ f* ⟷ *x* ∈ *A Γ* ∨ *d-IN f x = weight Γ x*
⟨*proof*⟩

abbreviation *terminal* :: (*'v*, *'more*) *web-scheme* ⇒ *'v current* ⇒ *'v set* (*TER*1)
where *terminal Γ f* ≡ *SAT Γ f* ∩ *SINK f*

6.2 Separation

inductive *separating-gen* :: ('*v*, '*more*) *graph-scheme* ⇒ '*v set* ⇒ '*v set* ⇒ '*v set* ⇒ *bool*
  for *G A B S*
where *separating*:
  (⋀*x y p*. ⟦ *x* ∈ *A*; *y* ∈ *B*; *path G x p y* ⟧ ⟹ (∃ *z* ∈ *set p*. *z* ∈ *S*) ∨ *x* ∈ *S*)
  ⟹ *separating-gen G A B S*

abbreviation *separating* :: ('*v*, '*more*) *web-scheme* ⇒ '*v set* ⇒ *bool*
where *separating* Γ ≡ *separating-gen* Γ (*A* Γ) (*B* Γ)

abbreviation *separating-network* :: ('*v*, '*more*) *network-scheme* ⇒ '*v set* ⇒ *bool*
where *separating-network* Δ ≡ *separating-gen* Δ {*source* Δ} {*sink* Δ}

lemma *separating-networkI* [*intro?*]:
  (⋀*p*. *path* Δ (*source* Δ) *p* (*sink* Δ) ⟹ (∃ *z* ∈ *set p*. *z* ∈ *S*) ∨ *source* Δ ∈ *S*)
  ⟹ *separating-network* Δ *S*
⟨*proof*⟩

lemma *separatingD*:
  ⋀*A B*. ⟦ *separating-gen G A B S*; *path G x p y*; *x* ∈ *A*; *y* ∈ *B* ⟧ ⟹ (∃ *z* ∈ *set p*. *z* ∈ *S*) ∨ *x* ∈ *S*
⟨*proof*⟩

lemma *separating-left* [*simp*]: ⋀*A B*. *A* ⊆ *A*′ ⟹ *separating-gen* Γ *A B A*′
⟨*proof*⟩

lemma *separating-weakening*:
  ⋀*A B*. ⟦ *separating-gen G A B S*; *S* ⊆ *S*′ ⟧ ⟹ *separating-gen G A B S*′
⟨*proof*⟩

definition *essential* :: ('*v*, '*more*) *graph-scheme* ⇒ '*v set* ⇒ '*v set* ⇒ '*v* ⇒ *bool*
where — Should we allow only simple paths here?
  ⋀*B*. *essential G B S x* ⟷ (∃ *p*. ∃ *y*∈*B*. *path G x p y* ∧ (*x* ≠ *y* ⟶ (∀ *z*∈*set p*. *z* = *x* ∨ *z* ∉ *S*)))

abbreviation *essential-web* :: ('*v*, '*more*) *web-scheme* ⇒ '*v set* ⇒ '*v set* (ℰ1)
where *essential-web* Γ *S* ≡ {*x*∈*S*. *essential* Γ (*B* Γ) *S x*}

lemma *essential-weight-update* [*simp*]:
  essential (*weight-update f G*) = *essential G*
⟨*proof*⟩

lemma *not-essentialD*:
  ⋀*B*. ⟦ ¬ *essential G B S x*; *path G x p y*; *y* ∈ *B* ⟧ ⟹ *x* ≠ *y* ∧ (∃ *z*∈*set p*. *z* ≠ *x* ∧ *z* ∈ *S*)
⟨*proof*⟩

lemma *essentialE* [*elim?*, *consumes 1*, *case-names essential*, *cases pred*: *essential*]:

$\bigwedge B.$ ⟦ *essential G B S x*; $\bigwedge p\ y.$ ⟦ *path G x p y*; *y* ∈ *B*; $\bigwedge z.$ ⟦ *x* ≠ *y*; *z* ∈ *set p* ⟧ ⟹ *z* = *x* ∨ *z* ∉ *S* ⟧ ⟹ *thesis* ⟧ ⟹ *thesis*
⟨*proof*⟩

lemma *essentialI* [*intro?*]:
$\bigwedge B.$ ⟦ *path G x p y*; *y* ∈ *B*; $\bigwedge z.$ ⟦ *x* ≠ *y*; *z* ∈ *set p* ⟧ ⟹ *z* = *x* ∨ *z* ∉ *S* ⟧ ⟹ *essential G B S x*
⟨*proof*⟩

lemma *essential-vertex*: $\bigwedge B.$ ⟦ *essential G B S x*; *x* ∉ *B* ⟧ ⟹*vertex G x*
⟨*proof*⟩

lemma *essential-BI*: $\bigwedge B.$ *x* ∈ *B* ⟹ *essential G B S x*
⟨*proof*⟩

lemma $\mathcal{E}$*-E* [*elim?*, *consumes 1*, *case-names* $\mathcal{E}$, *cases set*: *essential-web*]:
fixes Γ (**structure**)
assumes *x* ∈ $\mathcal{E}$ *S*
obtains *p y* **where** *path* Γ *x p y y* ∈ *B* Γ $\bigwedge z.$ ⟦ *x* ≠ *y*; *z* ∈ *set p* ⟧ ⟹ *z* = *x* ∨ *z* ∉ *S*
⟨*proof*⟩

lemma *essential-mono*: $\bigwedge B.$ ⟦ *essential G B S x*; *S′* ⊆ *S* ⟧ ⟹ *essential G B S′ x*
⟨*proof*⟩

lemma *separating-essential*: — Lem. 3.4 (cf. Lem. 2.14 in [5])
fixes *G A B S*
assumes *separating-gen G A B S*
shows *separating-gen G A B* {*x*∈*S*. *essential G B S x*} (**is** *separating-gen - - - ?E*)
⟨*proof*⟩

definition *roofed-gen* :: (*'v*, *'more*) *graph-scheme* ⇒ *'v set* ⇒ *'v set* ⇒ *'v set*
where *roofed-def*: $\bigwedge B.$ *roofed-gen G B S* = {*x*. ∀ *p*. ∀ *y*∈*B*. *path G x p y* ⟶ (∃ *z*∈*set p*. *z* ∈ *S*) ∨ *x* ∈ *S*}

abbreviation *roofed* :: (*'v*, *'more*) *web-scheme* ⇒ *'v set* ⇒ *'v set* (*RF*₁)
where *roofed* Γ ≡ *roofed-gen* Γ (*B* Γ)

abbreviation *roofed-network* :: (*'v*, *'more*) *network-scheme* ⇒ *'v set* ⇒ *'v set* (RF^N_1)
where *roofed-network* Δ ≡ *roofed-gen* Δ {*sink* Δ}

lemma *roofedI* [*intro?*]:
$\bigwedge B.$ (⋀*p y.* ⟦ *path G x p y*; *y* ∈ *B* ⟧ ⟹ (∃ *z*∈*set p*. *z* ∈ *S*) ∨ *x* ∈ *S*) ⟹ *x* ∈ *roofed-gen G B S*
⟨*proof*⟩

lemma *not-roofedE*: **fixes** *B*

assumes $x \notin$ *roofed-gen G B S*
obtains *p y* **where** *path G x p y y* $\in B$ $\bigwedge z.\ z \in$ *set* $(x \# p) \Longrightarrow z \notin S$
⟨*proof*⟩

lemma *roofed-greater*: $\bigwedge B.\ S \subseteq$ *roofed-gen G B S*
⟨*proof*⟩

lemma *roofed-greaterI*: $\bigwedge B.\ x \in S \Longrightarrow x \in$ *roofed-gen G B S*
⟨*proof*⟩

lemma *roofed-mono*: $\bigwedge B.\ S \subseteq S' \Longrightarrow$ *roofed-gen G B S* $\subseteq$ *roofed-gen G B S′*
⟨*proof*⟩

lemma *in-roofed-mono*: $\bigwedge B.\ [\![\ x \in$ *roofed-gen G B S*; $S \subseteq S'\ ]\!] \Longrightarrow x \in$ *roofed-gen G B S′*
⟨*proof*⟩

lemma *roofedD*: $\bigwedge B.\ [\![\ x \in$ *roofed-gen G B S*; *path G x p y*; $y \in B\ ]\!] \Longrightarrow (\exists z{\in}$*set p*. $z \in S) \vee x \in S$
⟨*proof*⟩

lemma *separating-RF-A*:
fixes *A B*
assumes *separating-gen G A B X*
shows $A \subseteq$ *roofed-gen G B X*
⟨*proof*⟩

lemma *roofed-idem*: **fixes** *B* **shows** *roofed-gen G B* (*roofed-gen G B S*) = *roofed-gen G B S*
⟨*proof*⟩

lemma *in-roofed-mono′*: $\bigwedge B.\ [\![\ x \in$ *roofed-gen G B S*; $S \subseteq$ *roofed-gen G B S′* $]\!]$ $\Longrightarrow x \in$ *roofed-gen G B S′*
⟨*proof*⟩

lemma *roofed-mono′*: $\bigwedge B.\ S \subseteq$ *roofed-gen G B S′* $\Longrightarrow$ *roofed-gen G B S* $\subseteq$ *roofed-gen G B S′*
⟨*proof*⟩

lemma *roofed-idem-Un1*: **fixes** *B* **shows** *roofed-gen G B* (*roofed-gen G B S* $\cup$ *T*) = *roofed-gen G B* (*S* $\cup$ *T*)
⟨*proof*⟩

lemma *roofed-UN*: **fixes** *A B*
shows *roofed-gen G B* $(\bigcup i{\in}A.$ *roofed-gen G B* (*X i*)) = *roofed-gen G B* $(\bigcup i{\in}A.$ *X i*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *RF-essential*: **fixes** Γ (**structure**) **shows** *RF* $(\mathcal{E}\ S)$ = *RF S*

⟨*proof*⟩

lemma *essentialE-RF*:
fixes Γ (**structure**) **and** *B*
assumes *essential* Γ *B S x*
obtains *p y* **where** *path* Γ *x p y y* ∈ *B distinct* (*x* # *p*) ⋀*z*. *z* ∈ *set p* ⟹ *z* ∉ *roofed-gen* Γ *B S*
⟨*proof*⟩

lemma $\mathcal{E}$*-E-RF*:
fixes Γ (**structure**)
assumes *x* ∈ $\mathcal{E}$ *S*
obtains *p y* **where** *path* Γ *x p y y* ∈ *B* Γ *distinct* (*x* # *p*) ⋀*z*. *z* ∈ *set p* ⟹ *z* ∉ *RF S*
⟨*proof*⟩

lemma *in-roofed-essentialD*:
fixes Γ (**structure**)
assumes *RF*: *x* ∈ *RF S*
and *ess*: *essential* Γ (*B* Γ) *S x*
shows *x* ∈ *S*
⟨*proof*⟩

lemma *separating-RF*: **fixes** Γ (**structure**) **shows** *separating* Γ (*RF S*) ⟷ *separating* Γ *S*
⟨*proof*⟩

definition *roofed-circ* :: ('*v*, '*more*) *web-scheme* ⇒ '*v set* ⇒ '*v set* (*RF*°₁)
where *roofed-circ* Γ *S* = *roofed* Γ *S* − $\mathcal{E}_\Gamma$ *S*

lemma *roofed-circI*: **fixes** Γ (**structure**) **shows**
⟦ *x* ∈ *RF T*; *x* ∈ *T* ⟹ ¬ *essential* Γ (*B* Γ) *T x* ⟧ ⟹ *x* ∈ *RF*° *T*
⟨*proof*⟩

lemma *roofed-circE*:
fixes Γ (**structure**)
assumes *x* ∈ *RF*° *T*
obtains *x* ∈ *RF T* ¬ *essential* Γ (*B* Γ) *T x*
⟨*proof*⟩

lemma $\mathcal{E}$*-*$\mathcal{E}$: **fixes** Γ (**structure**) **shows** $\mathcal{E}$ ($\mathcal{E}$ *S*) = $\mathcal{E}$ *S*
⟨*proof*⟩

lemma *roofed-circ-essential*: **fixes** Γ (**structure**) **shows** *RF*° ($\mathcal{E}$ *S*) = *RF*° *S*
⟨*proof*⟩

lemma *essential-RF*: **fixes** *B*
shows *essential G B* (*roofed-gen G B S*) = *essential G B S* (**is** *essential - - ?RF* = -)

⟨*proof*⟩

lemma $\mathcal{E}$*-RF*: **fixes** Γ (**structure**) **shows** $\mathcal{E}\ (RF\ S) = \mathcal{E}\ S$
⟨*proof*⟩

lemma *essential-*$\mathcal{E}$: **fixes** Γ (**structure**) **shows** *essential* $\Gamma\ (B\ \Gamma)\ (\mathcal{E}\ S) =$ *essential* $\Gamma\ (B\ \Gamma)\ S$
⟨*proof*⟩

lemma *RF-in-B*: **fixes** Γ (**structure**) **shows** $x \in B\ \Gamma \Longrightarrow x \in RF\ S \longleftrightarrow x \in S$
⟨*proof*⟩

lemma *RF-circ-edge-forward*:
fixes Γ (**structure**)
assumes *x*: $x \in RF^{\circ}\ S$
and *edge*: *edge* $\Gamma\ x\ y$
shows $y \in RF\ S$
⟨*proof*⟩

6.3 Waves

inductive *wave* :: ('*v*, '*more*) *web-scheme* ⇒ '*v current* ⇒ *bool*
for Γ (**structure**) **and** *f*
where
wave:
⟦ *separating* $\Gamma\ (TER\ f)$;
$\bigwedge x.\ x \notin RF\ (TER\ f) \Longrightarrow$ *d-OUT f x* = *0* ⟧
$\Longrightarrow$ *wave* $\Gamma\ f$

lemma *wave-0* [*simp*]: *wave* Γ *zero-current*
⟨*proof*⟩

lemma *waveD-separating*: *wave* $\Gamma\ f \Longrightarrow$ *separating* $\Gamma\ (TER_{\Gamma}\ f)$
⟨*proof*⟩

lemma *waveD-OUT*: ⟦ *wave* $\Gamma\ f$; $x \notin RF_{\Gamma}\ (TER_{\Gamma}\ f)$ ⟧ $\Longrightarrow$ *d-OUT f x* = *0*
⟨*proof*⟩

lemma *wave-A-in-RF*: **fixes** Γ (**structure**)
shows ⟦ *wave* $\Gamma\ f$; $x \in A\ \Gamma$ ⟧ $\Longrightarrow x \in RF\ (TER\ f)$
⟨*proof*⟩

lemma *wave-not-RF-IN-zero*:
fixes Γ (**structure**)
assumes *f*: *current* $\Gamma\ f$
and *w*: *wave* $\Gamma\ f$
and *x*: $x \notin RF\ (TER\ f)$
shows *d-IN f x* = *0*
⟨*proof*⟩

lemma *current-Sup*:
fixes Γ (**structure**)
assumes *chain*: *Complete-Partial-Order.chain* $(\leq)$ Y
and Y: $Y \neq \{\}$
and *current*: $\bigwedge f.\ f \in Y \Longrightarrow$ *current* Γ f
and *countable* [*simp*]: *countable* (*support-flow* (*Sup* Y))
shows *current* Γ (*Sup* Y)
⟨*proof*⟩

lemma *wave-lub*: — Lemma 4.3
fixes Γ (**structure**)
assumes *chain*: *Complete-Partial-Order.chain* $(\leq)$ Y
and Y: $Y \neq \{\}$
and *wave*: $\bigwedge f.\ f \in Y \Longrightarrow$ *wave* Γ f
and *countable* [*simp*]: *countable* (*support-flow* (*Sup* Y))
shows *wave* Γ (*Sup* Y)
⟨*proof*⟩

lemma *ex-maximal-wave*: — Corollary 4.4
fixes Γ (**structure**)
assumes *countable*: *countable* **E**
shows $\exists f.$ *current* Γ $f \wedge$ *wave* Γ $f \wedge (\forall w.$ *current* Γ $w \wedge$ *wave* Γ $w \wedge f \leq w \longrightarrow f = w)$
⟨*proof*⟩

lemma *essential-leI*:
fixes Γ (**structure**)
assumes g: *current* Γ g **and** w: *wave* Γ g
and *le*: $\bigwedge e.\ f\ e \leq g\ e$
and x: $x \in \mathcal{E}$ (*TER* g)
shows *essential* Γ (*B* Γ) (*TER* f) x
⟨*proof*⟩

lemma *essential-eq-leI*:
fixes Γ (**structure**)
assumes g: *current* Γ g **and** w: *wave* Γ g
and *le*: $\bigwedge e.\ f\ e \leq g\ e$
and *subset*: $\mathcal{E}$ (*TER* g) $\subseteq$ *TER* f
shows $\mathcal{E}$ (*TER* f) $=$ $\mathcal{E}$ (*TER* g)
⟨*proof*⟩

