Formalization of Timely Dataflow's Progress Tracking Protocol

Matthias Brun, Sára Decova, Andrea Lattuada, and Dmitriy Traytel

May 26, 2024

Abstract

Large-scale stream processing systems often follow the dataflow paradigm, which enforces a program structure that exposes a high degree of parallelism. The Timely Dataflow distributed system supports expressive cyclic dataflows for which it offers low-latency data- and pipeline-parallel stream processing. To achieve high expressiveness and performance, Timely Dataflow uses an intricate distributed protocol for tracking the computation's progress. We formalize this progress tracking protocol and verify its safety. Our formalization is described in detail in the forthcoming ITP'21 paper [3].

Contents

1 Introduction

The dataflow programming model represents a program as a directed graph of interconnected operators that perform per-tuple data transformations. A message (an incoming datum) arrives at an input (a root of the dataflow) and flows along the graph's edges into operators. Each operator takes the message, processes it, and emits any resulting derived messages.

In a dataflow system, all messages are associated with a timestamp, and operator instances need to know up-to-date (timestamp) *frontiers*—lower bounds on what timestamps may still appear as their inputs. When informed that all data for a range of timestamps has been delivered, an operator instance can complete the computation on input data for that range of timestamps, produce the resulting output, and retire those timestamps.

A *progress tracking mechanism* is a core component of the dataflow system. It receives information on outstanding timestamps from operator instances, exchanges this information with other system workers (cores, nodes) and disseminates up-to-date approximations of the frontiers to all operator instances. This AFP entry formally models and proves the safety of the progress tracking protocol of *Timely Dataflow* [1, 4], a dataflow programming model and a state-of-the-art streaming, data-parallel, distributed data processor. Specifically, we prove that the progress tracking protocol computes frontiers that always constitute safe lower bounds on what timestamps may still appear on the operator inputs. The formalization is described in detail in the forthcoming ITP'21 paper [3].

The ITP paper [3] closely follows this formalization's structure. In particular, the paper's presentation is split into four main sections each of which is present in the formalization (each in a separate theory file):

Algorithm/protocol	Section in this proof document	Section in [3]	Theory file
Abadi et al. [2]'s clocks protocol	Section 3	Section 3	Exchange_Abadi
Exchange protocol	Section 4	Section 4	Exchange
Local propagation algorithm	Section 7	Section 5	Propagate
Combined protocol	Section 8	Section 6	Combined

2 Auxiliary Lemmas

unbundle *multiset.lifting*

2.1 General

lemma *sum-list-hd-tl*:

fixes *xs* :: (- :: *group-add*) *list*
shows *xs* ≠ [] ⟹ *sum-list* (*tl xs*) = (− *hd xs*) + *sum-list xs*
by (*cases xs*) *simp-all*

2.2 Sums

lemma *Sum-eq-pick-changed-elem*:
assumes *finite M*
and *m* ∈ *M f m* = *g m* + Δ
and ⋀*n*. *n* ≠ *m* ∧ *n* ∈ *M* ⟹ *f n* = *g n*
shows (∑ *x*∈*M*. *f x*) = (∑ *x*∈*M*. *g x*) + Δ
using *assms*
proof (*induct M arbitrary*: *m rule*: *finite-induct*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert x F*)
then show *?case*
proof (*cases x*=*m*)
case *True*
with *insert* **have** *sum f F* = *sum g F*
by (*intro sum.cong*[*OF refl*]) *force*
with *insert True* **show** *?thesis*
by (*auto simp*: *add.commute add.left-commute*)
next
case *False*
with *insert* **show** *?thesis*
by (*auto simp*: *add.assoc*)
qed
qed

lemma *sum-pos-ex-elem-pos*: (*0*::*int*) < (∑ *m*∈*M*. *f m*) ⟹ ∃ *m*∈*M*. *0* < *f m*
by (*meson not-le sum-nonpos*)

lemma *sum-if-distrib-add*: *finite A* ⟹ *b* ∈ *A* ⟹ (∑ *a*∈*A*. *if a*=*b then X b* + *Y a else X a*) = (∑ *a*∈*A*. *X a*) + *Y b*
by (*simp add*: *Sum-eq-pick-changed-elem*)

2.3 Partial Orders

lemma (**in** *order*) *order-finite-set-exists-foundation*:
fixes *t* :: *'a*
assumes *finite M*
and *t* ∈ *M*
shows ∃ *s*∈*M*. *s* ≤ *t* ∧ (∀ *u*∈*M*. ¬ *u* < *s*)
using *assms*
by (*simp add*: *dual-order.strict-iff-order finite-has-minimal2*)

lemma *order-finite-set-obtain-foundation*:
fixes *t* :: - :: *order*

assumes *finite M*
and $t \in M$
obtains s **where** $s \in M\ s \leq t\ \forall u{\in}M.\ \neg\ u < s$
using *assms order-finite-set-exists-foundation* **by** *blast*

2.4 Multisets

lemma *finite-nonzero-count*: *finite* $\{t.\ count\ M\ t > 0\}$
using *count* **by** *auto*

lemma *finite-count*[*simp*]: *finite* $\{t.\ count\ M\ t > i\}$
by (*rule finite-subset*[*OF - finite-nonzero-count*[*of M*]]) (*auto simp only*: *set-mset-def*)

2.5 Signed Multisets

lemma *zcount-zmset-of-nonneg*[*simp*]: $0 \leq zcount\ (zmset\text{-}of\ M)\ t$
by *simp*

lemma *finite-zcount-pos*[*simp*]: *finite* $\{t.\ zcount\ M\ t > 0\}$
apply *transfer*
subgoal for *M*
apply (*rule finite-subset*[*OF - finite-Un*[*THEN iffD2*, *OF conjI*[*OF finite-nonzero-count finite-nonzero-count*]], *of - fst M snd M*])
apply (*auto simp only*: *set-mset-def fst-conv snd-conv split*: *prod.splits*)
done
done

lemma *finite-zcount-neg*[*simp*]: *finite* $\{t.\ zcount\ M\ t < 0\}$
apply *transfer*
subgoal for *M*
apply (*rule finite-subset*[*OF - finite-Un*[*THEN iffD2*, *OF conjI*[*OF finite-nonzero-count finite-nonzero-count*]], *of - fst M snd M*])
apply (*auto simp only*: *set-mset-def fst-conv snd-conv split*: *prod.splits*)
done
done

lemma *pos-zcount-in-zmset*: $0 < zcount\ M\ x \Longrightarrow x \in\#_z\ M$
by (*simp add*: *zcount-inI*)

lemma *zmset-elem-nonneg*: $x \in\#_z\ M \Longrightarrow (\bigwedge x.\ x \in\#_z\ M \Longrightarrow 0 \leq zcount\ M\ x) \Longrightarrow 0 < zcount\ M\ x$
by (*simp add*: *order.order-iff-strict zcount-eq-zero-iff*)

lemma *zero-le-sum-single*: $0 \leq zcount\ (\sum x{\in}M.\ \{\#f\ x\#\}_z)\ t$
by (*induct M rule*: *infinite-finite-induct*) *auto*

lemma *mem-zmset-of*[*simp*]: $x \in\#_z\ zmset\text{-}of\ M \longleftrightarrow x \in\#\ M$
by (*simp add*: *set-zmset-def*)

lemma *mset-neg-minus*: *mset-neg* (*abs-zmultiset* (*Mp*,*Mn*)) = *Mn*−*Mp*

by (*simp add*: *mset-neg.abs-eq*)

lemma *mset-pos-minus*: *mset-pos* (*abs-zmultiset* (*Mp*,*Mn*)) = *Mp*−*Mn*
by (*simp add*: *mset-pos.abs-eq*)

lemma *mset-neg-sum-set*: (⋀*m*. *m* ∈ *M* ⟹ *mset-neg* (*f m*) = {#}) ⟹ *mset-neg* (∑ *m*∈*M*. *f m*) = {#}
by (*induct M rule*: *infinite-finite-induct*) *auto*

lemma *mset-neg-empty-iff*: *mset-neg M* = {#} ⟷ (∀ *t*. *0* ≤ *zcount M t*)
apply *rule*
subgoal
by (*metis add.commute add.right-neutral mset-pos-as-neg zcount-zmset-of-nonneg zmset-of-empty*)
subgoal
apply (*induct rule*: *zmultiset.abs-induct*)
subgoal for *y*
apply (*induct y*)
apply (*subst mset-neg-minus*)
apply *transfer'*
apply (*simp add*: *Diff-eq-empty-iff-mset mset-subset-eqI*)
done
done
done

lemma *mset-neg-zcount-nonneg*: *mset-neg M* = {#} ⟹ *0* ≤ *zcount M t*
by (*subst* (*asm*) *mset-neg-empty-iff*) *simp*