6.4 Hindrances and looseness

inductive *hindrance-by* :: ('*v*, '*more*) *web-scheme* ⇒ '*v current* ⇒ *ennreal* ⇒ *bool*
for Γ (**structure**) **and** f **and** ε
where
hindrance-by:
⟦ $a \in A\ \Gamma$; $a \notin \mathcal{E}$ (*TER* f); *d-OUT* $f\ a <$ *weight* Γ a; $\varepsilon <$ *weight* Γ $a -$ *d-OUT* f

$a \rrbracket \Longrightarrow$ *hindrance-by* $\Gamma\ f\ \varepsilon$

inductive *hindrance* :: ('*v*, '*more*) *web-scheme* ⇒ '*v current* ⇒ *bool*
  for Γ (**structure**) **and** f
where
  hindrance:
  $\llbracket\ a \in A\ \Gamma;\ a \notin \mathcal{E}\ (TER\ f);\ d\text{-}OUT\ f\ a < weight\ \Gamma\ a\ \rrbracket \Longrightarrow$ *hindrance* $\Gamma\ f$

inductive *hindered* :: ('*v*, '*more*) *web-scheme* ⇒ *bool*
  for Γ (**structure**)
where *hindered*: $\llbracket$ *hindrance* $\Gamma\ f$; *current* $\Gamma\ f$; *wave* $\Gamma\ f\ \rrbracket \Longrightarrow$ *hindered* Γ

inductive *hindered-by* :: ('*v*, '*more*) *web-scheme* ⇒ *ennreal* ⇒ *bool*
  for Γ (**structure**) **and** ε
where *hindered-by*: $\llbracket$ *hindrance-by* $\Gamma\ f\ \varepsilon$; *current* $\Gamma\ f$; *wave* $\Gamma\ f\ \rrbracket \Longrightarrow$ *hindered-by* $\Gamma\ \varepsilon$

lemma *hindrance-into-hindrance-by*:
  assumes *hindrance* $\Gamma\ f$
  shows $\exists \varepsilon{>}0.$ *hindrance-by* $\Gamma\ f\ \varepsilon$
⟨*proof*⟩

lemma *hindrance-by-into-hindrance*: *hindrance-by* $\Gamma\ f\ \varepsilon \Longrightarrow$ *hindrance* $\Gamma\ f$
⟨*proof*⟩

lemma *hindrance-conv-hindrance-by*: *hindrance* $\Gamma\ f \longleftrightarrow (\exists \varepsilon{>}0.$ *hindrance-by* $\Gamma\ f\ \varepsilon)$
⟨*proof*⟩

lemma *hindered-into-hindered-by*: *hindered* $\Gamma \Longrightarrow \exists \varepsilon{>}0.$ *hindered-by* $\Gamma\ \varepsilon$
⟨*proof*⟩

lemma *hindered-by-into-hindered*: *hindered-by* $\Gamma\ \varepsilon \Longrightarrow$ *hindered* Γ
⟨*proof*⟩

lemma *hindered-conv-hindered-by*: *hindered* $\Gamma \longleftrightarrow (\exists \varepsilon{>}0.$ *hindered-by* $\Gamma\ \varepsilon)$
⟨*proof*⟩

inductive *loose* :: ('*v*, '*more*) *web-scheme* ⇒ *bool*
  for Γ
where
  loose: $\llbracket \bigwedge f.\ \llbracket$ *current* $\Gamma\ f$; *wave* $\Gamma\ f\ \rrbracket \Longrightarrow f =$ *zero-current*; $\neg$ *hindrance* Γ *zero-current* $\rrbracket$
  $\Longrightarrow$ *loose* Γ

lemma *looseD-hindrance*: *loose* $\Gamma \Longrightarrow \neg$ *hindrance* Γ *zero-current*
⟨*proof*⟩

lemma *looseD-wave*:

⟦ *loose* Γ; *current* Γ *f*; *wave* Γ *f* ⟧ ⟹ *f* = *zero-current*
⟨*proof*⟩

lemma *loose-unhindered*:
fixes Γ (**structure**)
assumes *loose* Γ
shows ¬ *hindered* Γ
⟨*proof*⟩

context
fixes Γ Γ′ :: ('*v*, '*more*) *web-scheme*
assumes [*simp*]: *edge* Γ = *edge* Γ′ *A* Γ = *A* Γ′ *B* Γ = *B* Γ′
and *weight-eq*: ⋀*x*. *x* ∉ *A* Γ′ ⟹ *weight* Γ *x* = *weight* Γ′ *x*
and *weight-le*: ⋀*a*. *a* ∈ *A* Γ′ ⟹ *weight* Γ *a* ≥ *weight* Γ′ *a*
begin

private lemma *essential-eq*: *essential* Γ = *essential* Γ′
⟨*proof*⟩ **lemma** *TER-eq*: TER_{Γ} *f* = $TER_{\Gamma'}$ *f*
⟨*proof*⟩ **lemma** *separating-eq*: *separating-gen* Γ = *separating-gen* Γ′
⟨*proof*⟩ **lemma** *roofed-eq*: ⋀*B*. *roofed-gen* Γ *B S* = *roofed-gen* Γ′ *B S*
⟨*proof*⟩

lemma *wave-eq-web*: — Observation 4.6
wave Γ *f* ⟷ *wave* Γ′ *f*
⟨*proof*⟩

lemma *current-mono-web*: *current* Γ′ *f* ⟹ *current* Γ *f*
⟨*proof*⟩

lemma *hindrance-mono-web*: *hindrance* Γ′ *f* ⟹ *hindrance* Γ *f*
⟨*proof*⟩

lemma *hindered-mono-web*: *hindered* Γ′ ⟹ *hindered* Γ
⟨*proof*⟩

end

6.5 Linkage

The following definition of orthogonality is stronger than the original definition 3.5 in [2] in that the outflow from any A-vertices in the set must saturate the vertex; $S \subseteq SAT\ \Gamma\ f$ is not enough.

With the original definition of orthogonal current, the reduction from networks to webs fails because the induced flow need not saturate edges going out of the source. Consider the network with three nodes s, x, and t and edges $(s,\ x)$ and $(x,\ t)$ with capacity *1*. Then, the corresponding web has the vertices $(s,\ x)$ and $(x,\ t)$ and one edge from $(s,\ x)$ to $(x,\ t)$. Clearly, the zero current *zero-current* is a web-flow and *TER zero-current* = {(*s*,

$x)\}$, which is essential. Moreover, *zero-current* and $\{(s,\ x)\}$ are orthogonal because *zero-current* trivially saturates $(s,\ x)$ as this is a vertex in A.

inductive *orthogonal-current* :: $('v,\ 'more)$ *web-scheme* $\Rightarrow$ $'v$ *current* $\Rightarrow$ $'v$ *set* $\Rightarrow$ *bool*
  for Γ (**structure**) **and** $f\ S$
where *orthogonal-current*:
  $[\![\ \bigwedge x.\ [\![\ x \in S;\ x \notin A\ \Gamma\ ]\!] \Longrightarrow \mathit{weight}\ \Gamma\ x \leq \mathit{d\text{-}IN}\ f\ x;$
    $\bigwedge x.\ [\![\ x \in S;\ x \in A\ \Gamma;\ x \notin B\ \Gamma\ ]\!] \Longrightarrow \mathit{d\text{-}OUT}\ f\ x = \mathit{weight}\ \Gamma\ x;$
    $\bigwedge u\ v.\ [\![\ v \in \mathit{RF}\ S;\ u \notin \mathit{RF}^{\circ}\ S\ ]\!] \Longrightarrow f\ (u,\ v) = 0\ ]\!]$
  $\Longrightarrow$ *orthogonal-current* $\Gamma\ f\ S$

lemma *orthogonal-currentD-SAT*: $[\![\ \mathit{orthogonal\text{-}current}\ \Gamma\ f\ S;\ x \in S\ ]\!] \Longrightarrow x \in \mathit{SAT}\ \Gamma\ f$
⟨*proof*⟩

lemma *orthogonal-currentD-A*: $[\![\ \mathit{orthogonal\text{-}current}\ \Gamma\ f\ S;\ x \in S;\ x \in A\ \Gamma;\ x \notin B\ \Gamma\ ]\!] \Longrightarrow \mathit{d\text{-}OUT}\ f\ x = \mathit{weight}\ \Gamma\ x$
⟨*proof*⟩

lemma *orthogonal-currentD-in*: $[\![\ \mathit{orthogonal\text{-}current}\ \Gamma\ f\ S;\ v \in \mathit{RF}_{\Gamma}\ S;\ u \notin \mathit{RF}^{\circ}{}_{\Gamma}\ S\ ]\!] \Longrightarrow f\ (u,\ v) = 0$
⟨*proof*⟩

inductive *linkage* :: $('v,\ 'more)$ *web-scheme* $\Rightarrow$ $'v$ *current* $\Rightarrow$ *bool*
  for $\Gamma\ f$
where — Omit the condition *web-flow*
  linkage: $(\bigwedge x.\ x \in A\ \Gamma \Longrightarrow \mathit{d\text{-}OUT}\ f\ x = \mathit{weight}\ \Gamma\ x) \Longrightarrow \mathit{linkage}\ \Gamma\ f$

lemma *linkageD*: $[\![\ \mathit{linkage}\ \Gamma\ f;\ x \in A\ \Gamma\ ]\!] \Longrightarrow \mathit{d\text{-}OUT}\ f\ x = \mathit{weight}\ \Gamma\ x$
⟨*proof*⟩

abbreviation *linkable* :: $('v,\ 'more)$ *web-scheme* $\Rightarrow$ *bool*
where $\mathit{linkable}\ \Gamma \equiv \exists f.\ \mathit{web\text{-}flow}\ \Gamma\ f \wedge \mathit{linkage}\ \Gamma\ f$

6.6 Trimming

context
  fixes Γ :: $('v,\ 'more)$ *web-scheme* (**structure**)
  and f :: $'v$ *current*
begin

inductive *trimming* :: $'v$ *current* $\Rightarrow$ *bool*
  for g
where
  trimming:
  — omits the condition that f is a wave
  $[\![\ \mathit{current}\ \Gamma\ g;\ \mathit{wave}\ \Gamma\ g;\ g \leq f;\ \bigwedge x.\ [\![\ x \in \mathit{RF}^{\circ}\ (\mathit{TER}\ f);\ x \notin A\ \Gamma\ ]\!] \Longrightarrow \mathit{KIR}\ g\ x;\ \mathcal{E}\ (\mathit{TER}\ g) - A\ \Gamma = \mathcal{E}\ (\mathit{TER}\ f) - A\ \Gamma\ ]\!]$
  $\Longrightarrow$ *trimming* g

lemma assumes *trimming g*
shows *trimmingD-current*: *current* Γ *g*
and *trimmingD-wave*: *wave* Γ *g*
and *trimmingD-le*: ⋀*e*. *g e* ≤ *f e*
and *trimmingD-KIR*: ⟦ *x* ∈ *RF*° (*TER f*); *x* ∉ *A* Γ ⟧ ⟹ *KIR g x*
and *trimmingD-*$\mathcal{E}$: $\mathcal{E}$ (*TER g*) − *A* Γ = $\mathcal{E}$ (*TER f*) − *A* Γ
⟨*proof*⟩

lemma *ex-trimming*: — Lemma 4.8
assumes *f*: *current* Γ *f*
and *w*: *wave* Γ *f*
and *countable*: *countable* **E**
and *weight-finite*: ⋀*x*. *weight* Γ *x* ≠ ⊤
shows ∃ *g*. *trimming g*
⟨*proof*⟩

end

lemma *trimming-*$\mathcal{E}$:
fixes Γ (**structure**)
assumes *w*: *wave* Γ *f* **and** *trimming*: *trimming* Γ *f g*
shows $\mathcal{E}$ (*TER f*) = $\mathcal{E}$ (*TER g*)
⟨*proof*⟩

6.7 Composition of waves via quotients

definition *quotient-web* :: ('*v*, '*more*) *web-scheme* ⇒ '*v current* ⇒ ('*v*, '*more*) *web-scheme*
where — Modifications to original Definition 4.9: No incoming edges to nodes in *A*, *B* Γ − *A* Γ is not part of *A* such that *A* contains only vertices is disjoint from *B*. The weight of vertices in *B* saturated by *f* is therefore set to *0*.
quotient-web Γ *f* =
(|*edge* = λ*x y*. *edge* Γ *x y* ∧ *x* ∉ *roofed-circ* Γ (*TER*$_\Gamma$ *f*) ∧ *y* ∉ *roofed* Γ (*TER*$_\Gamma$ *f*),
weight = λ*x*. *if x* ∈ *RF*°$_\Gamma$ (*TER*$_\Gamma$ *f*) ∨ *x* ∈ *TER*$_\Gamma$ *f* ∩ *B* Γ *then 0 else weight* Γ *x*,
A = $\mathcal{E}_\Gamma$ (*TER*$_\Gamma$ *f*) − (*B* Γ − *A* Γ),
B = *B* Γ,
... = *web.more* Γ|)

lemma *quotient-web-sel* [*simp*]:
fixes Γ (**structure**) **shows**
edge (*quotient-web* Γ *f*) *x y* ⟷ *edge* Γ *x y* ∧ *x* ∉ *RF*° (*TER f*) ∧ *y* ∉ *RF* (*TER f*)
weight (*quotient-web* Γ *f*) *x* = (*if x* ∈ *RF*° (*TER f*) ∨ *x* ∈ *TER*$_\Gamma$ *f* ∩ *B* Γ *then 0 else weight* Γ *x*)
A (*quotient-web* Γ *f*) = $\mathcal{E}$ (*TER f*)− (*B* Γ − *A* Γ)
B (*quotient-web* Γ *f*) = *B* Γ

web.more (*quotient-web* Γ *f*) = *web.more* Γ
⟨*proof*⟩

lemma *vertex-quotient-webD*: **fixes** Γ (**structure**) **shows**
vertex (*quotient-web* Γ *f*) *x* ⟹ *vertex* Γ *x* ∧ *x* ∉ *RF*° (*TER f*)
⟨*proof*⟩

lemma *path-quotient-web*:
fixes Γ (**structure**)
assumes *path* Γ *x p y*
and *x* ∉ *RF*° (*TER f*)
and ⋀*z*. *z* ∈ *set p* ⟹ *z* ∉ *RF* (*TER f*)
shows *path* (*quotient-web* Γ *f*) *x p y*
⟨*proof*⟩

definition *restrict-current* :: ('*v*, '*more*) *web-scheme* ⇒ '*v current* ⇒ '*v current* ⇒ '*v current*
where *restrict-current* Γ *f g* = (λ(*x*, *y*). *g* (*x*, *y*) ∗ *indicator* (− *RF*°$_\Gamma$ (*TER*$_\Gamma$ *f*)) *x* ∗ *indicator* (− *RF*$_\Gamma$ (*TER*$_\Gamma$ *f*)) *y*)