lemma *in-zmset-conv-pos-neg-disj*: *x* ∈#$_z$ *M* ⟷ *x* ∈# *mset-pos M* ∨ *x* ∈# *mset-neg M*
by (*metis count-mset-pos in-diff-zcount mem-zmset-of mset-pos-neg-partition nat-code*(*2*) *not-in-iff zcount-ne-zero-iff*)

lemma *in-zmset-notin-mset-pos*[*simp*]: *x* ∈#$_z$ *M* ⟹ *x* ∉# *mset-pos M* ⟹ *x* ∈# *mset-neg M*
by (*auto simp*: *in-zmset-conv-pos-neg-disj*)

lemma *in-zmset-notin-mset-neg*[*simp*]: *x* ∈#$_z$ *M* ⟹ *x* ∉# *mset-neg M* ⟹ *x* ∈# *mset-pos M*
by (*auto simp*: *in-zmset-conv-pos-neg-disj*)

lemma *in-mset-pos-in-zmset*: *x* ∈# *mset-pos M* ⟹ *x* ∈#$_z$ *M*
by (*auto intro*: *iffD2*[*OF in-zmset-conv-pos-neg-disj*])

lemma *in-mset-neg-in-zmset*: *x* ∈# *mset-neg M* ⟹ *x* ∈#$_z$ *M*
by (*auto intro*: *iffD2*[*OF in-zmset-conv-pos-neg-disj*])

lemma *set-zmset-eq-set-mset-union*: *set-zmset M* = *set-mset* (*mset-pos M*) ∪ *set-mset* (*mset-neg M*)

by (*auto dest*: *in-mset-pos-in-zmset in-mset-neg-in-zmset*)

lemma *member-mset-pos-iff-zcount*: $x \in\# \textit{mset-pos}\ M \longleftrightarrow 0 < \textit{zcount}\ M\ x$
using *not-in-iff pos-zcount-in-zmset* **by** *force*

lemma *member-mset-neg-iff-zcount*: $x \in\# \textit{mset-neg}\ M \longleftrightarrow \textit{zcount}\ M\ x < 0$
by (*metis member-mset-pos-iff-zcount mset-pos-uminus neg-le-0-iff-le not-le zcount-uminus*)

lemma *mset-pos-mset-neg-disjoint*[*simp*]: *set-mset* (*mset-pos* Δ) $\cap$ *set-mset* (*mset-neg* Δ) = {}
by (*auto simp*: *member-mset-pos-iff-zcount member-mset-neg-iff-zcount*)

lemma *zcount-sum*: *zcount* $(\sum M{\in}MM.\ f\ M)\ t = (\sum M{\in}MM.\ \textit{zcount}\ (f\ M)\ t)$
by (*induct MM rule*: *infinite-finite-induct*) *auto*

lemma *zcount-filter-invariant*: *zcount* $\{\#\ t'{\in}\#_z M.\ t'{=}t\ \#\}\ t = \textit{zcount}\ M\ t$
by *auto*

lemma *in-filter-zmset-in-zmset*[*simp*]: $x \in\#_z$ *filter-zmset* $P\ M \Longrightarrow x \in\#_z M$
by (*metis count-filter-zmset zcount-ne-zero-iff*)

lemma *pos-filter-zmset-pos-zmset*[*simp*]: $0 <$ *zcount* (*filter-zmset* $P\ M$) $x \Longrightarrow 0 <$ *zcount* $M\ x$
by (*metis* (*full-types*) *count-filter-zmset less-irrefl*)

lemma *neg-filter-zmset-neg-zmset*[*simp*]: $0 >$ *zcount* (*filter-zmset* $P\ M$) $x \Longrightarrow 0 >$ *zcount* $M\ x$
by (*metis* (*full-types*) *count-filter-zmset less-irrefl*)

lift-definition *update-zmultiset* :: $'t$ *zmultiset* $\Rightarrow$ $'t \Rightarrow$ *int* $\Rightarrow$ $'t$ *zmultiset* **is**
$\lambda(A,B)\ T\ D.$(*if* $D{>}0$ *then* (A + *replicate-mset* (*nat* D) T, B)
else (A,B + *replicate-mset* (*nat* $(-D)$) T))
by (*auto simp*: *equiv-zmset-def if-split*)

lemma *zcount-update-zmultiset*: *zcount* (*update-zmultiset* $M\ t\ n$) $t' =$ *zcount* $M\ t'$ + (*if* $t = t'$ *then* n *else* 0)
by *transfer auto*

lemma (**in** *order*) *order-zmset-exists-foundation*:
fixes $t :: 'a$
assumes $0 <$ *zcount* $M\ t$
shows $\exists s.\ s \le t \wedge 0 <$ *zcount* $M\ s \wedge (\forall u.\ 0 <$ *zcount* $M\ u \longrightarrow \neg\ u < s)$
using *assms*
proof −
let *?M* = $\{t.\ 0 <$ *zcount* $M\ t\}$
from *assms* **have** $t \in$ *?M*
by *simp*
then have $\exists s{\in}?M.\ s \le t \wedge (\forall u{\in}?M.\ \neg\ u < s)$

by − (*drule order-finite-set-exists-foundation*[*rotated 1*], *auto*)
then show *?thesis* **by** *auto*
qed

lemma (**in** *order*) *order-zmset-exists-foundation'*:
fixes $t :: 'a$
assumes $0 < zcount\ M\ t$
shows $\exists s.\ s \leq t \wedge 0 < zcount\ M\ s \wedge (\forall u{<}s.\ zcount\ M\ u \leq 0)$
using *assms order-zmset-exists-foundation*
by (*meson le-less-linear*)

lemma (**in** *order*) *order-zmset-exists-foundation-neg*:
fixes $t :: 'a$
assumes $zcount\ M\ t < 0$
shows $\exists s.\ s \leq t \wedge zcount\ M\ s < 0 \wedge (\forall u.\ zcount\ M\ u < 0 \longrightarrow \neg\ u < s)$
using *assms*
proof −
let $?M = \{t.\ zcount\ M\ t < 0\}$
from *assms* **have** $t \in ?M$
by *simp*
then have $\exists s{\in}?M.\ s \leq t \wedge (\forall u{\in}?M.\ \neg\ u < s)$
by − (*drule order-finite-set-exists-foundation*[*rotated 1*], *auto*)
then show *?thesis* **by** *auto*
qed

lemma (**in** *order*) *order-zmset-exists-foundation-neg'*:
fixes $t :: 'a$
assumes $zcount\ M\ t < 0$
shows $\exists s.\ s \leq t \wedge zcount\ M\ s < 0 \wedge (\forall u{<}s.\ 0 \leq zcount\ M\ u)$
using *assms order-zmset-exists-foundation-neg*
by (*meson le-less-linear*)

lemma (**in** *order*) *elem-order-zmset-exists-foundation*:
fixes $x :: 'a$
assumes $x \in\#_z\ M$
shows $\exists s{\in}\#_z M.\ s \leq x \wedge (\forall u{\in}\#_z M.\ \neg\ u < s)$
by (*rule order-finite-set-exists-foundation*[*OF finite-set-zmset*, *OF assms*(*1*)])

2.5.1 Image of a Signed Multiset

lift-definition *image-zmset* :: $('a \Rightarrow 'b) \Rightarrow 'a\ zmultiset \Rightarrow 'b\ zmultiset$ **is**
$\lambda f\ (M, N).\ (image\text{-}mset\ f\ M, image\text{-}mset\ f\ N)$
by (*auto simp*: *equiv-zmset-def simp flip*: *image-mset-union*)

syntax (*ASCII*)
-comprehension-zmset :: $'a \Rightarrow 'b \Rightarrow 'b\ zmultiset \Rightarrow 'a\ zmultiset$ (({#-/. - :#*z*
-#}))
syntax
-comprehension-zmset :: $'a \Rightarrow 'b \Rightarrow 'b\ zmultiset \Rightarrow 'a\ zmultiset$ (({#-/. - $\in\#_z$