abbreviation *restrict-curr* :: '*v current* ⇒ ('*v*, '*more*) *web-scheme* ⇒ '*v current* ⇒ '*v current* (- ↾ - '/ - [*100*, *0*, *100*] *100*)
where *restrict-curr g* Γ *f* ≡ *restrict-current* Γ *f g*

lemma *restrict-current-simps* [*simp*]: **fixes** Γ (**structure**) **shows**
(*g* ↾ Γ / *f*) (*x*, *y*) = (*g* (*x*, *y*) ∗ *indicator* (− *RF*° (*TER f*)) *x* ∗ *indicator* (− *RF* (*TER f*)) *y*)
⟨*proof*⟩

lemma *d-OUT-restrict-current-outside*: **fixes** Γ (**structure**) **shows**
x ∈ *RF*° (*TER f*) ⟹ *d-OUT* (*g* ↾ Γ / *f*) *x* = *0*
⟨*proof*⟩

lemma *d-IN-restrict-current-outside*: **fixes** Γ (**structure**) **shows**
x ∈ *RF* (*TER f*) ⟹ *d-IN* (*g* ↾ Γ / *f*) *x* = *0*
⟨*proof*⟩

lemma *restrict-current-le*: (*g* ↾ Γ / *f*) *e* ≤ *g e*
⟨*proof*⟩

lemma *d-OUT-restrict-current-le*: *d-OUT* (*g* ↾ Γ / *f*) *x* ≤ *d-OUT g x*
⟨*proof*⟩

lemma *d-IN-restrict-current-le*: *d-IN* (*g* ↾ Γ / *f*) *x* ≤ *d-IN g x*
⟨*proof*⟩

lemma *restrict-current-IN-not-RF*:
fixes Γ (**structure**)
assumes *g*: *current* Γ *g*

and x: $x \notin RF\ (TER\ f)$
shows *d-IN* $(g \restriction \Gamma\ /\ f)\ x = d\text{-}IN\ g\ x$
⟨*proof*⟩

lemma *restrict-current-IN-A*:
$a \in A\ (quotient\text{-}web\ \Gamma\ f) \Longrightarrow d\text{-}IN\ (g \restriction \Gamma\ /\ f)\ a = 0$
⟨*proof*⟩

lemma *restrict-current-nonneg*: $0 \leq g\ e \Longrightarrow 0 \leq (g \restriction \Gamma\ /\ f)\ e$
⟨*proof*⟩

lemma *in-SINK-restrict-current*: $x \in SINK\ g \Longrightarrow x \in SINK\ (g \restriction \Gamma\ /\ f)$
⟨*proof*⟩

lemma *SAT-restrict-current*:
fixes Γ (**structure**)
assumes f: *current* $\Gamma\ f$
and g: *current* $\Gamma\ g$
shows $SAT\ (quotient\text{-}web\ \Gamma\ f)\ (g \restriction \Gamma\ /\ f) = RF\ (TER\ f) \cup (SAT\ \Gamma\ g - A\ \Gamma)$
(**is** $SAT\ ?\Gamma\ ?g = ?rhs$)
⟨*proof*⟩

lemma *current-restrict-current*:
fixes Γ (**structure**)
assumes w: *wave* $\Gamma\ f$
and g: *current* $\Gamma\ g$
shows *current* $(quotient\text{-}web\ \Gamma\ f)\ (g \restriction \Gamma\ /\ f)$ (**is** *current* $?\Gamma\ ?g$)
⟨*proof*⟩

lemma *TER-restrict-current*:
fixes Γ (**structure**)
assumes f: *current* $\Gamma\ f$
and w: *wave* $\Gamma\ f$
and g: *current* $\Gamma\ g$
shows $TER\ g \subseteq TER_{quotient\text{-}web\ \Gamma\ f}\ (g \restriction \Gamma\ /\ f)$ (**is** - $\subseteq$ *?TER* **is** - $\subseteq$ $TER_{?\Gamma}$ *?g*)
⟨*proof*⟩

lemma *wave-restrict-current*:
fixes Γ (**structure**)
assumes f: *current* $\Gamma\ f$
and w: *wave* $\Gamma\ f$
and g: *current* $\Gamma\ g$
and w': *wave* $\Gamma\ g$
shows *wave* $(quotient\text{-}web\ \Gamma\ f)\ (g \restriction \Gamma\ /\ f)$ (**is** *wave* $?\Gamma\ ?g$)
⟨*proof*⟩

definition *plus-current* :: *'v current* ⇒ *'v current* ⇒ *'v current*
where *plus-current* $f\ g = (\lambda e.\ f\ e + g\ e)$

lemma *plus-current-simps* [*simp*]: *plus-current f g e* = *f e* + *g e*
⟨*proof*⟩

lemma *plus-zero-current* [*simp*]: *plus-current f zero-current* = *f*
⟨*proof*⟩

lemma *support-flow-plus-current*: *support-flow* (*plus-current f g*) ⊆ *support-flow f* ∪ *support-flow g*
⟨*proof*⟩

context
  fixes Γ :: ('*v*, '*more*) *web-scheme* (**structure**) **and** *f g*
  assumes *f*: *current* Γ *f*
  and *w*: *wave* Γ *f*
  and *g*: *current* (*quotient-web* Γ *f*) *g*
begin

lemma *OUT-plus-current*: *d-OUT* (*plus-current f g*) *x* = (*if x* ∈ *RF*° (*TER f*) *then d-OUT f x else d-OUT g x*) (**is** *d-OUT ?g - = -*)
⟨*proof*⟩

lemma *IN-plus-current*: *d-IN* (*plus-current f g*) *x* = (*if x* ∈ *RF* (*TER f*) *then d-IN f x else d-IN g x*) (**is** *d-IN ?g - = -*)
⟨*proof*⟩

lemma *in-TER-plus-current*:
  assumes *RF*: *x* ∉ *RF*° (*TER f*)
  and *x*: *x* ∈ $TER_{quotient\text{-}web\ \Gamma\ f\ g}$ (**is** *- ∈ ?TER -*)
  shows *x* ∈ *TER* (*plus-current f g*) (**is** *- ∈ TER ?g*)
⟨*proof*⟩

lemma *current-plus-current*: *current* Γ (*plus-current f g*) (**is** *current - ?g*)
⟨*proof*⟩

context
  assumes *w'*: *wave* (*quotient-web* Γ *f*) *g*
begin

lemma *separating-TER-plus-current*:
  assumes *x*: *x* ∈ *RF* (*TER f*) **and** *y*: *y* ∈ *B* Γ **and** *p*: *path* Γ *x p y*
  shows (∃ *z*∈*set p*. *z* ∈ *TER* (*plus-current f g*)) ∨ *x* ∈ *TER* (*plus-current f g*) (**is** *- ∨ - ∈ TER ?g*)
⟨*proof*⟩

lemma *wave-plus-current*: *wave* Γ (*plus-current f g*) (**is** *wave - ?g*)
⟨*proof*⟩

end

end

lemma *loose-quotient-web*:
fixes Γ :: ('*v*, '*more*) *web-scheme* (**structure**)
assumes *weight-finite*: $\bigwedge x.$ *weight* Γ $x \neq \top$
and f: *current* Γ f
and w: *wave* Γ f
and *maximal*: $\bigwedge w.$ $[\![$ *current* Γ w; *wave* Γ w; $f \leq w$ $]\!] \Longrightarrow f = w$
shows *loose* (*quotient-web* Γ f) (**is** *loose* *?Γ*)
⟨*proof*⟩

lemma *quotient-web-trimming*:
fixes Γ (**structure**)
assumes w: *wave* Γ f
and *trimming*: *trimming* Γ f g
shows *quotient-web* Γ f = *quotient-web* Γ g (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

6.8 Well-formed webs

locale *web* =
fixes Γ :: ('*v*, '*more*) *web-scheme* (**structure**)
assumes *A-in*: $x \in A\ \Gamma \Longrightarrow \neg$ *edge* Γ y x
and *B-out*: $x \in B\ \Gamma \Longrightarrow \neg$ *edge* Γ x y
and *A-vertex*: $A\ \Gamma \subseteq \mathbf{V}$
and *disjoint*: $A\ \Gamma \cap B\ \Gamma = \{\}$
and *no-loop*: $\bigwedge x. \neg$ *edge* Γ x x
and *weight-outside*: $\bigwedge x.\ x \notin \mathbf{V} \Longrightarrow$ *weight* Γ $x = 0$
and *weight-finite* [*simp*]: $\bigwedge x.$ *weight* Γ $x \neq \top$
begin

lemma *web-weight-update*:
assumes $\bigwedge x. \neg$ *vertex* Γ $x \Longrightarrow w\ x = 0$
and $\bigwedge x.\ w\ x \neq \top$
shows *web* ($\Gamma(\!|$*weight* := $w|\!)$)
⟨*proof*⟩

lemma *currentI* [*intro?*]:
assumes $\bigwedge x.$ *d-OUT* f $x \leq$ *weight* Γ x
and $\bigwedge x.$ *d-IN* f $x \leq$ *weight* Γ x
and *OUT-IN*: $\bigwedge x.$ $[\![$ $x \notin A\ \Gamma$; $x \notin B\ \Gamma$ $]\!] \Longrightarrow$ *d-OUT* f $x \leq$ *d-IN* f x
and *outside*: $\bigwedge e.\ e \notin \mathbf{E} \Longrightarrow f\ e = 0$
shows *current* Γ f
⟨*proof*⟩

lemma *currentD-finite-IN*:
assumes f: *current* Γ f
shows *d-IN* f $x \neq \top$

⟨*proof*⟩

lemma *currentD-finite-OUT*:
  assumes *f*: *current* Γ *f*
  shows *d-OUT f x* ≠ ⊤
⟨*proof*⟩

lemma *currentD-finite*:
  assumes *f*: *current* Γ *f*
  shows *f e* ≠ ⊤
⟨*proof*⟩

lemma *web-quotient-web*: *web* (*quotient-web* Γ *f*) (**is** *web ?*Γ)
⟨*proof*⟩

end

locale *countable-web* = *web* Γ
  for Γ :: ('*v*, '*more*) *web-scheme* (**structure**)
  +
  assumes *countable* [*simp*]: *countable* **E**
begin

lemma *countable-V* [*simp*]: *countable* **V**
⟨*proof*⟩

lemma *countable-web-quotient-web*: *countable-web* (*quotient-web* Γ *f*) (**is** *countable-web ?*Γ)
⟨*proof*⟩

end

6.9 Subtraction of a wave

definition *minus-web* :: ('*v*, '*more*) *web-scheme* ⇒ '*v current* ⇒ ('*v*, '*more*) *web-scheme* (**infixl** ⊖ *65*) — Definition 6.6
where Γ ⊖ *f* = Γ(|*weight* := λ*x*. *if x* ∈ *A* Γ *then weight* Γ *x* − *d-OUT f x else weight* Γ *x* + *d-OUT f x* − *d-IN f x*|)

lemma *minus-web-sel* [*simp*]:
  edge (Γ ⊖ *f*) = *edge* Γ
  weight (Γ ⊖ *f*) *x* = (*if x* ∈ *A* Γ *then weight* Γ *x* − *d-OUT f x else weight* Γ *x* + *d-OUT f x* − *d-IN f x*)
  A (Γ ⊖ *f*) = *A* Γ
  B (Γ ⊖ *f*) = *B* Γ
  $\mathbf{V}_{\Gamma \ominus f} = \mathbf{V}_{\Gamma}$
  $\mathbf{E}_{\Gamma \ominus f} = \mathbf{E}_{\Gamma}$
  web.more (Γ ⊖ *f*) = *web.more* Γ
⟨*proof*⟩

lemma *vertex-minus-web* [*simp*]: *vertex* ($\Gamma \ominus f$) = *vertex* Γ
⟨*proof*⟩

lemma *roofed-gen-minus-web* [*simp*]: *roofed-gen* ($\Gamma \ominus f$) = *roofed-gen* Γ
⟨*proof*⟩

lemma *minus-zero-current* [*simp*]: $\Gamma \ominus$ *zero-current* = Γ
⟨*proof*⟩

lemma (**in** *web*) *web-minus-web*:
assumes *f*: *current* Γ *f*
shows *web* ($\Gamma \ominus f$)
⟨*proof*⟩

6.10 Bipartite webs

locale *countable-bipartite-web* =
fixes Γ :: ('*v*, '*more*) *web-scheme* (**structure**)
assumes *bipartite-V*: $\mathbf{V} \subseteq A\ \Gamma \cup B\ \Gamma$
and *A-vertex*: $A\ \Gamma \subseteq \mathbf{V}$
and *bipartite-E*: *edge* $\Gamma\ x\ y \Longrightarrow x \in A\ \Gamma \wedge y \in B\ \Gamma$
and *disjoint*: $A\ \Gamma \cap B\ \Gamma = \{\}$
and *weight-outside*: $\bigwedge x.\ x \notin \mathbf{V} \Longrightarrow$ *weight* $\Gamma\ x = 0$
and *weight-finite* [*simp*]: $\bigwedge x.$ *weight* $\Gamma\ x \neq \top$
and *countable-E* [*simp*]: *countable* **E**
begin

lemma *not-vertex*: $[\![\ x \notin A\ \Gamma;\ x \notin B\ \Gamma\ ]\!] \Longrightarrow \neg$ *vertex* $\Gamma\ x$
⟨*proof*⟩

lemma *no-loop*: $\neg$ *edge* $\Gamma\ x\ x$
⟨*proof*⟩

lemma *edge-antiparallel*: *edge* $\Gamma\ x\ y \Longrightarrow \neg$ *edge* $\Gamma\ y\ x$
⟨*proof*⟩

lemma *A-in*: $x \in A\ \Gamma \Longrightarrow \neg$ *edge* $\Gamma\ y\ x$
⟨*proof*⟩

lemma *B-out*: $x \in B\ \Gamma \Longrightarrow \neg$ *edge* $\Gamma\ x\ y$
⟨*proof*⟩

sublocale *countable-web* ⟨*proof*⟩

lemma *currentD-OUT'*:
assumes *f*: *current* Γ *f*
and *x*: $x \notin A\ \Gamma$
shows *d-OUT f x* = *0*

⟨*proof*⟩

lemma *currentD-IN'*:
assumes *f*: *current* Γ *f*
and *x*: $x \notin B\ \Gamma$
shows *d-IN f x* = *0*
⟨*proof*⟩

lemma *current-bipartiteI* [*intro?*]:
assumes *OUT*: $\bigwedge x.$ *d-OUT f x* ≤ *weight* Γ *x*
and *IN*: $\bigwedge x.$ *d-IN f x* ≤ *weight* Γ *x*
and *outside*: $\bigwedge e.\ e \notin \mathbf{E} \Longrightarrow f\ e = 0$
shows *current* Γ *f*
⟨*proof*⟩

lemma *wave-bipartiteI* [*intro?*]:
assumes *sep*: *separating* Γ (*TER f*)
and *f*: *current* Γ *f*
shows *wave* Γ *f*
⟨*proof*⟩

lemma *web-flow-iff*: *web-flow* Γ *f* ⟷ *current* Γ *f*
⟨*proof*⟩

end

end

7 Reductions

theory *MFMC-Reduction* **imports**
MFMC-Web
begin

7.1 From a web to a bipartite web

definition *bipartite-web-of* :: ('*v*, '*more*) *web-scheme* ⇒ ('*v* + '*v*, '*more*) *web-scheme*
where
bipartite-web-of Γ =
(|*edge* = λ*uv uv'*. *case* (*uv*, *uv'*) *of* (*Inl u*, *Inr v*) ⇒ *edge* Γ *u v* ∨ *u* = *v* ∧ *u* ∈ *vertices* Γ ∧ *u* ∉ *A* Γ ∧ *v* ∉ *B* Γ | - ⇒ *False*,
weight = λ*uv*. *case uv of Inl u* ⇒ *if u* ∈ *B* Γ *then 0 else weight* Γ *u* | *Inr u* ⇒ *if u* ∈ *A* Γ *then 0 else weight* Γ *u*,
A = *Inl* ' (*vertices* Γ − *B* Γ),
B = *Inr* ' (− *A* Γ),
... = *web.more* Γ|)

lemma *bipartite-web-of-sel* [*simp*]: **fixes** Γ (**structure**) **shows**

edge (*bipartite-web-of* Γ) (*Inl u*) (*Inr v*) ⟷ *edge* Γ *u v* ∨ *u* = *v* ∧ *u* ∈ **V** ∧ *u* ∉ *A* Γ ∧ *v* ∉ *B* Γ
edge (*bipartite-web-of* Γ) *uv* (*Inl u*) ⟷ *False*
edge (*bipartite-web-of* Γ) (*Inr v*) *uv* ⟷ *False*
weight (*bipartite-web-of* Γ) (*Inl u*) = (*if u* ∈ *B* Γ *then 0 else weight* Γ *u*)
weight (*bipartite-web-of* Γ) (*Inr v*) = (*if v* ∈ *A* Γ *then 0 else weight* Γ *v*)
A (*bipartite-web-of* Γ) = *Inl* ‘ (**V** − *B* Γ)
B (*bipartite-web-of* Γ) = *Inr* ‘ (− *A* Γ)
⟨*proof*⟩

lemma *edge-bipartite-webI1*: *edge* Γ *u v* ⟹ *edge* (*bipartite-web-of* Γ) (*Inl u*) (*Inr v*)
⟨*proof*⟩

lemma *edge-bipartite-webI2*:
⟦ *u* ∈ $\mathbf{V}_\Gamma$; *u* ∉ *A* Γ; *u* ∉ *B* Γ ⟧ ⟹ *edge* (*bipartite-web-of* Γ) (*Inl u*) (*Inr u*)
⟨*proof*⟩

lemma *edge-bipartite-webE*:
fixes Γ (**structure**)
assumes *edge* (*bipartite-web-of* Γ) *uv uv′*
obtains *u v* **where** *uv* = *Inl u uv′* = *Inr v edge* Γ *u v*
| *u* **where** *uv* = *Inl u uv′* = *Inr u u* ∈ **V** *u* ∉ *A* Γ *u* ∉ *B* Γ
⟨*proof*⟩

lemma *E-bipartite-web*:
fixes Γ (**structure**) **shows**
$\mathbf{E}_{\textit{bipartite-web-of}\ \Gamma}$ = (λ(*x*, *y*). (*Inl x*, *Inr y*)) ‘ **E** ∪ (λ*x*. (*Inl x*, *Inr x*)) ‘ (**V** − *A* Γ − *B* Γ)
⟨*proof*⟩

context *web* **begin**

lemma *vertex-bipartite-web* [*simp*]:
vertex (*bipartite-web-of* Γ) (*Inl x*) ⟷ *vertex* Γ *x* ∧ *x* ∉ *B* Γ
vertex (*bipartite-web-of* Γ) (*Inr x*) ⟷ *vertex* Γ *x* ∧ *x* ∉ *A* Γ
⟨*proof*⟩

definition *separating-of-bipartite* :: (′*v* + ′*v*) *set* ⇒ ′*v set*
where
separating-of-bipartite S =
(*let A-S* = *Inl* −‘ *S*; *B-S* = *Inr* −‘ *S in* (*A-S* ∩ *B-S*) ∪ (*A* Γ ∩ *A-S*) ∪ (*B* Γ ∩ *B-S*))

context
fixes *S* :: (′*v* + ′*v*) *set*
assumes *sep*: *separating* (*bipartite-web-of* Γ) *S*
begin

Proof of separation follows [1]

lemma *separating-of-bipartite-aux*:
assumes *p*: *path* Γ *x p y* **and** *y*: $y \in B\ \Gamma$
and *x*: $x \in A\ \Gamma \vee Inr\ x \in S$
shows $(\exists\, z{\in}set\ p.\ z \in separating\text{-}of\text{-}bipartite\ S) \vee x \in separating\text{-}of\text{-}bipartite\ S$
⟨*proof*⟩

lemma *separating-of-bipartite*:
separating Γ (*separating-of-bipartite S*)
⟨*proof*⟩

end

lemma *current-bipartite-web-finite*:
assumes *f*: *current* (*bipartite-web-of* Γ) *f* (**is** *current* ?Γ -)
shows $f\ e \neq \top$
⟨*proof*⟩

definition *current-of-bipartite* :: $('v + 'v)\ current \Rightarrow 'v\ current$
where *current-of-bipartite f* = $(\lambda(x,\ y).\ f\ (Inl\ x,\ Inr\ y) * indicator\ \mathbf{E}\ (x,\ y))$

lemma *current-of-bipartite-simps* [*simp*]: *current-of-bipartite f* $(x,\ y) = f\ (Inl\ x,$ *Inr y*) ∗ *indicator* **E** $(x,\ y)$
⟨*proof*⟩

lemma *d-OUT-current-of-bipartite*:
assumes *f*: *current* (*bipartite-web-of* Γ) *f*
shows *d-OUT* (*current-of-bipartite f*) *x* = *d-OUT f* (*Inl x*) − *f* (*Inl x*, *Inr x*)
⟨*proof*⟩

lemma *d-IN-current-of-bipartite*:
assumes *f*: *current* (*bipartite-web-of* Γ) *f*
shows *d-IN* (*current-of-bipartite f*) *x* = *d-IN f* (*Inr x*) − *f* (*Inl x*, *Inr x*)
⟨*proof*⟩

lemma *current-current-of-bipartite*: — Lemma 6.3
assumes *f*: *current* (*bipartite-web-of* Γ) *f* (**is** *current* ?Γ -)
and *w*: *wave* (*bipartite-web-of* Γ) *f*
shows *current* Γ (*current-of-bipartite f*) (**is** *current* - *?f*)
⟨*proof*⟩

lemma *TER-current-of-bipartite*: — Lemma 6.3
assumes *f*: *current* (*bipartite-web-of* Γ) *f* (**is** *current* ?Γ -)
and *w*: *wave* (*bipartite-web-of* Γ) *f*
shows *TER* (*current-of-bipartite f*) = *separating-of-bipartite* ($TER_{bipartite\text{-}web\text{-}of\ \Gamma}$ *f*)
(**is** *TER ?f* = *separating-of-bipartite ?TER*)
⟨*proof*⟩

lemma *wave-current-of-bipartite*: — Lemma 6.3
  assumes *f*: *current* (*bipartite-web-of* Γ) *f* (**is** *current ?*Γ -)
  and *w*: *wave* (*bipartite-web-of* Γ) *f*
  shows *wave* Γ (*current-of-bipartite f*) (**is** *wave* - *?f*)
⟨*proof*⟩

end

context *countable-web* **begin**

lemma *countable-bipartite-web-of*: *countable-bipartite-web* (*bipartite-web-of* Γ) (**is** *countable-bipartite-web ?*Γ)
⟨*proof*⟩

end

context *web* **begin**

lemma *unhindered-bipartite-web-of*:
  assumes *loose*: *loose* Γ
  shows ¬ *hindered* (*bipartite-web-of* Γ)
⟨*proof*⟩

lemma (**in** −) *divide-less-1-iff-ennreal*: *a* / *b* < (*1*::*ennreal*) ⟷ (*0* < *b* ∧ *a* < *b* ∨ *b* = *0* ∧ *a* = *0* ∨ *b* = *top*)
  ⟨*proof*⟩

lemma *linkable-bipartite-web-ofD*:
  assumes *link*: *linkable* (*bipartite-web-of* Γ) (**is** *linkable ?*Γ)
  and *countable*: *countable* **E**
  shows *linkable* Γ
⟨*proof*⟩

end

7.2 Extending a wave by a linkage

lemma *linkage-quotient-webD*:
  fixes Γ :: (*'v*, *'more*) *web-scheme* (**structure**) **and** *h g*
  defines *k* ≡ *plus-current h g*
  assumes *f*: *current* Γ *f*
  and *w*: *wave* Γ *f*
  and *wg*: *web-flow* (*quotient-web* Γ *f*) *g* (**is** *web-flow ?*Γ -)
  and *link*: *linkage* (*quotient-web* Γ *f*) *g*
  and *trim*: *trimming* Γ *f h*
  shows *web-flow* Γ *k*
  and *orthogonal-current* Γ *k* (*E* (*TER f*))
⟨*proof*⟩

context *countable-web* **begin**

lemma *ex-orthogonal-current'*: — Lemma 4.15
assumes *loose-linkable*: $\bigwedge f.\ [\![\ \mathit{current}\ \Gamma\ f;\ \mathit{wave}\ \Gamma\ f;\ \mathit{loose}\ (\mathit{quotient\text{-}web}\ \Gamma\ f)\ ]\!] \Longrightarrow \mathit{linkable}\ (\mathit{quotient\text{-}web}\ \Gamma\ f)$
shows $\exists f\ S.\ \mathit{web\text{-}flow}\ \Gamma\ f \wedge \mathit{separating}\ \Gamma\ S \wedge \mathit{orthogonal\text{-}current}\ \Gamma\ f\ S$
⟨*proof*⟩

end

7.3 From a network to a web

definition *web-of-network* :: ('*v*, '*more*) *network-scheme* ⇒ ('*v edge*, '*more*) *web-scheme*
where
web-of-network Δ =
(|*edge* = λ(*x*, *y*) (*y'*, *z*). *y'* = *y* ∧ *edge* Δ *x y* ∧ *edge* Δ *y z*,
weight = *capacity* Δ,
A = {(*source* Δ, *x*)|*x*. *edge* Δ (*source* Δ) *x*},
B = {(*x*, *sink* Δ)|*x*. *edge* Δ *x* (*sink* Δ)},
... = *network.more* Δ|)

lemma *web-of-network-sel* [*simp*]:
fixes Δ (**structure**) **shows**
edge (*web-of-network* Δ) *e e'* ⟷ *e* ∈ **E** ∧ *e'* ∈ **E** ∧ *snd e* = *fst e'*
weight (*web-of-network* Δ) *e* = *capacity* Δ *e*
A (*web-of-network* Δ) = {(*source* Δ, *x*)|*x*. *edge* Δ (*source* Δ) *x*}
B (*web-of-network* Δ) = {(*x*, *sink* Δ)|*x*. *edge* Δ *x* (*sink* Δ)}
⟨*proof*⟩

lemma *vertex-web-of-network* [*simp*]:
vertex (*web-of-network* Δ) (*x*, *y*) ⟷ *edge* Δ *x y* ∧ (∃ *z*. *edge* Δ *y z* ∨ *edge* Δ *z x*)
⟨*proof*⟩

definition *flow-of-current* :: ('*v*, '*more*) *network-scheme* ⇒ '*v edge current* ⇒ '*v flow*
where *flow-of-current* Δ *f e* = *max* (*d-OUT f e*) (*d-IN f e*)

lemma *flow-flow-of-current*:
fixes Δ (**structure**) **and** Γ
defines [*simp*]: Γ ≡ *web-of-network* Δ
assumes *fw*: *web-flow* Γ *f*
shows *flow* Δ (*flow-of-current* Δ *f*) (**is** *flow* - *?f*)
⟨*proof*⟩

The reduction of Conjecture 1.2 to Conjecture 3.6 is flawed in [2]. Not every essential A-B separating set of vertices in *web-of-network* Δ is an s-t-cut in Δ, as the following counterexample shows.

The network Δ has five nodes *s*, *t*, *x*, *y* and *z* and edges (*s*, *x*), (*x*, *y*), (*y*,

z), (y, t) and (z, t). For *web-of-network* Δ, the set $S = \{(x, y), (y, z)\}$ is essential and A-B separating. ((x, y) is essential due to the path $[(y, z)]$ and (y, z) is essential due to the path $[(z, t)]$). However, S is not a cut in Δ because the node y has an outgoing edge that is in S and one that is not in S.

However, this can be remedied if all edges carry positive capacity. Then, orthogonality of the current rules out the above possibility.

lemma *cut-RF-separating*:
  fixes Δ (**structure**)
  assumes *sep*: *separating-network* Δ V
  and *sink*: *sink* $\Delta \notin V$
  shows *cut* Δ (RF^N V)
⟨*proof*⟩

context
  fixes Δ :: ('*v*, '*more*) *network-scheme* **and** Γ (**structure**)
  defines Γ-*def*: $\Gamma \equiv$ *web-of-network* Δ
begin

lemma *separating-network-cut-of-sep*:
  assumes *sep*: *separating* Γ S
  and *source-sink*: *source* $\Delta \neq$ *sink* Δ
  shows *separating-network* Δ (*fst* ‘ $\mathcal{E}$ S)
⟨*proof*⟩

definition *cut-of-sep* :: ('*v* × '*v*) *set* ⇒ '*v set*
where *cut-of-sep* $S = {RF^N}_\Delta$ (*fst* ‘ $\mathcal{E}$ S)

lemma *separating-cut*:
  assumes *sep*: *separating* Γ S
  and *neq*: *source* $\Delta \neq$ *sink* Δ
  and *sink-out*: $\bigwedge x.\ \neg$ *edge* Δ (*sink* Δ) x
  shows *cut* Δ (*cut-of-sep* S)
  ⟨*proof*⟩

context
  fixes f :: '*v edge current* **and** S
  assumes *wf*: *web-flow* Γ f
  and *ortho*: *orthogonal-current* Γ f S
  and *sep*: *separating* Γ S
  and *capacity-pos*: $\bigwedge e.\ e \in \mathbf{E}_\Delta \Longrightarrow$ *capacity* Δ $e > 0$
begin

private lemma *f*: *current* Γ f ⟨*proof*⟩

lemma *orthogonal-leave-RF*:
  assumes *e*: *edge* Δ x y
  and *x*: $x \in$ (*cut-of-sep* S)

and *y*: $y \notin (\mathit{cut\text{-}of\text{-}sep}\ S)$
shows $(x, y) \in S$
⟨*proof*⟩

lemma *orthogonal-flow-of-current*:
assumes *source-sink*: $\mathit{source}\ \Delta \neq \mathit{sink}\ \Delta$
and *sink-out*: $\bigwedge x.\ \neg\ \mathit{edge}\ \Delta\ (\mathit{sink}\ \Delta)\ x$
and *no-direct-edge*: $\neg\ \mathit{edge}\ \Delta\ (\mathit{source}\ \Delta)\ (\mathit{sink}\ \Delta)$ — Otherwise, A and B of the web would not be disjoint.
shows *orthogonal* Δ (*flow-of-current* $\Delta\ f$) (*cut-of-sep* S) (**is** *orthogonal - ?f ?S*)
⟨*proof*⟩

end

end

7.4 Avoiding antiparallel edges and self-loops

context *antiparallel-edges* **begin**

abbreviation $\mathit{cut}' :: 'a\ \mathit{vertex\ set} \Rightarrow 'a\ \mathit{set}$ **where** $\mathit{cut}'\ S \equiv \mathit{Vertex} -` S$

lemma *cut-cut'*: $\mathit{cut}\ \Delta''\ S \Longrightarrow \mathit{cut}\ \Delta\ (\mathit{cut}'\ S)$
⟨*proof*⟩

lemma *IN-Edge*: $\mathbf{IN}_{\Delta''}\ (\mathit{Edge}\ x\ y) = (\mathit{if\ edge}\ \Delta\ x\ y\ \mathit{then}\ \{\mathit{Vertex}\ x\}\ \mathit{else}\ \{\})$
⟨*proof*⟩