-#}))
translations
{#e. x ∈#$_z$ M#} ⇌ *CONST image-zmset* (λx. e) M

lemma *image-zmset-empty*[*simp*]: *image-zmset f* {#}$_z$ = {#}$_z$
by *transfer* (*auto simp*: *equiv-zmset-def*)

lemma *image-zmset-single*[*simp*]: *image-zmset f* {#x#}$_z$ = {#f x#}$_z$
by *transfer* (*simp add*: *equiv-zmset-def*)

lemma *image-zmset-union*[*simp*]: *image-zmset f* (M + N) = *image-zmset f M* + *image-zmset f N*
by *transfer* (*auto simp*: *equiv-zmset-def*)

lemma *image-zmset-Diff*[*simp*]: *image-zmset f* (A − B) = *image-zmset f A* − *image-zmset f B*
proof −
have *image-zmset f* (A − B + B) = *image-zmset f* (A − B) + *image-zmset f B*
using *image-zmset-union* **by** *blast*
then show *?thesis* **by** *simp*
qed

lemma *mset-neg-image-zmset*: *mset-neg M* = {#} ⟹ *mset-neg* (*image-zmset f M*) = {#}
unfolding *multiset-eq-iff count-empty*
by *transfer* (*auto simp add*: *image-mset-subseteq-mono mset-subset-eqI mset-subset-eq-count*)

lemma *nonneg-zcount-image-zmset*[*simp*]: (⋀t. 0 ≤ *zcount M t*) ⟹ 0 ≤ *zcount* (*image-zmset f M*) t
by (*meson mset-neg-empty-iff mset-neg-image-zmset*)

lemma *image-zmset-add-zmset*[*simp*]: *image-zmset f* (*add-zmset t M*) = *add-zmset* (*f t*) (*image-zmset f M*)
by *transfer* (*auto simp*: *equiv-zmset-def*)

lemma *pos-zcount-image-zmset*[*simp*]: (⋀t. 0 ≤ *zcount M t*) ⟹ 0 < *zcount M t* ⟹ 0 < *zcount* (*image-zmset f M*) (*f t*)
apply *transfer*
subgoal for *M t f*
apply (*induct M*)
subgoal for *Mp Mn*
apply *simp*
apply (*metis count-diff count-image-mset-ge-count image-mset-Diff less-le-trans subseteq-mset-def zero-less-diff*)
done
done
done

lemma *set-zmset-transfer*[*transfer-rule*]:

(*rel-fun* (*pcr-zmultiset* (=)) (*rel-set* (=)))
(λ(*Mp*, *Mn*). *set-mset Mp* ∪ *set-mset Mn* − {*x*. *count Mp x* = *count Mn x*})
set-zmset
by (*auto simp*: *rel-fun-def pcr-zmultiset-def cr-zmultiset-def*
rel-set-eq multiset.rel-eq set-zmset-def zcount.abs-eq count-eq-zero-iff[*symmetric*]
simp del: *zcount-ne-zero-iff*)

lemma *zcount-image-zmset*:
zcount (*image-zmset f M*) *x* = (∑ *y* ∈ *f* −‘ {*x*} ∩ *set-zmset M*. *zcount M y*)
apply (*transfer fixing*: *f x*)
subgoal for *M*
apply (*cases M*; *clarify*)
subgoal for *Mp Mn*
unfolding *count-image-mset int-sum*
proof −
have (∑ *x*∈*f* −‘ {*x*} ∩ *set-mset Mp*. *int* (*count Mp x*)) =
(∑ *x*∈*f* −‘ {*x*} ∩ (*set-mset Mp* ∪ *set-mset Mn*). *int* (*count Mp x*)) (**is** *?S1* = -)
by (*subst sum.same-carrier*[**where** *C*=*f* −‘ {*x*} ∩ (*set-mset Mp* ∪ *set-mset Mn*)])
(*auto simp*: *count-eq-zero-iff*)
moreover
have (∑ *x*∈*f* −‘ {*x*} ∩ *set-mset Mn*. *int* (*count Mn x*)) =
(∑ *x*∈*f* −‘ {*x*} ∩ (*set-mset Mp* ∪ *set-mset Mn*). *int* (*count Mn x*))(**is** *?S2* = -)
by (*subst sum.same-carrier*[**where** *C*=*f* −‘ {*x*} ∩ (*set-mset Mp* ∪ *set-mset Mn*)])
(*auto simp*: *count-eq-zero-iff*)
moreover
have (∑ *x*∈*f* −‘ {*x*} ∩ (*set-mset Mp* ∪ *set-mset Mn* − {*x*. *count Mp x* = *count Mn x*}). *int* (*count Mp x*) − *int* (*count Mn x*))
= (∑ *x*∈*f* −‘ {*x*} ∩ (*set-mset Mp* ∪ *set-mset Mn*). *int* (*count Mp x*) − *int* (*count Mn x*))
(**is** *?S* = -)
by (*subst sum.same-carrier*[**where** *C*=*f* −‘ {*x*} ∩ (*set-mset Mp* ∪ *set-mset Mn*)]) *auto*
ultimately show *?S1* − *?S2* = *?S*
by (*auto simp*: *sum-subtractf*)
qed
done
done

lemma *zmset-empty-image-zmset-empty*: (⋀*t*. *zcount M t* = *0*) ⟹ *zcount* (*image-zmset f M*) *t* = *0*
by (*auto simp*: *zcount-image-zmset*)

lemma *in-image-zmset-in-zmset*: *t* ∈#$_z$ *image-zmset f M* ⟹ ∃ *t*. *t* ∈#$_z$ *M*
by (*rule ccontr*) *simp*

lemma *zcount-image-zmset-zero*: $(\bigwedge m.\ m \in\#_z\ M \Longrightarrow f\ m \neq x) \Longrightarrow x \notin\#_z$ *image-zmset f M*
unfolding *set-zmset-def*
by (*simp add*: *zcount-image-zmset*) (*metis Int-emptyI sum.empty vimage-singleton-eq*)

lemma *image-zmset-pre*: $t \in\#_z$ *image-zmset f M* $\Longrightarrow \exists m.\ m \in\#_z\ M \wedge f\ m = t$
proof (*rule ccontr*)
assume *t*: $t \in\#_z$ *image-zmset f M*
assume $\nexists m.\ m \in\#_z\ M \wedge f\ m = t$
then have $m \in\#_z\ M \Longrightarrow \neg\ f\ m = t$ **for** *m*
by *blast*
then have *zcount* (*image-zmset f M*) *t* = *0*
by (*meson t zcount-image-zmset-zero*)
with *t* **show** *False*
by (*meson zcount-ne-zero-iff*)
qed

lemma *pos-image-zmset-obtain-pre*:
$(\bigwedge t.\ 0 \leq$ *zcount M t*$) \Longrightarrow 0 <$ *zcount* (*image-zmset f M*) *t* $\Longrightarrow \exists m.\ 0 <$ *zcount M m* $\wedge$ *f m* = *t*
proof −
assume *nonneg*: *0* $\leq$ *zcount M t* **for** *t*
assume *0* $<$ *zcount* (*image-zmset f M*) *t*
then have *t* $\in\#_z$ *image-zmset f M*
by (*simp add*: *pos-zcount-in-zmset*)
then obtain *x* **where** *x*: $x \in\#_z$ *M f x* = *t*
by (*auto dest*: *image-zmset-pre*)
with *nonneg* **have** *0* $<$ *zcount M x*
by (*meson zmset-elem-nonneg*)
with *x* **show** *?thesis*
by *auto*
qed

2.6 Streams

definition *relates* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a$ *stream* $\Rightarrow$ *bool* **where**
relates φ *s* = φ (*shd s*) (*shd* (*stl s*))

lemma *relatesD*[*dest*]: *relates P s* $\Longrightarrow$ *P* (*shd s*) (*shd* (*stl s*))
unfolding *relates-def* **by** *simp*

lemma *alw-relatesD*[*dest*]: *alw* (*relates P*) *s* $\Longrightarrow$ *P* (*shd s*) (*shd* (*stl s*))
by *auto*

lemma *relatesI*[*intro*]: *P* (*shd s*) (*shd* (*stl s*)) $\Longrightarrow$ *relates P s*
by (*auto simp*: *relates-def*)

lemma *alw-holds-smap-conv-comp*: *alw* (*holds P*) (*smap f s*) = *alw* ($\lambda s.$ (*P o f*) (*shd s*)) *s*

apply (*rule iffI*)
apply (*coinduction arbitrary*: *s*)
apply *auto* []
apply (*coinduction arbitrary*: *s*)
apply *auto*
done

lemma *alw-relates*: *alw* (*relates P*) *s* ⟷ *P* (*shd s*) (*shd* (*stl s*)) ∧ *alw* (*relates P*) (*stl s*)
apply (*rule iffI*)
apply (*auto simp*: *relates-def dest*: *alwD*) []
apply (*coinduction arbitrary*: *s*)
apply (*auto simp*: *relates-def*)
done

2.7 Notation

no-notation *AND* (**infix** *aand 60*)
no-notation *OR* (**infix** *or 60*)
no-notation *IMPL* (**infix** *imp 60*)

notation *AND* (**infixr** *aand 70*)
notation *OR* (**infixr** *or 65*)
notation *IMPL* (**infixr** *imp 60*)

lifting-update *multiset.lifting*
lifting-forget *multiset.lifting*

3 Clocks Protocol

type-synonym *'t count-vec* = *'t multiset*
type-synonym *'t delta-vec* = *'t zmultiset*

definition *vacant-upto* :: *'t delta-vec* ⇒ *'t* :: *order* ⇒ *bool* **where**
vacant-upto a t = (∀ *s*. *s* ≤ *t* ⟶ *zcount a s* = *0*)

abbreviation *nonpos-upto* :: *'t delta-vec* ⇒ *'t* :: *order* ⇒ *bool* **where**
nonpos-upto a t ≡ ∀ *s*. *s* ≤ *t* ⟶ *zcount a s* ≤ *0*

definition *supported-strong* :: *'t delta-vec* ⇒ *'t* :: *order* ⇒ *bool* **where**
supported-strong a t = (∃ *s*. *s* < *t* ∧ *zcount a s* < *0* ∧ *nonpos-upto a s*)

definition *supported* :: *'t delta-vec* ⇒ *'t* :: *order* ⇒ *bool* **where**
supported a t = (∃ *s*. *s* < *t* ∧ *zcount a s* < *0*)

definition *upright* :: *'t* :: *order delta-vec* ⇒ *bool* **where**
upright a = (∀ *t*. *zcount a t* > *0* ⟶ *supported a t*)

lemma *upright-alt*: *upright a* $\longleftrightarrow$ $(\forall\, t.$ *zcount a t* $> 0 \longrightarrow$ *supported-strong a t*)
unfolding *upright-def supported-def supported-strong-def*
by (*rule iffI*) (*meson order.strict-trans2 order.strict-trans1 order-zmset-exists-foundation'*)+

definition *beta-upright* :: $'t$ :: *order delta-vec* $\Rightarrow$ $'t$ :: *order delta-vec* $\Rightarrow$ *bool* **where**
beta-upright va vb = $(\forall\, t.$ *zcount va t* $> 0 \longrightarrow (\exists\, s.\ s < t \wedge$ (*zcount va s* $< 0 \vee$ *zcount vb s* < 0)))

lemma *beta-upright-alt*:
beta-upright va vb = $(\forall\, t.$ *zcount va t* $> 0 \longrightarrow (\exists\, s.\ s < t \wedge$ (*zcount va s* $< 0 \vee$ *zcount vb s* < 0) $\wedge$ *nonpos-upto va s*))
unfolding *beta-upright-def*
apply (*rule iffI*)
apply *clarsimp*
apply (*drule order-zmset-exists-foundation*)
apply (*metis le-less-linear less-le-trans order.strict-trans1*)
apply *blast*
done

record ($'p$, $'t$) *configuration* =
c-records :: $'t$ *delta-vec*
c-temp :: $'p \Rightarrow$ $'t$ *delta-vec*
c-msg :: $'p \Rightarrow$ $'p \Rightarrow$ $'t$ *delta-vec list*
c-glob :: $'p \Rightarrow$ $'t$ *delta-vec*

type-synonym ($'p$, $'t$) *computation* = ($'p$, $'t$) *configuration stream*

definition *init-config* :: ($'p$ :: *finite*, $'t$ :: *order*) *configuration* $\Rightarrow$ *bool* **where**
init-config c =
((∀ *p*. *c-temp c p* = $\{\#\}_z$) ∧
(∀ *p1 p2*. *c-msg c p1 p2* = []) ∧
(∀ *p*. *c-glob c p* = *c-records c*) ∧
(∀ *t*. 0 ≤ *zcount* (*c-records c*) *t*))

definition *next-performop'* :: ($'p$, $'t$ :: *order*) *configuration* $\Rightarrow$ ($'p$, $'t$) *configuration* $\Rightarrow$ $'p \Rightarrow$ $'t$ *count-vec* $\Rightarrow$ $'t$ *count-vec* $\Rightarrow$ *bool* **where**
next-performop' c0 c1 p c r =
(*let* Δ = *zmset-of r* − *zmset-of c in*
(∀ *t*. *int* (*count c t*) ≤ *zcount* (*c-records c0*) *t*)
∧ *upright* Δ
∧ *c1* = *c0*(|*c-records* := *c-records c0* + Δ,
c-temp := (*c-temp c0*)(*p* := *c-temp c0 p* + Δ)|))

abbreviation *next-performop* **where**
next-performop s ≡ (∃ *p* (*c* :: $'t$ :: *order count-vec*) (*r*::$'t$ *count-vec*). *next-performop'* (*shd s*) (*shd* (*stl s*)) *p c r*)

definition *next-sendupd′* **where**
next-sendupd′ c0 c1 p tt =
(*let* γ = {#*t* ∈#$_z$ *c-temp c0 p*. *t* ∈ *tt*#} *in*
$\gamma \neq 0$
∧ *upright* (*c-temp c0 p* − γ)
∧ *c1* = *c0*(|*c-msg* := (*c-msg c0*)(*p* := λ*q*. *c-msg c0 p q* @ [γ]),
c-temp := (*c-temp c0*)(*p* := *c-temp c0 p* − γ)|))

abbreviation *next-sendupd* **where**
next-sendupd s ≡ (∃ *p tt*. *next-sendupd′* (*shd s*) (*shd* (*stl s*)) *p tt*)

definition *next-recvupd′* **where**
next-recvupd′ c0 c1 p q =
(*c-msg c0 p q* ≠ []
∧ *c1* = *c0*(|*c-msg* := (*c-msg c0*)(*p* := (*c-msg c0 p*)(*q* := *tl* (*c-msg c0 p q*))),
c-glob := (*c-glob c0*)(*q* := *c-glob c0 q* + *hd* (*c-msg c0 p q*))|))

abbreviation *next-recvupd* **where**
next-recvupd s ≡ (∃ *p q*. *next-recvupd′* (*shd s*) (*shd* (*stl s*)) *p q*)

definition *next* **where**
next s = (*next-performop s* ∨ *next-sendupd s* ∨ *next-recvupd s* ∨ (*shd* (*stl s*) = *shd s*))

definition *spec* :: ('*p* :: *finite*, '*t* :: *order*) *computation* ⇒ *bool* **where**
spec s = (*holds init-config s* ∧ *alw next s*)

abbreviation *GlobVacantUpto* **where**
GlobVacantUpto c q t ≡ *vacant-upto* (*c-glob c q*) *t*

abbreviation *NrecVacantUpto* **where**
NrecVacantUpto c t ≡ *vacant-upto* (*c-records c*) *t*

definition *SafeGlobVacantUptoImpliesStickyNrec* :: ('*p* :: *finite*, '*t* :: *order*) *computation* ⇒ *bool* **where**
SafeGlobVacantUptoImpliesStickyNrec s =
(*let c* = *shd s in* ∀ *t q*. *GlobVacantUpto c q t* ⟶ *alw* (*holds* (λ*c*. *NrecVacantUpto c t*)) *s*)

definition *SafeStickyNrecVacantUpto* :: ('*p* :: *finite*, '*t* :: *order*) *computation* ⇒ *bool* **where**
SafeStickyNrecVacantUpto s =
(*let c* = *shd s in* ∀ *t*. *NrecVacantUpto c t* ⟶ *alw* (*holds* (λ*c*. *NrecVacantUpto c t*)) *s*)

definition *InvGlobVacantUptoImpliesNrec* :: ('*p* :: *finite*, '*t* :: *order*) *configuration*

```
⇒ bool where
  InvGlobVacantUptoImpliesNrec c =
     (∀ t q. vacant-upto (c-glob c q) t ⟶ vacant-upto (c-records c) t)

definition InvTempUpright where
  InvTempUpright c = (∀ p. upright (c-temp c p))

lemma init-InvTempUpright: init-config c ⟹ InvTempUpright c
  by (simp add: InvTempUpright-def init-config-def upright-def)

lemma upright-obtain-support:
  assumes upright a
    and zcount a t > 0
  obtains s where s < t zcount a s < 0 nonpos-upto a s
  using assms unfolding upright-alt supported-strong-def
  apply atomize-elim
  using order.strict-implies-order apply blast
  done

lemma upright-vec-add:
  assumes upright v1
    and   upright v2
  shows   upright (v1 + v2)
proof −
  let ?v0 = v1 + v2
  { fix t
    assume upr1: upright v1
    assume upr2: upright v2
    assume zcnt: 0 < zcount ?v0 t
    { fix va vb :: 'a zmultiset
      fix t
      assume upra: upright va
      assume uprb: upright vb
      assume zcnt: 0 < zcount va t
      from upra zcnt obtain x where x: x < t zcount va x < 0 nonpos-upto va x
        using upright-obtain-support by blast
      with uprb have supported-strong (va+vb) t
        apply (cases ∃ s. s ≤ x ∧ 0 < zcount vb s)
         apply (clarsimp simp: upright-alt supported-strong-def)
          apply (meson add-nonpos-neg less-imp-le order.strict-trans2 order.trans
add-nonpos-nonpos)
        apply simp
        apply (force simp: supported-strong-def intro!: exI[of - x])
        done
    }
   with upr1 upr2 zcnt have supported-strong ?v0 t unfolding supported-strong-def
     apply (cases 0 < zcount v1 t; cases 0 < zcount v2 t)
       apply auto [2]
      apply (subst (1 2) add.commute)
```

```
      apply auto
     done
  }
  with assms show ?thesis
    by (simp add: upright-alt)
qed

lemma next-InvTempUpright: holds InvTempUpright s ⟹ next s ⟹ nxt (holds
InvTempUpright) s
  unfolding next-def apply simp
  apply (elim disjE)
  subgoal
    unfolding InvTempUpright-def next-performop'-def
    by (auto simp: Let-def upright-vec-add)
  subgoal
    unfolding InvTempUpright-def next-sendupd'-def
    by (auto simp: Let-def upright-vec-add)
  subgoal
    unfolding InvTempUpright-def next-recvupd'-def
    by (auto simp: upright-vec-add)
  subgoal by simp
  done

lemma alw-InvTempUpright: spec s ⟹ alw (holds InvTempUpright) s
  apply (rule alw-invar)
   apply (simp add: spec-def init-InvTempUpright)
  apply (metis (no-types, lifting) alw-iff-sdrop next-InvTempUpright spec-def)
  done

definition IncomingInfo where
  IncomingInfo c k p q = (sum-list (drop k (c-msg c p q)) + c-temp c p)

definition InvIncomingInfoUpright where
  InvIncomingInfoUpright c = (∀ k p q. upright (IncomingInfo c k p q))

lemma upright-0: upright 0
  by (simp add: upright-def)

lemma init-InvIncomingInfoUpright: init-config c ⟹ InvIncomingInfoUpright c
  by (simp add: upright-0 upright-vec-add init-config-def InvIncomingInfoUpright-def
IncomingInfo-def)

lemma next-InvIncomingInfoUpright: holds InvIncomingInfoUpright s ⟹ next s
⟹ nxt (holds InvIncomingInfoUpright) s
  unfolding next-def
  apply simp
  apply (elim disjE)
  subgoal
   by (auto simp: add.assoc[symmetric] upright-vec-add next-performop'-def Let-def
```