lemma *OUT-Edge*: $\mathbf{OUT}_{\Delta''}\ (\mathit{Edge}\ x\ y) = (\mathit{if\ edge}\ \Delta\ x\ y\ \mathit{then}\ \{\mathit{Vertex}\ y\}\ \mathit{else}\ \{\})$
⟨*proof*⟩

interpretation Δ'': *countable-network* Δ'' ⟨*proof*⟩

lemma *d-IN-Edge*:
assumes *f*: *flow* $\Delta''\ f$
shows *d-IN* $f\ (\mathit{Edge}\ x\ y) = f\ (\mathit{Vertex}\ x,\ \mathit{Edge}\ x\ y)$
⟨*proof*⟩

lemma *d-OUT-Edge*:
assumes *f*: *flow* $\Delta''\ f$
shows *d-OUT* $f\ (\mathit{Edge}\ x\ y) = f\ (\mathit{Edge}\ x\ y,\ \mathit{Vertex}\ y)$
⟨*proof*⟩

lemma *orthogonal-cut'*:
assumes *ortho*: *orthogonal* $\Delta''\ f\ S$
and *f*: *flow* $\Delta''\ f$
shows *orthogonal* Δ (*collect* f) ($\mathit{cut}'\ S$)
⟨*proof*⟩

end

context *countable-network* **begin**

lemma *countable-web-web-of-network*:
assumes *source-in*: $\bigwedge x.\ \neg\ edge\ \Delta\ x\ (source\ \Delta)$
and *sink-out*: $\bigwedge y.\ \neg\ edge\ \Delta\ (sink\ \Delta)\ y$
and *undead*: $\bigwedge x\ y.\ edge\ \Delta\ x\ y \Longrightarrow (\exists\, z.\ edge\ \Delta\ y\ z) \vee (\exists\, z.\ edge\ \Delta\ z\ x)$
and *source-sink*: $\neg\ edge\ \Delta\ (source\ \Delta)\ (sink\ \Delta)$
and *no-loop*: $\bigwedge x.\ \neg\ edge\ \Delta\ x\ x$
shows *countable-web* (*web-of-network* Δ) (**is** *countable-web* *?*Γ)
⟨*proof*⟩

lemma *max-flow-min-cut′*:
assumes *ex-orthogonal-current*: $\exists f\ S.$ *web-flow* (*web-of-network* Δ) $f \wedge$ *separating* (*web-of-network* Δ) $S \wedge$ *orthogonal-current* (*web-of-network* Δ) $f\ S$
and *source-in*: $\bigwedge x.\ \neg\ edge\ \Delta\ x\ (source\ \Delta)$
and *sink-out*: $\bigwedge y.\ \neg\ edge\ \Delta\ (sink\ \Delta)\ y$
and *undead*: $\bigwedge x\ y.\ edge\ \Delta\ x\ y \Longrightarrow (\exists\, z.\ edge\ \Delta\ y\ z) \vee (\exists\, z.\ edge\ \Delta\ z\ x)$
and *source-sink*: $\neg\ edge\ \Delta\ (source\ \Delta)\ (sink\ \Delta)$
and *no-loop*: $\bigwedge x.\ \neg\ edge\ \Delta\ x\ x$
and *capacity-pos*: $\bigwedge e.\ e \in \mathbf{E} \Longrightarrow capacity\ \Delta\ e > 0$
shows $\exists f\ S.$ *flow* $\Delta\ f \wedge$ *cut* $\Delta\ S \wedge$ *orthogonal* $\Delta\ f\ S$
⟨*proof*⟩

7.5 Eliminating zero edges and incoming edges to *source* and outgoing edges of *sink*

definition Δ''' :: *'v network* **where** $\Delta''' =$
$(\!|edge = \lambda x\ y.\ edge\ \Delta\ x\ y \wedge capacity\ \Delta\ (x,\ y) > 0 \wedge y \neq source\ \Delta \wedge x \neq sink\ \Delta,$
$capacity = \lambda(x,\ y).\ if\ x = sink\ \Delta \vee y = source\ \Delta\ then\ 0\ else\ capacity\ \Delta\ (x,\ y),$
$source = source\ \Delta,$
$sink = sink\ \Delta|\!)$

lemma Δ'''*-sel* [*simp*]:
edge $\Delta'''\ x\ y \longleftrightarrow$ *edge* $\Delta\ x\ y \wedge$ *capacity* $\Delta\ (x,\ y) > 0 \wedge y \neq source\ \Delta \wedge x \neq sink\ \Delta$
capacity $\Delta'''\ (x,\ y) =$ (*if* $x = sink\ \Delta \vee y = source\ \Delta$ *then 0 else capacity* $\Delta\ (x,\ y)$)
source $\Delta''' = source\ \Delta$
sink $\Delta''' = sink\ \Delta$
for $x\ y$ ⟨*proof*⟩

lemma Δ'''*-countable-network*: *countable-network* Δ'''
⟨*proof*⟩

lemma *flow-Δ'''*:
assumes *f*: *flow* Δ''' *f* **and** *cut*: *cut* Δ''' *S* **and** *ortho*: *orthogonal* Δ''' *f S*
shows *flow* Δ *f cut* Δ *S orthogonal* Δ *f S*
⟨*proof*⟩

end

end

8 The max-flow min-cut theorem in bounded networks

8.1 Linkages in unhindered bipartite webs

theory *MFMC-Bounded* **imports**
Matrix-For-Marginals
MFMC-Reduction
begin

context *countable-bipartite-web* **begin**

lemma *countable-A* [*simp*]: *countable* (*A* Γ)
⟨*proof*⟩

lemma *unhindered-criterion* [*rule-format*]:
assumes ¬ *hindered* Γ
shows ∀ *X* ⊆ *A* Γ. *finite X* ⟶ (∑⁺ *x*∈*X*. *weight* Γ *x*) ≤ (∑⁺ *y*∈**E** `` *X*. *weight* Γ *y*)
⟨*proof*⟩

end

lemma *nn-integral-count-space-top-approx*:
fixes *f* :: *nat* => *ennreal* **and** *b* :: *ennreal*
assumes *nn-integral* (*count-space UNIV*) *f* = *top*
and *b* < *top*
obtains *n* **where** *b* < *sum f* {..<*n*}
⟨*proof*⟩

lemma *One-le-of-nat-ennreal*: (*1* :: *ennreal*) ≤ *of-nat x* ⟷ *1* ≤ *x*
⟨*proof*⟩

locale *bounded-countable-bipartite-web* = *countable-bipartite-web* Γ
for Γ :: ('*v*, '*more*) *web-scheme* (**structure**)
+
assumes *bounded-B*: *x* ∈ *A* Γ ⟹ (∑⁺ *y* ∈ **E** `` {*x*}. *weight* Γ *y*) < ⊤
begin

theorem *unhindered-linkable-bounded*:

assumes ¬ *hindered* Γ
shows *linkable* Γ
⟨*proof*⟩

end

8.2 Glueing the reductions together

locale *bounded-countable-web* = *countable-web* Γ
for Γ :: ('*v*, '*more*) *web-scheme* (**structure**)
+
assumes *bounded-out*: $x \in \mathbf{V} - B\ \Gamma \Longrightarrow (\sum^+ y \in \mathbf{E}\ ``\ \{x\}.\ weight\ \Gamma\ y) < \top$
begin

lemma *bounded-countable-bipartite-web-of*: *bounded-countable-bipartite-web* (*bipartite-web-of* Γ)
(**is** *bounded-countable-bipartite-web* ?Γ)
⟨*proof*⟩

theorem *loose-linkable-bounded*:
assumes *loose* Γ
shows *linkable* Γ
⟨*proof*⟩

lemma *bounded-countable-web-quotient-web*: *bounded-countable-web* (*quotient-web* Γ *f*) (**is** *bounded-countable-web* ?Γ)
⟨*proof*⟩

lemma *ex-orthogonal-current*:
$\exists f\ S.$ *web-flow* Γ *f* ∧ *separating* Γ *S* ∧ *orthogonal-current* Γ *f S*
⟨*proof*⟩

end

locale *bounded-countable-network* = *countable-network* Δ
for Δ :: ('*v*, '*more*) *network-scheme* (**structure**) +
assumes *out*: ⟦ $x \in \mathbf{V}$; $x \neq$ *source* Δ ⟧ ⟹ *d-OUT* (*capacity* Δ) $x < \top$

context *antiparallel-edges* **begin**

lemma Δ''*-bounded-countable-network*: *bounded-countable-network* Δ''
if $\bigwedge x.$ ⟦ $x \in \mathbf{V}$; $x \neq$ *source* Δ ⟧ ⟹ *d-OUT* (*capacity* Δ) $x < \top$
⟨*proof*⟩

end

context *bounded-countable-network* **begin**

lemma *bounded-countable-web-web-of-network*:

assumes *source-in*: $\bigwedge x.\ \neg\ edge\ \Delta\ x\ (source\ \Delta)$
and *sink-out*: $\bigwedge y.\ \neg\ edge\ \Delta\ (sink\ \Delta)\ y$
and *undead*: $\bigwedge x\ y.\ edge\ \Delta\ x\ y \Longrightarrow (\exists\, z.\ edge\ \Delta\ y\ z) \vee (\exists\, z.\ edge\ \Delta\ z\ x)$
and *source-sink*: $\neg\ edge\ \Delta\ (source\ \Delta)\ (sink\ \Delta)$
and *no-loop*: $\bigwedge x.\ \neg\ edge\ \Delta\ x\ x$
shows *bounded-countable-web* (*web-of-network* Δ) (**is** *bounded-countable-web* $?\Gamma$)
⟨*proof*⟩

context begin

qualified lemma *max-flow-min-cut′-bounded*:
assumes *source-in*: $\bigwedge x.\ \neg\ edge\ \Delta\ x\ (source\ \Delta)$
and *sink-out*: $\bigwedge y.\ \neg\ edge\ \Delta\ (sink\ \Delta)\ y$
and *undead*: $\bigwedge x\ y.\ edge\ \Delta\ x\ y \Longrightarrow (\exists\, z.\ edge\ \Delta\ y\ z) \vee (\exists\, z.\ edge\ \Delta\ z\ x)$
and *source-sink*: $\neg\ edge\ \Delta\ (source\ \Delta)\ (sink\ \Delta)$
and *no-loop*: $\bigwedge x.\ \neg\ edge\ \Delta\ x\ x$
and *capacity-pos*: $\bigwedge e.\ e \in \mathbf{E} \Longrightarrow capacity\ \Delta\ e > 0$
shows $\exists f\ S.\ flow\ \Delta\ f \wedge cut\ \Delta\ S \wedge orthogonal\ \Delta\ f\ S$
⟨*proof*⟩ **lemma** *max-flow-min-cut″-bounded*:
assumes *sink-out*: $\bigwedge y.\ \neg\ edge\ \Delta\ (sink\ \Delta)\ y$
and *source-in*: $\bigwedge x.\ \neg\ edge\ \Delta\ x\ (source\ \Delta)$
and *no-loop*: $\bigwedge x.\ \neg\ edge\ \Delta\ x\ x$
and *capacity-pos*: $\bigwedge e.\ e \in \mathbf{E} \Longrightarrow capacity\ \Delta\ e > 0$
shows $\exists f\ S.\ flow\ \Delta\ f \wedge cut\ \Delta\ S \wedge orthogonal\ \Delta\ f\ S$
⟨*proof*⟩ **lemma** *max-flow-min-cut‴-bounded*:
assumes *sink-out*: $\bigwedge y.\ \neg\ edge\ \Delta\ (sink\ \Delta)\ y$
and *source-in*: $\bigwedge x.\ \neg\ edge\ \Delta\ x\ (source\ \Delta)$
and *capacity-pos*: $\bigwedge e.\ e \in \mathbf{E} \Longrightarrow capacity\ \Delta\ e > 0$
shows $\exists f\ S.\ flow\ \Delta\ f \wedge cut\ \Delta\ S \wedge orthogonal\ \Delta\ f\ S$
⟨*proof*⟩

lemma Δ'''*-bounded-countable-network*: *bounded-countable-network* Δ'''
⟨*proof*⟩

theorem *max-flow-min-cut-bounded*:
$\exists f\ S.\ flow\ \Delta\ f \wedge cut\ \Delta\ S \wedge orthogonal\ \Delta\ f\ S$
⟨*proof*⟩

end

end

end
theory *MFMC-Flow-Attainability* **imports**
MFMC-Network
begin

9 Attainability of flows in networks

9.1 Cleaning up flows

If there is a flow along antiparallel edges, it suffices to consider the difference.

definition *cleanup* :: *'a flow* ⇒ *'a flow*
where *cleanup f* = (λ(*a*, *b*). *if f* (*a*, *b*) > *f* (*b*, *a*) *then f* (*a*, *b*) − *f* (*b*, *a*) *else 0*)

lemma *cleanup-simps* [*simp*]:
cleanup f (*a*, *b*) = (*if f* (*a*, *b*) > *f* (*b*, *a*) *then f* (*a*, *b*) − *f* (*b*, *a*) *else 0*)
⟨*proof*⟩

lemma *value-flow-cleanup*:
assumes [*simp*]: ⋀*x*. *f* (*x*, *source* Δ) = *0*
shows *value-flow* Δ (*cleanup f*) = *value-flow* Δ *f*
⟨*proof*⟩

lemma *KIR-cleanup*:
assumes *KIR*: *KIR f x*
and *finite-IN*: *d-IN f x* ≠ ⊤
shows *KIR* (*cleanup f*) *x*
⟨*proof*⟩

locale *flow-attainability* = *countable-network* Δ
for Δ :: (*'v*, *'more*) *network-scheme* (**structure**)
+
assumes *finite-capacity*: ⋀*x*. *x* ≠ *sink* Δ ⟹ *d-IN* (*capacity* Δ) *x* ≠ ⊤ ∨ *d-OUT* (*capacity* Δ) *x* ≠ ⊤
and *no-loop*: ⋀*x*. ¬ *edge* Δ *x x*
and *source-in*: ⋀*x*. ¬ *edge* Δ *x* (*source* Δ)
begin

lemma *source-in-not-cycle*:
assumes *cycle* Δ *p*
shows (*x*, *source* Δ) ∉ *set* (*cycle-edges p*)
⟨*proof*⟩

lemma *source-out-not-cycle*:
cycle Δ *p* ⟹ (*source* Δ, *x*) ∉ *set* (*cycle-edges p*)
⟨*proof*⟩

lemma *flowD-source-IN*:
assumes *flow* Δ *f*
shows *d-IN f* (*source* Δ) = *0*
⟨*proof*⟩

lemma *flowD-finite-IN*:
assumes *f*: *flow* Δ *f* **and** *x*: *x* ≠ *sink* Δ
shows *d-IN f x* ≠ *top*