InvIncomingInfoUpright-def IncomingInfo-def)
subgoal
unfolding *next-sendupd'-def Let-def InvIncomingInfoUpright-def Incoming-Info-def*
apply *clarsimp*
subgoal for *p tt k q*
apply (*cases k* ≤ *length* (*c-msg* (*shd s*) *p q*))
apply *auto*
done
done
subgoal
unfolding *next-recvupd'-def Let-def InvIncomingInfoUpright-def Incoming-Info-def*
apply (*clarsimp simp*: *drop-Suc*[*symmetric*])
done
subgoal
by *simp*
done

lemma *alw-InvIncomingInfoUpright*: *spec s* ⟹ *alw* (*holds InvIncomingInfoUpright*) *s*
by (*metis* (*mono-tags*, *lifting*) *alw-iff-sdrop alw-invar holds.elims*(*2*) *holds.elims*(*3*) *init-InvIncomingInfoUpright next-InvIncomingInfoUpright spec-def*)

definition *GlobalIncomingInfo* :: ('*p* :: *finite*, '*t*) *configuration* ⇒ *nat* ⇒ '*p* ⇒ '*p* ⇒ '*t delta-vec* **where**
GlobalIncomingInfo c k p q = (∑ *p'* ∈ *UNIV*. *IncomingInfo c* (*if p'* = *p then k else 0*) *p' q*)

abbreviation *GlobalIncomingInfoAt* **where**
GlobalIncomingInfoAt c q ≡ *GlobalIncomingInfo c 0 q q*

definition *InvGlobalRecordCount* **where**
InvGlobalRecordCount c = (∀ *q*. *c-records c* = *GlobalIncomingInfoAt c q* + *c-glob c q*)

lemma *init-InvGlobalRecordCount*: *holds init-config s* ⟹ *holds InvGlobalRecord-Count s*
by (*simp add*: *InvGlobalRecordCount-def init-config-def GlobalIncomingInfo-def IncomingInfo-def*)

lemma *if-eq-same*: (*if a* = *b then f b else f a*) = *f a*
by *auto*

lemma *next-InvGlobalRecordCount*: *holds InvGlobalRecordCount s* ⟹ *next s* ⟹ *nxt* (*holds InvGlobalRecordCount*) *s*
unfolding *InvGlobalRecordCount-def init-config-def GlobalIncomingInfo-def IncomingInfo-def next-def*

```
  apply (elim disjE)
  subgoal
    apply (clarsimp simp: next-performop'-def Let-def)
    subgoal for p c q r
      apply (simp add: sum.distrib)
      apply (subst sum-if-distrib-add)
        apply (simp-all add: add.assoc)
      done
    done
  subgoal
    apply (clarsimp simp: next-sendupd'-def Let-def)
    subgoal for p tt q
      apply (simp add: if-distrib[of λf. f -])
      apply (simp add: if-distrib[of sum-list])
      apply (subst sum-list-append)
      apply (simp add: sum.distrib)
      apply (subst sum-if-distrib-add)
        apply simp
       apply simp
      apply (subst diff-conv-add-uminus)
      apply (subst sum-if-distrib-add)
        apply (auto simp: sum-if-distrib-add)
      done
    done
  subgoal
    apply (clarsimp simp: next-recvupd'-def Let-def fun-upd-def)
    subgoal for p q q'
      apply (simp add: if-distrib[of λf. f -])
      apply safe
       apply (simp add: if-distrib[of sum-list])
       apply (subst sum-list-hd-tl)
        apply simp
       apply (subst add.commute)
       apply (simp add: sum.distrib)
       apply (subst sum-if-distrib-add)
         apply simp
        apply simp
       apply (simp add: add.assoc)
      apply (subst if-eq-same)
      apply simp
      done
    done
  subgoal
    by simp
  done


lemma alw-InvGlobalRecordCount: spec s ⟹ alw (holds InvGlobalRecordCount) s
  by (metis (no-types, lifting) alw-iff-sdrop alw-invar init-InvGlobalRecordCount
```

next-InvGlobalRecordCount spec-def)

definition *InvGlobalIncomingInfoUpright* **where**
InvGlobalIncomingInfoUpright c = (∀ *k p q*. *upright* (*GlobalIncomingInfo c k p q*))

lemma *upright-sum-upright*: *finite X* ⟹ ∀ *x*. *upright* (*A x*) ⟹ *upright* (∑ *x*∈*X*. *A x*)
by (*induct X rule*: *finite-induct*) (*auto simp*: *upright-0 upright-vec-add*)

lemma *InvIncomingInfoUpright-imp-InvGlobalIncomingInfoUpright*: *holds InvIncomingInfoUpright s* ⟹ *holds InvGlobalIncomingInfoUpright s*
by (*simp add*: *InvIncomingInfoUpright-def InvGlobalIncomingInfoUpright-def GlobalIncomingInfo-def upright-sum-upright*)

lemma *alw-InvGlobalIncomingInfoUpright*: *spec s* ⟹ *alw* (*holds InvGlobalIncomingInfoUpright*) *s*
using *InvIncomingInfoUpright-imp-InvGlobalIncomingInfoUpright alw-InvIncomingInfoUpright alw-mono* **by** *blast*

abbreviation *nrec-pos* **where**
nrec-pos c ≡ ∀ *t*. *zcount* (*c-records c*) *t* ≥ *0*

lemma *init-nrec-pos*: *holds init-config s* ⟹ *holds nrec-pos s*
by (*simp add*: *init-config-def*)

lemma *next-nrec-pos*: *holds nrec-pos s* ⟹ *next s* ⟹ *nxt* (*holds nrec-pos*) *s*
unfolding *next-def*
apply *simp*
apply *clarify*
apply (*elim disjE*)
subgoal for *t*
unfolding *next-performop′-def Let-def*
apply *clarify*
subgoal for *p c r*
apply (*simp add*: *add-diff-eq add.commute add-increasing*)
done
done
subgoal for *t*
by (*auto simp*: *next-sendupd′-def Let-def*)
subgoal for *t*
by (*auto simp*: *next-recvupd′-def Let-def*)
subgoal
by *simp*
done

lemma *alw-nrec-pos*: *spec s* ⟹ *alw* (*holds nrec-pos*) *s*
by (*metis* (*mono-tags*, *lifting*) *alw-iff-sdrop alw-invar init-nrec-pos next-nrec-pos*

spec-def)

lemma *next-performop-vacant*:
vacant-upto (*c-records* (*shd s*)) *t* ⟹ *next-performop s* ⟹ *vacant-upto* (*c-records* (*shd* (*stl s*))) *t*
unfolding *next-performop'-def Let-def vacant-upto-def*
apply *clarsimp*
subgoal for *p c u r*
apply (*clarsimp simp*: *upright-def supported-def*)
apply (*metis* (*no-types*, *opaque-lifting*) *gr-implies-not-zero of-nat-le-0-iff order.strict-implies-order order-trans zero-less-iff-neq-zero*)
done
done

lemma *next-sendupd-vacant*:
vacant-upto (*c-records* (*shd s*)) *t* ⟹ *next-sendupd s* ⟹ *vacant-upto* (*c-records* (*shd* (*stl s*))) *t*
by (*auto simp add*: *next-sendupd'-def Let-def*)

lemma *next-recvupd-vacant*:
vacant-upto (*c-records* (*shd s*)) *t* ⟹ *next-recvupd s* ⟹ *vacant-upto* (*c-records* (*shd* (*stl s*))) *t*
by (*auto simp add*: *next-recvupd'-def Let-def*)

lemma *spec-imp-SafeStickyNrecVacantUpto-aux*: *alw next s* ⟹ *alw SafeStickyNrecVacantUpto s*
apply (*coinduction arbitrary*: *s*)
subgoal for *s*
unfolding *spec-def next-def SafeStickyNrecVacantUpto-def Let-def*
apply (*rule exI*[*of* - *s*])
apply *safe*
subgoal for *t*
apply (*coinduction arbitrary*: *s rule*: *alw.coinduct*)
apply *clarsimp*
apply (*rule conjI*)
apply *blast*
apply (*erule alw.cases*)
apply *clarsimp*
apply (*elim disjE*)
apply (*simp-all add*: *next-performop-vacant next-sendupd-vacant next-recvupd-vacant*)
done
by *blast*
done

lemma *spec-imp-SafeStickyNrecVacantUpto*: *spec s* ⟹ *alw SafeStickyNrecVacantUpto s*
unfolding *spec-def*
by (*blast intro*: *spec-imp-SafeStickyNrecVacantUpto-aux*)

lemma *invs-imp-InvGlobVacantUptoImpliesNrec*:
assumes *holds InvGlobalIncomingInfoUpright s*
assumes *holds InvGlobalRecordCount s*
assumes *holds nrec-pos s*
shows *holds InvGlobVacantUptoImpliesNrec s*
using *assms* **unfolding** *InvGlobVacantUptoImpliesNrec-def*
apply *simp*
apply *clarify*
apply (*rule ccontr*)
apply (*simp add*: *vacant-upto-def*)
apply *clarify*
subgoal for *t q u*
proof −
assume *globvut*: ∀ *sa*≤*t*. *zcount* (*c-glob* (*shd s*) *q*) *sa* = *0*
assume *uleqt*: *u* ≤ *t*
assume *u* ∈#$_z$ *c-records* (*shd s*)
with *assms*(*3*) **have** *0* < *zcount* (*c-records* (*shd s*)) *u*
by (*simp add*: *order.not-eq-order-implies-strict*)
with *assms*(*2*) *globvut uleqt* **have** ∗: *0* < *zcount* (*GlobalIncomingInfoAt* (*shd s*) *q*) *u*
unfolding *InvGlobalRecordCount-def*
by (*auto dest*: *spec*[*of* - *q*])
from *assms*(*1*)[*unfolded InvGlobalIncomingInfoUpright-def*] **have** *upright* (*GlobalIncomingInfoAt* (*shd s*) *q*)
by *simp*
with ∗ **obtain** *v* **where** ∗∗: *v* ≤ *u zcount* (*GlobalIncomingInfoAt* (*shd s*) *q*) *v* < *0*
by (*meson order.strict-iff-order upright-def supported-def*)
with *assms*(*2*) **have** *zcount* (*c-records* (*shd s*)) *v* < *0*
by (*metis* (*no-types*, *opaque-lifting*) *InvGlobalRecordCount-def add.right-neutral order.trans globvut holds.elims*(*2*) *uleqt zcount-union*)
with *assms*(*3*) **show** *False*
using *atLeastatMost-empty* **by** *auto*
qed
done

lemma *spec-imp-inv1*: *spec s* ⟹ *alw* (*holds InvGlobVacantUptoImpliesNrec*) *s*
by (*metis* (*mono-tags*, *lifting*) *alw-iff-sdrop invs-imp-InvGlobVacantUptoImpliesNrec alw-InvGlobalIncomingInfoUpright alw-InvGlobalRecordCount alw-nrec-pos*)

lemma *safe2-inv1-imp-safe*: *SafeStickyNrecVacantUpto s* ⟹ *holds InvGlobVacantUptoImpliesNrec s* ⟹ *SafeGlobVacantUptoImpliesStickyNrec s*
by (*simp add*: *InvGlobVacantUptoImpliesNrec-def SafeStickyNrecVacantUpto-def SafeGlobVacantUptoImpliesStickyNrec-def*)

lemma *spec-imp-safe*: *spec s* ⟹ *alw SafeGlobVacantUptoImpliesStickyNrec s*
by (*meson alw-iff-sdrop safe2-inv1-imp-safe spec-imp-SafeStickyNrecVacantUpto spec-imp-inv1*)

lemma *beta-upright-0*: *beta-upright 0 vb*
unfolding *beta-upright-def*
by *auto*

definition *PositiveImplies* **where**
PositiveImplies v w = (∀ *t*. *zcount v t* > *0* ⟶ *zcount w t* > *0*)

lemma *betaupright-PositiveImplies*: *upright* (*va* + *vb*) ⟹ *PositiveImplies va* (*va* + *vb*) ⟹ *beta-upright va vb*
unfolding *beta-upright-def PositiveImplies-def*
apply *clarify*
subgoal for *t*
apply (*erule upright-obtain-support*[*of* - *t*])
apply *simp*
subgoal for *s*
apply (*rule exI*[*of* - *s*])
apply *simp*
apply (*simp add*: *add-less-zeroD*)
done
done
done

lemma *betaupright-obtain-support*:
assumes *beta-upright va vb*
zcount va t > *0*
obtains *s* **where** *s* < *t zcount va s* < *0* ∨ *zcount vb s* < *0 nonpos-upto va s*
using *assms* **by** (*auto simp*: *beta-upright-alt*)