⟨*proof*⟩

lemma *flowD-finite-OUT*:
  assumes *flow* Δ *f x* ≠ *source* Δ *x* ≠ *sink* Δ
  shows *d-OUT f x* ≠ ⊤
⟨*proof*⟩

end

locale *flow-network* = *flow-attainability*
  +
  fixes *g* :: *'v flow*
  assumes *g*: *flow* Δ *g*
  and *g-finite*: *value-flow* Δ *g* ≠ ⊤
  and *nontrivial*: **V** − {*source* Δ, *sink* Δ} ≠ {}
begin

lemma *g-outside*: *e* ∉ **E** ⟹ *g e* = *0*
⟨*proof*⟩

lemma *g-loop* [*simp*]: *g* (*x*, *x*) = *0*
⟨*proof*⟩

lemma *finite-IN-g*: *x* ≠ *sink* Δ ⟹ *d-IN g x* ≠ *top*
⟨*proof*⟩

lemma *finite-OUT-g*:
  assumes *x* ≠ *sink* Δ
  shows *d-OUT g x* ≠ *top*
⟨*proof*⟩

lemma *g-source-in* [*simp*]: *g* (*x*, *source* Δ) = *0*
⟨*proof*⟩

lemma *finite-g* [*simp*]: *g e* ≠ *top*
  ⟨*proof*⟩

definition *enum-v* :: *nat* ⇒ *'v*
where *enum-v n* = *from-nat-into* (**V** − {*source* Δ, *sink* Δ}) (*fst* (*prod-decode n*))

lemma *range-enum-v*: *range enum-v* ⊆ **V** − {*source* Δ, *sink* Δ}
⟨*proof*⟩

lemma *enum-v-repeat*:
  assumes *x*: *x* ∈ **V** *x* ≠ *source* Δ *x* ≠ *sink* Δ
  shows ∃ *i'*>*i*. *enum-v i'* = *x*
⟨*proof*⟩

fun *h-plus* :: *nat* ⇒ *'v edge* ⇒ *ennreal*
where
  h-plus 0 (*x*, *y*) = (*if x* = *source* Δ *then g* (*x*, *y*) *else 0*)
| *h-plus* (*Suc i*) (*x*, *y*) =
  (*if enum-v* (*Suc i*) = *x* ∧ *d-OUT* (*h-plus i*) *x* < *d-IN* (*h-plus i*) *x then*
   let total = *d-IN* (*h-plus i*) *x* − *d-OUT* (*h-plus i*) *x*;
       share = *g* (*x*, *y*) − *h-plus i* (*x*, *y*);
       shares = *d-OUT g x* − *d-OUT* (*h-plus i*) *x*
   in h-plus i (*x*, *y*) + *share* ∗ *total* / *shares*
   else h-plus i (*x*, *y*))

lemma *h-plus-le-g*: *h-plus i e* ≤ *g e*
⟨*proof*⟩

lemma *h-plus-outside*: *e* ∉ **E** ⟹ *h-plus i e* = *0*
⟨*proof*⟩

lemma *h-plus-not-infty* [*simp*]: *h-plus i e* ≠ *top*
⟨*proof*⟩

lemma *h-plus-mono*: *h-plus i e* ≤ *h-plus* (*Suc i*) *e*
⟨*proof*⟩

lemma *h-plus-mono'*: *i* ≤ *j* ⟹ *h-plus i e* ≤ *h-plus j e*
⟨*proof*⟩

lemma *d-OUT-h-plus-not-infty'*: *x* ≠ *sink* Δ ⟹ *d-OUT* (*h-plus i*) *x* ≠ *top*
⟨*proof*⟩

lemma *h-plus-OUT-le-IN*:
  assumes *x* ≠ *source* Δ
  shows *d-OUT* (*h-plus i*) *x* ≤ *d-IN* (*h-plus i*) *x*
⟨*proof*⟩

lemma *h-plus-OUT-eq-IN*:
  assumes *enum*: *enum-v* (*Suc i*) = *x*
  shows *d-OUT* (*h-plus* (*Suc i*)) *x* = *d-IN* (*h-plus i*) *x*
⟨*proof*⟩

lemma *h-plus-source-in* [*simp*]: *h-plus i* (*x*, *source* Δ) = *0*
⟨*proof*⟩

lemma *h-plus-sum-finite*: (∑⁺ *e*. *h-plus i e*) ≠ *top*
⟨*proof*⟩

lemma *d-OUT-h-plus-not-infty* [*simp*]: *d-OUT* (*h-plus i*) *x* ≠ *top*
⟨*proof*⟩

definition *enum-cycle* :: *nat* ⇒ *'v path*
where *enum-cycle* = *from-nat-into* (*cycles* Δ)

lemma *cycle-enum-cycle* [*simp*]: *cycles* Δ ≠ {} ⟹ *cycle* Δ (*enum-cycle n*)
⟨*proof*⟩

context
  fixes *h'* :: *'v flow*
  assumes *finite-h'*: *h' e* ≠ *top*
begin

fun *h-minus-aux* :: *nat* ⇒ *'v edge* ⇒ *ennreal*
where
  h-minus-aux 0 e = *0*
| *h-minus-aux* (*Suc j*) *e* =
  (*if e* ∈ *set* (*cycle-edges* (*enum-cycle j*)) *then*
      h-minus-aux j e + *Min* {*h' e'* − *h-minus-aux j e'*|*e'*. *e'*∈*set* (*cycle-edges* (*enum-cycle j*))}
   else h-minus-aux j e)

lemma *h-minus-aux-le-h'*: *h-minus-aux j e* ≤ *h' e*
⟨*proof*⟩

lemma *h-minus-aux-finite* [*simp*]: *h-minus-aux j e* ≠ *top*
⟨*proof*⟩

lemma *h-minus-aux-mono*: *h-minus-aux j e* ≤ *h-minus-aux* (*Suc j*) *e*
⟨*proof*⟩

lemma *d-OUT-h-minus-aux*:
  assumes *cycles* Δ ≠ {}
  shows *d-OUT* (*h-minus-aux j*) *x* = *d-IN* (*h-minus-aux j*) *x*
⟨*proof*⟩

lemma *h-minus-aux-source*:
  assumes *cycles* Δ ≠ {}
  shows *h-minus-aux j* (*source* Δ, *y*) = *0*
⟨*proof*⟩

lemma *h-minus-aux-cycle*:
  fixes *j* **defines** *C* ≡ *enum-cycle j*
  assumes *cycles* Δ ≠ {}
  shows ∃ *e*∈*set* (*cycle-edges C*). *h-minus-aux* (*Suc j*) *e* = *h' e*
⟨*proof*⟩

end

fun *h-minus* :: *nat* ⇒ *'v edge* ⇒ *ennreal*
where

h-minus 0 e = 0
| *h-minus (Suc i) e = h-minus i e + (SUP j. h-minus-aux (λe'. h-plus (Suc i) e' − h-minus i e') j e)*

lemma *h-minus-le-h-plus*: *h-minus i e ≤ h-plus i e*
⟨*proof*⟩

lemma *finite-h'*: *h-plus (Suc i) e − h-minus i e ≠ top*
⟨*proof*⟩

lemma *h-minus-mono*: *h-minus i e ≤ h-minus (Suc i) e*
⟨*proof*⟩

lemma *h-minus-finite* [*simp*]: *h-minus i e ≠ ⊤*
⟨*proof*⟩

lemma *d-OUT-h-minus*:
assumes *cycles*: *cycles Δ ≠ {}*
shows *d-OUT (h-minus i) x = d-IN (h-minus i) x*
⟨*proof*⟩

lemma *h-minus-source*:
assumes *cycles Δ ≠ {}*
shows *h-minus n (source Δ, y) = 0*
⟨*proof*⟩

lemma *h-minus-source-in* [*simp*]: *h-minus i (x, source Δ) = 0*
⟨*proof*⟩

lemma *h-minus-OUT-finite* [*simp*]: *d-OUT (h-minus i) x ≠ top*
⟨*proof*⟩

lemma *h-minus-cycle*:
assumes *cycle Δ C*
shows *∃ e∈set (cycle-edges C). h-minus i e = h-plus i e*
⟨*proof*⟩

abbreviation *lim-h-plus* :: *'v edge ⇒ ennreal*
where *lim-h-plus e ≡ SUP n. h-plus n e*

abbreviation *lim-h-minus* :: *'v edge ⇒ ennreal*
where *lim-h-minus e ≡ SUP n. h-minus n e*

lemma *lim-h-plus-le-g*: *lim-h-plus e ≤ g e*
⟨*proof*⟩

lemma *lim-h-plus-finite* [*simp*]: *lim-h-plus e ≠ top*
⟨*proof*⟩

lemma *lim-h-minus-le-lim-h-plus*: *lim-h-minus e* ≤ *lim-h-plus e*
⟨*proof*⟩

lemma *lim-h-minus-finite* [*simp*]: *lim-h-minus e* ≠ *top*
⟨*proof*⟩

lemma *lim-h-minus-IN-finite* [*simp*]:
assumes *x* ≠ *sink* Δ
shows *d-IN lim-h-minus x* ≠ *top*
⟨*proof*⟩

lemma *lim-h-plus-OUT-IN*:
assumes *x* ≠ *source* Δ *x* ≠ *sink* Δ
shows *d-OUT lim-h-plus x* = *d-IN lim-h-plus x*
⟨*proof*⟩

lemma *lim-h-minus-OUT-IN*:
assumes *cycles*: *cycles* Δ ≠ {}
shows *d-OUT lim-h-minus x* = *d-IN lim-h-minus x*
⟨*proof*⟩

definition *h* :: *'v edge* ⇒ *ennreal*
where *h e* = *lim-h-plus e* − (*if cycles* Δ ≠ {} *then lim-h-minus e else 0*)

lemma *h-le-lim-h-plus*: *h e* ≤ *lim-h-plus e*
⟨*proof*⟩

lemma *h-le-g*: *h e* ≤ *g e*
⟨*proof*⟩

lemma *flow-h*: *flow* Δ *h*
⟨*proof*⟩

lemma *value-h-plus*: *value-flow* Δ (*h-plus i*) = *value-flow* Δ *g* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *value-h*: *value-flow* Δ *h* = *value-flow* Δ *g* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

definition *h-diff* :: *nat* ⇒ *'v edge* ⇒ *ennreal*
where *h-diff i e* = *h-plus i e* − (*if cycles* Δ ≠ {} *then h-minus i e else 0*)

lemma *d-IN-h-source* [*simp*]: *d-IN* (*h-diff i*) (*source* Δ) = *0*
⟨*proof*⟩

lemma *h-diff-le-h-plus*: *h-diff i e* ≤ *h-plus i e*
⟨*proof*⟩

lemma *h-diff-le-g*: *h-diff i e* $\leq$ *g e*
⟨*proof*⟩

lemma *h-diff-loop* [*simp*]: *h-diff i* (*x*, *x*) = *0*
⟨*proof*⟩

lemma *supp-h-diff-edges*: *support-flow* (*h-diff i*) $\subseteq$ **E**
⟨*proof*⟩

lemma *h-diff-OUT-le-IN*:
assumes *x* $\neq$ *source* Δ
shows *d-OUT* (*h-diff i*) *x* $\leq$ *d-IN* (*h-diff i*) *x*
⟨*proof*⟩

lemma *h-diff-cycle*:
assumes *cycle* Δ *p*
shows $\exists$ *e*$\in$*set* (*cycle-edges p*). *h-diff i e* = *0*
⟨*proof*⟩

lemma *d-IN-h-le-value'*: *d-IN* (*h-diff i*) *x* $\leq$ *value-flow* Δ (*h-plus i*)
⟨*proof*⟩

lemma *d-IN-h-le-value*: *d-IN h x* $\leq$ *value-flow* Δ *h* (**is** *?lhs* $\leq$ *?rhs*)
⟨*proof*⟩

lemma *flow-cleanup*: — Lemma 5.4
$\exists$ *h* $\leq$ *g*. *flow* Δ *h* $\wedge$ *value-flow* Δ *h* = *value-flow* Δ *g* $\wedge$ ($\forall$ *x*. *d-IN h x* $\leq$ *value-flow* Δ *h*)
⟨*proof*⟩

end

9.2 Residual network

context *countable-network* **begin**

definition *residual-network* :: *'v flow* $\Rightarrow$ (*'v*, *'more*) *network-scheme*
where *residual-network f* =
(|*edge* = λ*x y*. *edge* Δ *x y* $\vee$ *edge* Δ *y x* $\wedge$ *y* $\neq$ *source* Δ,
capacity = λ(*x*, *y*). *if edge* Δ *x y then capacity* Δ (*x*, *y*) − *f* (*x*, *y*) *else if y* = *source* Δ *then 0 else f* (*y*, *x*),
source = *source* Δ, *sink* = *sink* Δ, ... = *network.more* Δ |)

lemma *residual-network-sel* [*simp*]:
edge (*residual-network f*) *x y* $\longleftrightarrow$ *edge* Δ *x y* $\vee$ *edge* Δ *y x* $\wedge$ *y* $\neq$ *source* Δ
capacity (*residual-network f*) (*x*, *y*) = (*if edge* Δ *x y then capacity* Δ (*x*, *y*) − *f* (*x*, *y*) *else if y* = *source* Δ *then 0 else f* (*y*, *x*))
source (*residual-network f*) = *source* Δ
sink (*residual-network f*) = *sink* Δ

network.more (*residual-network f*) = *network.more* Δ
⟨*proof*⟩

lemma **E**-*residual-network*: $\mathbf{E}_{\textit{residual-network } f} = \mathbf{E} \cup \{(x,\ y).\ (y,\ x) \in \mathbf{E} \wedge y \neq \textit{source}\ \Delta\}$
⟨*proof*⟩

lemma *vertices-residual-network* [*simp*]: *vertex* (*residual-network f*) = *vertex* Δ
⟨*proof*⟩

inductive *wf-residual-network* :: *bool*
where ⟦ ⋀*x y*. (*x*, *y*) ∈ **E** ⟹ (*y*, *x*) ∉ **E**; (*source* Δ, *sink* Δ) ∉ **E** ⟧ ⟹ *wf-residual-network*

lemma *wf-residual-networkD*:
⟦ *wf-residual-network*; *edge* Δ *x y* ⟧ ⟹ ¬ *edge* Δ *y x*
⟦ *wf-residual-network*; *e* ∈ **E** ⟧ ⟹ *prod.swap e* ∉ **E**
⟦ *wf-residual-network*; *edge* Δ (*source* Δ) (*sink* Δ) ⟧ ⟹ *False*
⟨*proof*⟩

lemma *residual-countable-network*:
assumes *wf*: *wf-residual-network*
and *f*: *flow* Δ *f*
shows *countable-network* (*residual-network f*) (**is** *countable-network* ?Δ)
⟨*proof*⟩

end

context *antiparallel-edges* **begin**

interpretation Δ′: *countable-network* Δ′ ⟨*proof*⟩

lemma Δ′-*flow-attainability*:
assumes *flow-attainability-axioms* Δ
shows *flow-attainability* Δ′
⟨*proof*⟩

lemma Δ′-*wf-residual-network*:
assumes *no-loop*: ⋀*x*. ¬ *edge* Δ *x x*
shows Δ′.*wf-residual-network*
⟨*proof*⟩

end

9.3 The attainability theorem

context *flow-attainability* **begin**

lemma *residual-flow-attainability*:

assumes *wf*: *wf-residual-network*
and *f*: *flow* Δ *f*
shows *flow-attainability* (*residual-network f*) (**is** *flow-attainability* ?Δ)
⟨*proof*⟩

end

definition *plus-flow* :: ('*v*, '*more*) *graph-scheme* ⇒ '*v flow* ⇒ '*v flow* ⇒ '*v flow*
(**infixr** $\oplus_1$ *65*)
where *plus-flow G f g* = (λ(*x*, *y*). *if edge G x y then f* (*x*, *y*) + *g* (*x*, *y*) − *g* (*y*, *x*) *else 0*)

lemma *plus-flow-simps* [*simp*]: **fixes** *G* (**structure**) **shows**
(*f* ⊕ *g*) (*x*, *y*) = (*if edge G x y then f* (*x*, *y*) + *g* (*x*, *y*) − *g* (*y*, *x*) *else 0*)
⟨*proof*⟩

lemma *plus-flow-outside*: **fixes** *G* (**structure**) **shows** *e* ∉ **E** ⟹ (*f* ⊕ *g*) *e* = *0*
⟨*proof*⟩

lemma
fixes Δ (**structure**)
assumes *f-outside*: $\bigwedge$*e*. *e* ∉ **E** ⟹ *f e* = *0*
and *g-le-f*: $\bigwedge$*x y*. *edge* Δ *x y* ⟹ *g* (*y*, *x*) ≤ *f* (*x*, *y*)
shows *OUT-plus-flow*: *d-IN g x* ≠ *top* ⟹ *d-OUT* (*f* ⊕ *g*) *x* = *d-OUT f x* + ($\sum^+$ *y*∈*UNIV*. *g* (*x*, *y*) ∗ *indicator* **E** (*x*, *y*)) − ($\sum^+$ *y*. *g* (*y*, *x*) ∗ *indicator* **E** (*x*, *y*))
(**is** - ⟹ *?OUT* **is** - ⟹ - = - + *?g-out* − *?g-out*′)
and *IN-plus-flow*: *d-OUT g x* ≠ *top* ⟹ *d-IN* (*f* ⊕ *g*) *x* = *d-IN f x* + ($\sum^+$ *y*∈*UNIV*. *g* (*y*, *x*) ∗ *indicator* **E** (*y*, *x*)) − ($\sum^+$ *y*. *g* (*x*, *y*) ∗ *indicator* **E** (*y*, *x*))
(**is** - ⟹ *?IN* **is** - ⟹ - = - + *?g-in* − *?g-in*′)
⟨*proof*⟩

context *countable-network* **begin**

lemma *d-IN-plus-flow*:
assumes *wf*: *wf-residual-network*
and *f*: *flow* Δ *f*
and *g*: *flow* (*residual-network f*) *g*
shows *d-IN* (*f* ⊕ *g*) *x* ≤ *d-IN f x* + *d-IN g x*
⟨*proof*⟩

lemma *scale-flow*:
assumes *f*: *flow* Δ *f*
and *c*: *c* ≤ *1*
shows *flow* Δ (λ*e*. *c* ∗ *f e*)
⟨*proof*⟩

lemma *value-scale-flow*:
value-flow Δ (λ*e*. *c* ∗ *f e*) = *c* ∗ *value-flow* Δ *f*