lemma *betaupright-upright-vut*:
assumes *beta-upright va vb*
and *upright vb*
and *vacant-upto* (*va* + *vb*) *t*
shows *vacant-upto va t*
proof −
{ fix *s*
assume *s*: *s* ≤ *t zcount va s* > *0*
with *assms* **obtain** *x* **where** *x*: *x* < *s zcount va x* < *0* ∨ *zcount vb x* < *0 nonpos-upto va x*
using *betaupright-obtain-support* **by** *blast*
then have *False*
proof (*cases zcount va x* < *0*)
case *True*
with *assms*(*2*,*3*) *s x*(*1*,*3*) **show** *?thesis*
unfolding *vacant-upto-def*
apply *clarsimp*

```
      apply (erule upright-obtain-support[of vb x])
       apply (metis add-less-same-cancel2 order.trans order.strict-implies-order)
        apply (metis add-less-same-cancel1 add-neg-neg order.order-iff-strict order.trans less-irrefl)
      done
   next
     case False
     with assms s x have x ≤ t zcount va x > 0
       apply −
       apply simp
      apply (metis (no-types, opaque-lifting) add.left-neutral order.order-iff-strict order.trans vacant-upto-def zcount-union)
      done
     with assms(2,3) s x show ?thesis
      by force
   qed
 }
 note r = this
 from assms(2,3) show ?thesis
   unfolding vacant-upto-def
   apply clarsimp
  apply (metis (no-types, opaque-lifting) r add-cancel-right-left order.order-iff-strict order.trans le-less-linear less-add-same-cancel2 upright-obtain-support)
   done
qed

lemma beta-upright-add:
 assumes upright vb
   and     upright vc
   and     beta-upright va vb
 shows   beta-upright va (vb + vc)
proof −
 { fix t
   assume 0 < zcount va t
   assume assm: ¬(∃ s<t. (zcount va s < 0 ∨ zcount vb s + zcount vc s < 0) ∧ ¬(∃ u ≤ s. zcount va u > 0))
   from ‹0 < zcount va t› assms(3) obtain x where x: x < t ∧ (zcount va x < 0 ∨ zcount vb x < 0) ∧ nonpos-upto va x
     using betaupright-obtain-support by blast
   then have ¬ zcount va x < 0
     using assm by force
   with x have zcount vb x < 0
     by blast
   from assm x have ¬ zcount vb x + zcount vc x < 0
     using not-le by blast
   with ‹zcount vb x < 0› have zcount vc x > 0
     by clarsimp
   with assms(2) obtain y where y: y < x ∧ zcount vc y < 0 ∧ nonpos-upto vc y
```

```
    using upright-obtain-support by blast
  with x have y < t
    using order.strict-trans by blast
  from assm x y have ¬ zcount vb y + zcount vc y < 0
    by (metis order.strict-implies-order order.strict-trans1 not-less)
  with y have zcount vb y > 0
    by linarith
  with assms(1) obtain z where z: z < y ∧ zcount vb z < 0
    by (auto simp: upright-def supported-def)
  with ‹y < t› have z < t
    using order.strict-trans by blast
  with x y z have ¬ zcount vb z + zcount vc z < 0
    by (metis assm less-imp-le not-less order.strict-trans order.strict-trans1)
  with z have zcount vc z > 0
    by linarith
  with y z have False
    using order.strict-implies-order not-less by blast
 }
 then show ?thesis
  using beta-upright-def zcount-union by fastforce
qed

definition InfoAt where
 InfoAt c k p q = (if 0 ≤ k ∧ k < length (c-msg c p q) then (c-msg c p q) ! k else 0)

definition InvInfoAtBetaUpright where
 InvInfoAtBetaUpright c = (∀ k p q. beta-upright (InfoAt c k p q) (IncomingInfo c (k+1) p q))

lemma init-InvInfoAtBetaUpright: init-config c ⟹ InvInfoAtBetaUpright c
  unfolding init-config-def InvInfoAtBetaUpright-def beta-upright-def Incoming-Info-def InfoAt-def
 by simp

lemma next-inv[consumes 1, case-names next-performop next-sendupd next-recvupd stutter]:
 assumes next s
  and    next-performop s ⟹ P
  and    next-sendupd s ⟹ P
  and    next-recvupd s ⟹ P
  and    shd (stl s) = shd s ⟹ P
 shows  P
 using assms unfolding next-def by blast


lemma next-InvInfoAtBetaUpright:
 assumes a1: next s
  and    a2: InvInfoAtBetaUpright (shd s)
```

```
    and      a3: InvIncomingInfoUpright (shd s)
    and      a4: InvTempUpright (shd s)
  shows   InvInfoAtBetaUpright (shd (stl s))
  using assms
proof (cases rule: next-inv)
  case next-performop
  then show ?thesis
    unfolding next-performop'-def Let-def InvInfoAtBetaUpright-def
    apply clarify
    subgoal for p c r k' p' q'
    proof (cases p=p')
      let ?Δ = zmset-of r − zmset-of c
      assume upright-Δ: upright ?Δ
      assume conf: shd (stl s) = shd s(|c-records := c-records (shd s) + (zmset-of r
− zmset-of c),
        c-temp := (c-temp (shd s))(p := c-temp (shd s) p + (zmset-of r − zmset-of
c))|)
      case True
      then have iid: IncomingInfo (shd (stl s)) (k'+1) p' q' = IncomingInfo (shd
s) (k'+1) p' q' + ?Δ
        by (simp add: IncomingInfo-def conf)
      from a2 have bu: beta-upright (InfoAt (shd s) k' p' q') (IncomingInfo (shd s)
(k'+1) p' q')
        using InvInfoAtBetaUpright-def by fastforce
      show ?thesis
        unfolding iid
        apply (rule beta-upright-add)
          apply (meson InvIncomingInfoUpright-def a3)
         apply (rule upright-Δ)
        using bu unfolding conf InfoAt-def
        apply auto
        done
    next
      let ?Δ = zmset-of r − zmset-of c
      assume conf: shd (stl s) = shd s(|c-records := c-records (shd s) + (zmset-of r
− zmset-of c),
        c-temp := (c-temp (shd s))(p := c-temp (shd s) p + (zmset-of r − zmset-of
c))|)
      from a2 have bu: beta-upright (InfoAt (shd s) k p q) (IncomingInfo (shd s)
(k + 1) p q) for k p q
        using InvInfoAtBetaUpright-def by fastforce
      case False
      then have ii: IncomingInfo (shd (stl s)) (k'+1) p' q' = IncomingInfo (shd s)
(k'+1) p' q'
        unfolding IncomingInfo-def by (simp add: conf)
      with bu[of k' p' q'] show ?thesis unfolding conf InfoAt-def
        by auto
    qed
    done
```

```
next
  case next-sendupd
  then show ?thesis
    unfolding next-sendupd'-def Let-def InvInfoAtBetaUpright-def
    apply clarify
    subgoal for p tt k' p' q'
    proof (cases p=p')
      let ?γ = {#t ∈#z c-temp (shd s) p. t ∈ tt#}
      assume conf: shd (stl s) = (shd s)(|c-msg := (c-msg (shd s))(p := λq. c-msg
(shd s) p q @ [?γ]),
              c-temp := (c-temp (shd s))(p := c-temp (shd s) p − ?γ)|)
      from a2 have buia: beta-upright (InfoAt (shd s) k' p' q') (IncomingInfo (shd
s) (k'+1) p' q')
        using InvInfoAtBetaUpright-def by force
      from a4 have tu: upright (c-temp (shd s) p)
        by (simp add: InvTempUpright-def)
      case True
      then show ?thesis
      proof (cases k' rule: linorder-cases[where y = length (c-msg (shd s) p' q')])
        case greater
        then have InfoAt (shd (stl s)) k' p' q' = 0
          by (auto simp: conf InfoAt-def)
        then show ?thesis
          by (simp add: beta-upright-0)
      next
        case equal
        with True conf have InfoAt (shd (stl s)) k' p' q' = ?γ
          by (simp add: InfoAt-def)
       then have pi: PositiveImplies (InfoAt (shd (stl s)) k' p' q') (c-temp (shd s)
p)
          by (simp add: PositiveImplies-def)
        from conf have c-temp (shd s) p = c-temp (shd (stl s)) p + ?γ
          by simp
        with equal True conf tu pi have butemp: beta-upright (InfoAt (shd (stl s))
k' p' q') (c-temp (shd (stl s)) p)
          apply −
          apply (rule betaupright-PositiveImplies)
           apply (auto simp add: InfoAt-def)
          done
       with True equal conf have IncomingInfo (shd (stl s)) (k'+1) p' q' = c-temp
(shd (stl s)) p
          by (simp add: IncomingInfo-def)
        with butemp show ?thesis
          by simp
      next
        case less
        with conf have unch-ia: InfoAt (shd (stl s)) k' p' q' = InfoAt (shd s) k' p'
q'
          by (auto simp: nth-append InfoAt-def)
```