⟨*proof*⟩

lemma *value-flow*:
  assumes *f*: *flow* Δ *f*
  and *source-out*: ⋀*y*. *edge* Δ (*source* Δ) *y* ⟷ *y* = *x*
  shows *value-flow* Δ *f* = *f* (*source* Δ, *x*)
⟨*proof*⟩

end

context *flow-attainability* **begin**

lemma *value-plus-flow*:
  assumes *wf*: *wf-residual-network*
  and *f*: *flow* Δ *f*
  and *g*: *flow* (*residual-network f*) *g*
  shows *value-flow* Δ (*f* ⊕ *g*) = *value-flow* Δ *f* + *value-flow* Δ *g*
⟨*proof*⟩

lemma *flow-residual-add*: — Lemma 5.3
  assumes *wf*: *wf-residual-network*
  and *f*: *flow* Δ *f*
  and *g*: *flow* (*residual-network f*) *g*
  shows *flow* Δ (*f* ⊕ *g*)
⟨*proof*⟩

definition *minus-flow* :: *'v flow* ⇒ *'v flow* ⇒ *'v flow* (**infixl** ⊖ *65*)
where
  f ⊖ *g* = (λ(*x*, *y*). *if edge* Δ *x y then f* (*x*, *y*) − *g* (*x*, *y*) *else if edge* Δ *y x then g* (*y*, *x*) − *f* (*y*, *x*) *else 0*)

lemma *minus-flow-simps* [*simp*]:
  (*f* ⊖ *g*) (*x*, *y*) = (*if edge* Δ *x y then f* (*x*, *y*) − *g* (*x*, *y*) *else if edge* Δ *y x then g* (*y*, *x*) − *f* (*y*, *x*) *else 0*)
⟨*proof*⟩

lemma *minus-flow*:
  assumes *wf*: *wf-residual-network*
  and *f*: *flow* Δ *f*
  and *g*: *flow* Δ *g*
  and *value-le*: *value-flow* Δ *g* ≤ *value-flow* Δ *f*
  and *f-finite*: *f* (*source* Δ, *x*) ≠ ⊤
  and *source-out*: ⋀*y*. *edge* Δ (*source* Δ) *y* ⟷ *y* = *x*
  shows *flow* (*residual-network g*) (*f* ⊖ *g*) (**is** *flow ?*Δ *?f*)
⟨*proof*⟩

lemma *value-minus-flow*:
  assumes *f*: *flow* Δ *f*
  and *g*: *flow* Δ *g*

and *value-le*: *value-flow* Δ $g \leq$ *value-flow* Δ f
and *source-out*: $\bigwedge y$. *edge* Δ (*source* Δ) $y \longleftrightarrow y = x$
shows *value-flow* Δ $(f \ominus g)$ = *value-flow* Δ f − *value-flow* Δ g (**is** *?value*)
⟨*proof*⟩

context
fixes α
defines $\alpha \equiv$ (*SUP* $g \in \{g.$ *flow* Δ $g\}.$ *value-flow* Δ g)
begin

lemma *flow-by-value*:
assumes $v < \alpha$
and *real*[*rule-format*]: $\forall f.\ \alpha = \top \longrightarrow$ *flow* Δ $f \longrightarrow$ *value-flow* Δ $f < \alpha$
obtains f **where** *flow* Δ f *value-flow* Δ $f = v$
⟨*proof*⟩

theorem *ex-max-flow′*:
assumes *wf*: *wf-residual-network*
assumes *source-out*: $\bigwedge y$. *edge* Δ (*source* Δ) $y \longleftrightarrow y = x$
and *nontrivial*: $\mathbf{V} - \{$*source* Δ, *sink* $\Delta\} \neq \{\}$
and *real*: $\alpha =$ *ennreal* α' **and** α'-*nonneg*[*simp*]: $0 \leq \alpha'$
shows $\exists f.$ *flow* Δ $f \wedge$ *value-flow* Δ $f = \alpha \wedge (\forall x.$ *d-IN* f $x \leq$ *value-flow* Δ $f)$
⟨*proof*⟩

theorem *ex-max-flow″*: — eliminate assumption of no antiparallel edges using locale *wf-residual-network*
assumes *source-out*: $\bigwedge y$. *edge* Δ (*source* Δ) $y \longleftrightarrow y = x$
and *nontrivial*: $\mathbf{E} \neq \{\}$
and *real*: $\alpha =$ *ennreal* α' **and** *nn*[*simp*]: $0 \leq \alpha'$
shows $\exists f.$ *flow* Δ $f \wedge$ *value-flow* Δ $f = \alpha \wedge (\forall x.$ *d-IN* f $x \leq$ *value-flow* Δ $f)$
⟨*proof*⟩

context begin — We eliminate the assumption of only one edge leaving the source by introducing a new source vertex.
private datatype (*plugins del*: *transfer size*) *′v′ node* = *SOURCE* | *Inner* (*inner*: *′v′*)

private lemma *not-Inner-conv*: $x \notin$ *range Inner* $\longleftrightarrow x =$ *SOURCE*
⟨*proof*⟩ **lemma** *inj-on-Inner* [*simp*]: *inj-on Inner A*
⟨*proof*⟩ **inductive** *edge′* :: *′v node* ⇒ *′v node* ⇒ *bool*
where
SOURCE: *edge′ SOURCE* (*Inner* (*source* Δ))
| *Inner*: *edge* Δ x $y \Longrightarrow$ *edge′* (*Inner* x) (*Inner* y)

private inductive-simps *edge′-simps* [*simp*]:
edge′ SOURCE x
edge′ (*Inner y*) *x*
edge′ y SOURCE
edge′ y (*Inner x*)

private fun *capacity′* :: *′v node flow*
where
  capacity′ (*SOURCE*, *Inner x*) = (*if x* = *source* Δ *then* α *else 0*)
| *capacity′* (*Inner x*, *Inner y*) = *capacity* Δ (*x*, *y*)
| *capacity′* - = *0*

private lemma *capacity′-source-in* [*simp*]: *capacity′* (*y*, *Inner* (*source* Δ)) = (*if y* = *SOURCE then* α *else 0*)
⟨*proof*⟩ **definition** Δ′ :: *′v node network*
where Δ′ = (|*edge* = *edge′*, *capacity* = *capacity′*, *source* = *SOURCE*, *sink* = *Inner* (*sink* Δ)|)

private lemma Δ′*-sel* [*simp*]:
  edge Δ′ = *edge′*
  capacity Δ′ = *capacity′*
  source Δ′ = *SOURCE*
  sink Δ′ = *Inner* (*sink* Δ)
⟨*proof*⟩ **lemma E-**Δ′: $\mathbf{E}_{\Delta'}$ = {(*SOURCE*, *Inner* (*source* Δ))} ∪ (λ(*x*, *y*). (*Inner x*, *Inner y*)) ‘ **E**
⟨*proof*⟩ **lemma** Δ′*-countable-network*:
  assumes α ≠ ⊤
  shows *countable-network* Δ′
⟨*proof*⟩ **lemma** Δ′*-flow-attainability*:
  assumes α ≠ ⊤
  shows *flow-attainability* Δ′
⟨*proof*⟩ **fun** *lift* :: *′v flow* ⇒ *′v node flow*
where
  lift f (*SOURCE*, *Inner y*) = (*if y* = *source* Δ *then value-flow* Δ *f else 0*)
| *lift f* (*Inner x*, *Inner y*) = *f* (*x*, *y*)
| *lift f* - = *0*

private lemma *d-OUT-lift-Inner* [*simp*]: *d-OUT* (*lift f*) (*Inner x*) = *d-OUT f x* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩ **lemma** *d-OUT-lift-SOURCE* [*simp*]: *d-OUT* (*lift f*) *SOURCE* = *value-flow* Δ *f* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩ **lemma** *d-IN-lift-Inner* [*simp*]:
  assumes *x* ≠ *source* Δ
  shows *d-IN* (*lift f*) (*Inner x*) = *d-IN f x* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩ **lemma** *d-IN-lift-source* [*simp*]: *d-IN* (*lift f*) (*Inner* (*source* Δ)) = *value-flow* Δ *f* + *d-IN f* (*source* Δ) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩ **lemma** *flow-lift* [*simp*]:
  assumes *flow* Δ *f*
  shows *flow* Δ′ (*lift f*)
⟨*proof*⟩ **abbreviation** (*input*) *unlift* :: *′v node flow* ⇒ *′v flow*
where *unlift f* ≡ (λ(*x*, *y*). *f* (*Inner x*, *Inner y*))

private lemma *flow-unlift* [*simp*]:
  assumes *f*: *flow* Δ′ *f*

shows *flow* Δ (*unlift f*)
⟨*proof*⟩ **lemma** *value-unlift*:
assumes *f*: *flow* Δ' *f*
shows *value-flow* Δ (*unlift f*) = *value-flow* Δ' *f*
⟨*proof*⟩

theorem *ex-max-flow*:
$\exists f$. *flow* Δ *f* $\wedge$ *value-flow* Δ *f* = α $\wedge$ ($\forall x$. *d-IN f x* $\leq$ *value-flow* Δ *f*)
⟨*proof*⟩

end

end

end

end

10 The max-flow min-cut theorems in unbounded networks

theory *MFMC-Unbounded* **imports**
MFMC-Web
MFMC-Flow-Attainability
MFMC-Reduction
begin

10.1 More about waves

lemma *SINK-plus-current*: *SINK* (*plus-current f g*) = *SINK f* $\cap$ *SINK g*
⟨*proof*⟩

abbreviation *plus-web* :: ('*v*, '*more*) *web-scheme* $\Rightarrow$ '*v current* $\Rightarrow$ '*v current* $\Rightarrow$ '*v current* (- $\frown_1$ - [*66*, *66*] *65*)
where *plus-web* Γ *f g* $\equiv$ *plus-current f* (*g* $\restriction$ Γ / *f*)

lemma *d-OUT-plus-web*:
fixes Γ (**structure**)
shows *d-OUT* (*f* $\frown$ *g*) *x* = *d-OUT f x* + *d-OUT* (*g* $\restriction$ Γ / *f*) *x* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *d-IN-plus-web*:
fixes Γ (**structure**)
shows *d-IN* (*f* $\frown$ *g*) *y* = *d-IN f y* + *d-IN* (*g* $\restriction$ Γ / *f*) *y* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *plus-web-greater*: *f e* $\leq$ (*f* $\frown_\Gamma$ *g*) *e*
⟨*proof*⟩

lemma *current-plus-web*:
fixes Γ (**structure**)
shows ⟦ *current* Γ *f*; *wave* Γ *f*; *current* Γ *g* ⟧ ⟹ *current* Γ (*f* ⌢ *g*)
⟨*proof*⟩

context
fixes Γ :: (*'v*, *'more*) *web-scheme* (**structure**)
and *f g* :: *'v current*
assumes *f*: *current* Γ *f*
and *w*: *wave* Γ *f*
and *g*: *current* Γ *g*
begin

context
fixes *x* :: *'v*
assumes *x*: *x* ∈ *ℰ* (*TER f* ∪ *TER g*)
begin

qualified lemma *RF-f*: *x* ∉ *RF*° (*TER f*)
⟨*proof*⟩ **lemma** *RF-g*: *x* ∉ *RF*° (*TER g*)
⟨*proof*⟩

lemma *TER-plus-web-aux*:
assumes *SINK*: *x* ∈ *SINK* (*g* ↿ Γ / *f*) (**is** - ∈ *SINK ?g*)
shows *x* ∈ *TER* (*f* ⌢ *g*)
⟨*proof*⟩ **lemma** *SINK-TER-in''*:
assumes ⋀*x*. *x* ∉ *RF* (*TER g*) ⟹ *d-OUT g x* = *0*
shows *x* ∈ *SINK g*
⟨*proof*⟩

end

lemma *wave-plus*: *wave* (*quotient-web* Γ *f*) (*g* ↿ Γ / *f*) ⟹ *wave* Γ (*f* ⌢ *g*)
⟨*proof*⟩

lemma *TER-plus-web''*:
assumes ⋀*x*. *x* ∉ *RF* (*TER g*) ⟹ *d-OUT g x* = *0*
shows *ℰ* (*TER f* ∪ *TER g*) ⊆ *TER* (*f* ⌢ *g*)
⟨*proof*⟩

lemma *TER-plus-web'*: *wave* Γ *g* ⟹ *ℰ* (*TER f* ∪ *TER g*) ⊆ *TER* (*f* ⌢ *g*)
⟨*proof*⟩

lemma *wave-plus'*: *wave* Γ *g* ⟹ *wave* Γ (*f* ⌢ *g*)
⟨*proof*⟩

end

lemma *RF-TER-plus-web*:
fixes Γ (**structure**)
assumes *f*: *current* Γ *f*
and *w*: *wave* Γ *f*
and *g*: *current* Γ *g*
and *w′*: *wave* Γ *g*
shows *RF* (*TER* (*f* ⌢ *g*)) = *RF* (*TER f* ∪ *TER g*)
⟨*proof*⟩

lemma *RF-TER-Sup*:
fixes Γ (**structure**)
assumes *f*: ⋀*f*. *f* ∈ *Y* ⟹ *current* Γ *f*
and *w*: ⋀*f*. *f* ∈ *Y* ⟹ *wave* Γ *f*
and *Y*: *Complete-Partial-Order.chain* (≤) *Y Y* ≠ {} *countable* (*support-flow* (*Sup Y*))
shows *RF* (*TER* (*Sup Y*)) = *RF* (⋃*f*∈*Y*. *TER f*)
⟨*proof*⟩

10.2 Hindered webs with reduced weights

context *countable-bipartite-web* **begin**

context
fixes *u* :: *′v* ⇒ *ennreal*
and ε
defines ε ≡ (∫⁺ *y*. *u y* ∂*count-space* (*B* Γ))
assumes *u-outside*: ⋀*x*. *x* ∉ *B* Γ ⟹ *u x* = *0*
and *finite-ε*: ε ≠ ⊤
begin

private lemma *u-A*: *x* ∈ *A* Γ ⟹ *u x* = *0*
⟨*proof*⟩ **lemma** *u-finite*: *u y* ≠ ⊤
⟨*proof*⟩

lemma *hindered-reduce*: — Lemma 6.7
assumes *u*: *u* ≤ *weight* Γ
assumes *hindered-by*: *hindered-by* (Γ(|*weight* := *weight* Γ − *u*|)) ε (**is** *hindered-by* *?*Γ -)
shows *hindered* Γ
⟨*proof*⟩

end

corollary *hindered-reduce-current*: — Corollary 6.8
fixes ε *g*
defines ε ≡ ∑⁺ *x*∈*B* Γ. *d-IN g x* − *d-OUT g x*
assumes *g*: *current* Γ *g*
and *ε-finite*: ε ≠ ⊤
and *hindered*: *hindered-by* (Γ ⊖ *g*) ε

shows *hindered* Γ
⟨*proof*⟩

end

10.3 Reduced weight in a loose web

definition *reduce-weight* :: ('*v*, '*more*) *web-scheme* ⇒ '*v* ⇒ *real* ⇒ ('*v*, '*more*) *web-scheme*
where *reduce-weight* Γ *x r* = Γ(|*weight* := λ*y*. *weight* Γ *y* − (*if x* = *y then r else* 0)|)

lemma *reduce-weight-sel* [*simp*]:
edge (*reduce-weight* Γ *x r*) = *edge* Γ
A (*reduce-weight* Γ *x r*) = *A* Γ
B (*reduce-weight* Γ *x r*) = *B* Γ
vertex (*reduce-weight* Γ *x r*) = *vertex* Γ
weight (*reduce-weight* Γ *x r*) *y* = (*if x* = *y then weight* Γ *x* − *r else weight* Γ *y*)
web.more (*reduce-weight* Γ *x r*) = *web.more* Γ
⟨*proof*⟩

lemma *essential-reduce-weight* [*simp*]: *essential* (*reduce-weight* Γ *x r*) = *essential* Γ
⟨*proof*⟩

lemma *roofed-reduce-weight* [*simp*]: *roofed-gen* (*reduce-weight* Γ *x r*) = *roofed-gen* Γ
⟨*proof*⟩

context *countable-bipartite-web* **begin**

context begin
private datatype (*plugins del*: *transfer size*) '*a vertex* = *SOURCE* | *SINK* | *Inner* (*inner*: '*a*)

private lemma *notin-range-Inner*: *x* ∉ *range Inner* ⟷ *x* = *SOURCE* ∨ *x* = *SINK*
⟨*proof*⟩ **lemma** *inj-Inner* [*simp*]: ⋀*A*. *inj-on Inner A*
⟨*proof*⟩

lemma *unhinder-bipartite*:
assumes *h*: ⋀*n* :: *nat*. *current* Γ (*h n*)
and *SAT*: ⋀*n*. (*B* Γ ∩ **V**) − {*b*} ⊆ *SAT* Γ (*h n*)
and *b*: *b* ∈ *B* Γ
and *IN*: (*SUP n*. *d-IN* (*h n*) *b*) = *weight* Γ *b*
and *h0-b*: ⋀*n*. *d-IN* (*h* 0) *b* ≤ *d-IN* (*h n*) *b*
and *b-V*: *b* ∈ **V**
shows ∃ *h'*. *current* Γ *h'* ∧ *wave* Γ *h'* ∧ *B* Γ ∩ **V** ⊆ *SAT* Γ *h'*
⟨*proof*⟩

end

lemma *countable-bipartite-web-reduce-weight*:
assumes *weight* $\Gamma\ x \geq w$
shows *countable-bipartite-web* (*reduce-weight* $\Gamma\ x\ w$)
⟨*proof*⟩

lemma *unhinder*: — Lemma 6.9
assumes *loose*: *loose* Γ
and *b*: $b \in B\ \Gamma$
and *wb*: *weight* $\Gamma\ b > 0$
and δ: $\delta > 0$
shows $\exists \varepsilon{>}0.\ \varepsilon < \delta \wedge \neg$ *hindered* (*reduce-weight* $\Gamma\ b\ \varepsilon$)
⟨*proof*⟩

end

10.4 Single-vertex saturation in unhindered bipartite webs

The proof of lemma 6.10 in [2] is flawed. The transfinite steps (taking the least upper bound) only preserves unhinderedness, but not looseness. However, the single steps to non-limit ordinals assumes that $\Omega - f_i$ is loose in order to apply Lemma 6.9.

Counterexample: The bipartite web with three nodes a_1, a_2, a_3 in A and two nodes b_1, b_2 in B and edges $(a_1,\ b_1)$, $(a_2,\ b_1)$, $(a_2,\ b_2)$, $(a_3,\ b_2)$ and weights $a_1 = a_3 = 1$ and $a_2 = 2$ and $b_1 = 3$ and $b_2 = 2$. Then, we can get a sequence of weight reductions on b_2 from *2* to *1.5*, *1.25*, *1.125*, etc. with limit *1*. All maximal waves in the restricted webs in the sequence are *zero-current*, so in the limit, we get $k = 0$ and $\varepsilon = 1$ for a_2 and b_2. Now, the restricted web for the two is not loose because it contains the wave which assigns 1 to $(a_3,\ b_2)$.

We prove a stronger version which only assumes and ensures on unhinderedness.

context *countable-bipartite-web* **begin**

lemma *web-flow-iff*: *web-flow* $\Gamma\ f \longleftrightarrow$ *current* $\Gamma\ f$
⟨*proof*⟩

lemma *countable-bipartite-web-minus-web*:
assumes *f*: *current* $\Gamma\ f$
shows *countable-bipartite-web* $(\Gamma \ominus f)$
⟨*proof*⟩

lemma *current-plus-current-minus*:
assumes *f*: *current* $\Gamma\ f$
and *g*: *current* $(\Gamma \ominus f)\ g$

shows *current* Γ (*plus-current f g*) (**is** *current - ?fg*)
⟨*proof*⟩

lemma *wave-plus-current-minus*:
assumes *f*: *current* Γ *f*
and *w*: *wave* Γ *f*
and *g*: *current* (Γ ⊖ *f*) *g*
and *w'*: *wave* (Γ ⊖ *f*) *g*
shows *wave* Γ (*plus-current f g*) (**is** *wave - ?fg*)
⟨*proof*⟩

lemma *minus-plus-current*:
assumes *f*: *current* Γ *f*
and *g*: *current* (Γ ⊖ *f*) *g*
shows Γ ⊖ *plus-current f g* = Γ ⊖ *f* ⊖ *g* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *unhindered-minus-web*:
assumes *unhindered*: ¬ *hindered* Γ
and *f*: *current* Γ *f*
and *w*: *wave* Γ *f*
shows ¬ *hindered* (Γ ⊖ *f*)
⟨*proof*⟩

lemma *loose-minus-web*:
assumes *unhindered*: ¬ *hindered* Γ
and *f*: *current* Γ *f*
and *w*: *wave* Γ *f*
and *maximal*: ⋀*w*. ⟦ *current* Γ *w*; *wave* Γ *w*; *f* ≤ *w* ⟧ ⟹ *f* = *w*
shows *loose* (Γ ⊖ *f*) (**is** *loose ?*Γ)
⟨*proof*⟩

lemma *weight-minus-web*:
assumes *f*: *current* Γ *f*
shows *weight* (Γ ⊖ *f*) *x* = (*if x* ∈ *A* Γ *then weight* Γ *x* − *d-OUT f x else weight* Γ *x* − *d-IN f x*)
⟨*proof*⟩

lemma (**in** −) *separating-minus-web* [*simp*]: *separating-gen* (*G* ⊖ *f*) = *separating-gen G*
⟨*proof*⟩

lemma *current-minus*:
assumes *f*: *current* Γ *f*
and *g*: *current* Γ *g*
and *le*: ⋀*e*. *g e* ≤ *f e*
shows *current* (Γ ⊖ *g*) (*f* − *g*)
⟨*proof*⟩

lemma
assumes *w*: *wave* Γ *f*
and *g*: *current* Γ *g*
and *le*: ⋀*e*. *g e* ≤ *f e*
shows *wave-minus*: *wave* (Γ ⊖ *g*) (*f* − *g*)
and *TER-minus*: *TER f* ⊆ *TER*$_{\Gamma \ominus g}$ (*f* − *g*)
⟨*proof*⟩

lemma (**in** −) *essential-minus-web* [*simp*]: *essential* (Γ ⊖ *f*) = *essential* Γ
⟨*proof*⟩

lemma (**in** −) *RF-in-essential*: **fixes** *B* **shows** *essential* Γ *B S x* ⟹ *x* ∈ *roofed-gen* Γ *B S* ⟷ *x* ∈ *S*
⟨*proof*⟩

lemma (**in** −) *d-OUT-fun-upd*:
assumes *f* (*x*, *y*) ≠ ⊤ *f* (*x*, *y*) ≥ *0 k* ≠ ⊤ *k* ≥ *0*
shows *d-OUT* (*f*((*x*, *y*) := *k*)) *x*′ = (*if x* = *x*′ *then d-OUT f x* − *f* (*x*, *y*) + *k* *else d-OUT f x*′)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *unhindered-saturate1*: — Lemma 6.10
assumes *unhindered*: ¬ *hindered* Γ
and *a*: *a* ∈ *A* Γ
shows ∃*f*. *current* Γ *f* ∧ *d-OUT f a* = *weight* Γ *a* ∧ ¬ *hindered* (Γ ⊖ *f*)
⟨*proof*⟩

end

10.5 Linkability of unhindered bipartite webs

context *countable-bipartite-web* **begin**

theorem *unhindered-linkable*:
assumes *unhindered*: ¬ *hindered* Γ
shows *linkable* Γ
⟨*proof*⟩

end

context *countable-web* **begin**

theorem *loose-linkable*: — Theorem 6.2
assumes *loose* Γ
shows *linkable* Γ
⟨*proof*⟩

lemma *ex-orthogonal-current*: — Lemma 4.15
$\exists f\ S.$ *web-flow* $\Gamma\ f \wedge$ *separating* $\Gamma\ S \wedge$ *orthogonal-current* $\Gamma\ f\ S$
⟨*proof*⟩

end

10.6 Glueing the reductions together

context *countable-network* **begin**

context begin

qualified lemma *max-flow-min-cut′*:
assumes *source-in*: $\bigwedge x.\ \neg$ *edge* $\Delta\ x$ (*source* Δ)
and *sink-out*: $\bigwedge y.\ \neg$ *edge* Δ (*sink* Δ) y
and *undead*: $\bigwedge x\ y.$ *edge* $\Delta\ x\ y \Longrightarrow (\exists z.$ *edge* $\Delta\ y\ z) \vee (\exists z.$ *edge* $\Delta\ z\ x)$
and *source-sink*: $\neg$ *edge* Δ (*source* Δ) (*sink* Δ)
and *no-loop*: $\bigwedge x.\ \neg$ *edge* $\Delta\ x\ x$
and *capacity-pos*: $\bigwedge e.\ e \in \mathbf{E} \Longrightarrow$ *capacity* $\Delta\ e > 0$
shows $\exists f\ S.$ *flow* $\Delta\ f \wedge$ *cut* $\Delta\ S \wedge$ *orthogonal* $\Delta\ f\ S$
⟨*proof*⟩ **lemma** *max-flow-min-cut″*:
assumes *sink-out*: $\bigwedge y.\ \neg$ *edge* Δ (*sink* Δ) y
and *source-in*: $\bigwedge x.\ \neg$ *edge* $\Delta\ x$ (*source* Δ)
and *no-loop*: $\bigwedge x.\ \neg$ *edge* $\Delta\ x\ x$
and *capacity-pos*: $\bigwedge e.\ e \in \mathbf{E} \Longrightarrow$ *capacity* $\Delta\ e > 0$
shows $\exists f\ S.$ *flow* $\Delta\ f \wedge$ *cut* $\Delta\ S \wedge$ *orthogonal* $\Delta\ f\ S$
⟨*proof*⟩ **lemma** *max-flow-min-cut‴*:
assumes *sink-out*: $\bigwedge y.\ \neg$ *edge* Δ (*sink* Δ) y
and *source-in*: $\bigwedge x.\ \neg$ *edge* $\Delta\ x$ (*source* Δ)
and *capacity-pos*: $\bigwedge e.\ e \in \mathbf{E} \Longrightarrow$ *capacity* $\Delta\ e > 0$
shows $\exists f\ S.$ *flow* $\Delta\ f \wedge$ *cut* $\Delta\ S \wedge$ *orthogonal* $\Delta\ f\ S$
⟨*proof*⟩

theorem *max-flow-min-cut*:
$\exists f\ S.$ *flow* $\Delta\ f \wedge$ *cut* $\Delta\ S \wedge$ *orthogonal* $\Delta\ f\ S$
⟨*proof*⟩

end

end

end

theory *Max-Flow-Min-Cut-Countable* **imports**
MFMC-Bounded
MFMC-Unbounded
begin

11 The Max-Flow Min-Cut theorem

theorem *max-flow-min-cut-countable*:
fixes Δ (**structure**)
assumes *countable-E* [*simp*]: *countable* **E**
and *source-neq-sink* [*simp*]: *source* $\Delta \neq$ *sink* Δ
and *capacity-outside*: $\forall\, e.\ e \notin \mathbf{E} \longrightarrow$ *capacity* $\Delta\ e = 0$
and *capacity-finite* [*simp*]: $\forall\, e.$ *capacity* $\Delta\ e \neq \top$
shows $\exists f\ S.$ *flow* $\Delta\ f \wedge$ *cut* $\Delta\ S \wedge$ *orthogonal* $\Delta\ f\ S$
⟨*proof*⟩

hide-const (**open**) *A B weight*

end

theory *Rel-PMF-Characterisation* **imports**
Matrix-For-Marginals
begin

12 Characterisation of *rel-pmf*

proposition *rel-pmf-measureI*:
fixes $p :: 'a\ pmf$ **and** $q :: 'b\ pmf$
assumes *le*: $\bigwedge A.$ *measure* (*measure-pmf p*) $A \leq$ *measure* (*measure-pmf q*) $\{y.\ \exists\, x{\in}A.\ R\ x\ y\}$
shows *rel-pmf R p q*
⟨*proof*⟩

corollary *rel-pmf-distr-mono*: *rel-pmf R OO rel-pmf S* $\leq$ *rel-pmf* (*R OO S*)
— This fact has already been proven for the registration of $'a\ pmf$ as a BNF, but this proof is much shorter and more elegant. See [3] for a comparison of formalisations.
⟨*proof*⟩

12.1 Code generation for *rel-pmf*

proposition *rel-pmf-measureI'*:
fixes $p :: 'a\ pmf$ **and** $q :: 'b\ pmf$
assumes *le*: $\bigwedge A.\ A \subseteq$ *set-pmf p* $\Longrightarrow$ *measure-pmf.prob p A* $\leq$ *measure-pmf.prob q* $\{y \in$ *set-pmf q*. $\exists\, x{\in}A.\ R\ x\ y\}$
shows *rel-pmf R p q*
⟨*proof*⟩

lemma *rel-pmf-code* [*code*]:
rel-pmf R p q $\longleftrightarrow$
(*let B = set-pmf q in*
$\forall\, A{\in}$*Pow* (*set-pmf p*). *measure-pmf.prob p A* $\leq$ *measure-pmf.prob q* (*snd* ‘ *Set.filter* (*case-prod R*) ($A \times B$)))
⟨*proof*⟩

end

References

[1] R. Aharoni. Menger's theorem for graphs containing no infinite paths. *Europ. J. Combinatorics*, 4:201–204, 1983.

[2] R. Aharoni, E. Berger, A. Georgakopoulos, A. Perlstein, and P. Sprüssel. The max-flow min-cut theorem for countable networks. *J. Combin. Theory Ser. B*, 101:1–17, 2011.

[3] J. Hölzl, A. Lochbihler, and D. Traytel. A formalized hierarchy of probabilistic system types. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving (ITP 2015)*, volume 9236 of *LNCS*, pages 203–220. Springer, 2015.

[4] H. G. Kellerer. Funktionen auf Produkträumen mit vorgegebenen Marginal-Funktionen. *Math. Annalen*, 144:323–344, 1961.