from *conf less* **have** *IncomingInfo* (*shd* (*stl s*)) (*k'*+*1*) *p' q'* = *IncomingInfo* (*shd s*) (*k'*+*1*) *p' q'*
by (*auto simp*: *IncomingInfo-def*)
with *buia unch-ia* **show** *?thesis* **by** *simp*
qed
next
let *?γ* = {#*t* ∈#$_z$ *c-temp* (*shd s*) *p*. *t* ∈ *tt*#}
assume *conf*: *shd* (*stl s*) = (*shd s*)(|*c-msg* := (*c-msg* (*shd s*))(*p* := λ*q*. *c-msg* (*shd s*) *p q* @ [*?γ*]),
c-temp := (*c-temp* (*shd s*))(*p* := *c-temp* (*shd s*) *p* − *?γ*)|)
from *a2* **have** *buia*: *beta-upright* (*InfoAt* (*shd s*) *k' p' q'*) (*IncomingInfo* (*shd s*) (*k'*+*1*) *p' q'*)
using *InvInfoAtBetaUpright-def* **by** *force*
case *False*
with *conf* **have** *unchia*: *InfoAt* (*shd* (*stl s*)) *k' p' q'* = *InfoAt* (*shd s*) *k' p' q'*
by (*simp add*: *InfoAt-def*)
from *False conf* **have** *unchii*: *IncomingInfo* (*shd* (*stl s*)) (*k'*+*1*) *p' q'* = *IncomingInfo* (*shd s*) (*k'*+*1*) *p' q'*
by (*simp add*: *IncomingInfo-def*)
from *unchia unchii buia* **show** *?thesis*
by *simp*
qed
done
next
case *next-recvupd*
then show *?thesis*
unfolding *next-recvupd'-def Let-def InvInfoAtBetaUpright-def*
apply *clarify*
subgoal for *p q k' p' q'*
proof (*cases p* = *p'* ∧ *q* = *q'*)
assume *conf*: *shd* (*stl s*) = (*shd s*)(|*c-msg* := (*c-msg* (*shd s*))(*p* := (*c-msg* (*shd s*) *p*)(*q* := *tl* (*c-msg* (*shd s*) *p q*))),
c-glob := (*c-glob* (*shd s*))(*q* := *c-glob* (*shd s*) *q* + *hd* (*c-msg* (*shd s*) *p q*))|)
case *True*
with *conf* **have** *iisuc*: *IncomingInfo* (*shd* (*stl s*)) (*k'*+*1*) *p' q'* = *IncomingInfo* (*shd s*) (*k'*+*2*) *p' q'*
by (*simp add*: *drop-Suc IncomingInfo-def*)
with *True conf* **have** *iasuc*: *InfoAt* (*shd* (*stl s*)) *k' p' q'* = *InfoAt* (*shd s*) (*k'*+*1*) *p' q'*
by (*simp add*: *less-diff-conv nth-tl InfoAt-def*)
from *a2* **have** *beta-upright* (*InfoAt* (*shd s*) (*k'*+*1*) *p' q'*) (*IncomingInfo* (*shd s*) (*k'*+*2*) *p' q'*)
using *InvInfoAtBetaUpright-def* **by** *fastforce*
with *iisuc iasuc* **show** *?thesis*
by *simp*
next
assume *conf*: *shd* (*stl s*) = (*shd s*)(|*c-msg* := (*c-msg* (*shd s*))(*p* := (*c-msg* (*shd s*) *p*)(*q* := *tl* (*c-msg* (*shd s*) *p q*))),

```
                c-glob := (c-glob (shd s))(q := c-glob (shd s) q + hd (c-msg (shd s) p
q))⦘
      from a2 have buia: beta-upright (InfoAt (shd s) k' p' q') (IncomingInfo (shd
s) (k'+1) p' q')
        by (simp add: InvInfoAtBetaUpright-def)
      case False
     with conf have unchii: IncomingInfo (shd (stl s)) (k'+1) p' q' = IncomingInfo
(shd s) (k'+1) p' q'
        by (auto simp: IncomingInfo-def)
      from False conf have unchia: InfoAt (shd (stl s)) k' p' q' = InfoAt (shd s)
k' p' q'
        by (auto simp: InfoAt-def)
      from unchii unchia buia show ?thesis
        by simp
    qed
    done
qed simp

lemma alw-InvInfoAtBetaUpright-aux: alw (holds InvTempUpright) s ⟹ alw
(holds InvIncomingInfoUpright) s ⟹ holds InvInfoAtBetaUpright s ⟹ alw next
s ⟹ alw (holds InvInfoAtBetaUpright) s
 by (coinduction arbitrary: s rule: alw.coinduct) (auto intro!: next-InvInfoAtBetaUpright)

lemma alw-InvInfoAtBetaUpright: spec s ⟹ alw (holds InvInfoAtBetaUpright) s
 by (simp add: alw-InvTempUpright alw-InvIncomingInfoUpright alw-InvInfoAtBetaUpright-aux
init-InvInfoAtBetaUpright spec-def)

definition InvGlobalInfoAtBetaUpright where
 InvGlobalInfoAtBetaUpright c = (∀ k p q. beta-upright (InfoAt c k p q) (GlobalIncomingInfo
c (k+1) p q))

lemma finite-induct-select [consumes 1, case-names empty select]:
  assumes finite S
    and empty: P {}
    and select: ⋀T. finite T ⟹ T ⊂ S ⟹ P T ⟹ ∃ s∈S − T. P (insert s T)
  shows P S
proof −
  from assms(1) have P S ∧ finite S
    by (induct S rule: finite-induct-select) (auto intro: empty select)
  then show ?thesis by blast
qed

lemma predicate-sum-decompose:
  fixes f :: 'a ⇒ ('b :: ab-group-add)
  assumes finite X
    and     x∈X
    and     A (f x)
    and     ∀ Z. B (sum f Z)
    and     ⋀x Z. A (f x) ⟹ B (sum f Z) ⟹ A (f x + sum f Z)
```

```
  and     ⋀x Z. B (f x) ⟹ A (sum f Z) ⟹ A (f x + sum f Z)
 shows A (∑ x∈X. f x)
 using assms(1,2,3)
 apply (induct X rule: finite-induct-select)
  apply simp
 apply (simp only: sum.insert-remove)
 subgoal for T
  apply (cases x ∈ T; simp add: assms(3))
   apply (drule psubset-imp-ex-mem)
   apply clarsimp
  subgoal for z
    apply (rule bexI[of - z])
     apply (rule assms(6)[of z T])
      apply (rule assms(4)[THEN spec, of {z}, simplified])
     apply simp
    apply simp
    done
  apply clarsimp
  apply (drule bspec[of - - x])
   apply safe
   apply (rule assms(2))
  using assms(4,5) apply blast
  done
 done

lemma invs-imp-InvGlobalInfoAtBetaUpright:
 assumes holds InvInfoAtBetaUpright s
   and     holds InvGlobalIncomingInfoUpright s
   and     holds InvIncomingInfoUpright s
 shows   holds InvGlobalInfoAtBetaUpright s
proof −
 have uii: ∀ k p q. upright (IncomingInfo (shd s) k p q)
   by (rule assms(3)[unfolded InvIncomingInfoUpright-def, simplified])
 have ugii: ∀ k p q. upright (GlobalIncomingInfo (shd s) k p q)
   by (rule assms(2)[unfolded InvGlobalIncomingInfoUpright-def, simplified])
 have buia: ∀ k p q. beta-upright (InfoAt (shd s) k p q) (IncomingInfo (shd s) (Suc
k) p q)
   by (rule assms(1)[unfolded InvInfoAtBetaUpright-def, simplified])
 from uii ugii buia have ∀ k p q. beta-upright (InfoAt (shd s) k p q) (GlobalIncomingInfo
(shd s) (Suc k) p q)
   unfolding GlobalIncomingInfo-def
   apply −
   apply (rule allI)+
   subgoal for k p q
    apply (rule predicate-sum-decompose[of UNIV p λv. beta-upright (InfoAt (shd
s) k p q) v λp'. IncomingInfo (shd s) (if p' = p then Suc k else 0) p' q upright])
          apply simp
         apply simp
        apply simp
```

```
      apply (simp add: upright-sum-upright)
     subgoal for p' X
      apply (rule beta-upright-add)
       apply simp
      apply simp
     apply simp
     done
    subgoal for p' X
     apply (subst add.commute)
     apply (rule beta-upright-add)
      apply simp
      apply (simp add: upright-sum-upright)
     apply clarsimp
     apply simp
     done
   done
  done
 then show ?thesis
  by (simp add: InvGlobalInfoAtBetaUpright-def)
qed

lemma alw-InvGlobalInfoAtBetaUpright: spec s ⟹ alw (holds InvGlobalInfoAtBetaUpright) s
 by (meson alw-InvGlobalIncomingInfoUpright alw-InvIncomingInfoUpright alw-InvInfoAtBetaUpright alw-iff-sdrop invs-imp-InvGlobalInfoAtBetaUpright)

definition SafeStickyGlobVacantUpto :: ('p :: finite, 't :: order) computation ⇒ bool where
 SafeStickyGlobVacantUpto s = (∀ q t. GlobVacantUpto (shd s) q t ⟶ alw (holds (λc. GlobVacantUpto c q t)) s)

lemma gvut1:
 GlobVacantUpto (shd s) q t ⟹ next-performop s ⟹ GlobVacantUpto (shd (stl s)) q t
 by (auto simp add: next-performop'-def Let-def vacant-upto-def upright-def)

lemma gvut2:
 GlobVacantUpto (shd s) q t ⟹ next-sendupd s ⟹ GlobVacantUpto (shd (stl s)) q t
 by (auto simp add: next-sendupd'-def Let-def)

lemma gvut3:
 assumes
  gvu: GlobVacantUpto (shd s) q t and
  igvuin: InvGlobVacantUptoImpliesNrec (shd s) and
  igrc: InvGlobalRecordCount (shd s) and
  igiiu: InvGlobalIncomingInfoUpright (shd s) and
  igiabu: InvGlobalInfoAtBetaUpright (shd s) and
  next: next-recvupd s
```

```
  shows GlobVacantUpto (shd (stl s)) q t
proof −
  { fix p
    let ?GII0 = GlobalIncomingInfo (shd s) 0 p q
    let ?GII1 = GlobalIncomingInfo (shd s) 1 p q
    let ?κ = hd (c-msg (shd s) p q)
    from igiiu have uGII1: upright ?GII1
      unfolding InvGlobalIncomingInfoUpright-def by simp
    assume globk: c-glob (shd (stl s)) q = c-glob (shd s) q + ?κ
    assume nonempty: c-msg (shd s) p q ≠ []
    then have sumGIIsk: ?GII0 = ?GII1 + ?κ
      unfolding GlobalIncomingInfo-def IncomingInfo-def
      by (auto simp: sum.remove ac-simps neq-Nil-conv)
    from nonempty have IA0k: ?κ = InfoAt (shd s) 0 p q
      by (simp add: InfoAt-def hd-conv-nth)
    from igiabu nonempty have bukGII1: beta-upright ?κ ?GII1
    proof −
      note igiabu
      then have beta-upright (InfoAt (shd s) 0 p q) (GlobalIncomingInfo (shd s) 1
p q)
        by (simp add: InvGlobalInfoAtBetaUpright-def)
      with IA0k show ?thesis
        by simp
    qed
    from igvuin gvu have nvu: NrecVacantUpto (shd s) t
      unfolding InvGlobVacantUptoImpliesNrec-def by blast
    with igrc have c-records (shd s) = c-glob (shd s) q + ?GII0
     unfolding GlobalIncomingInfo-def IncomingInfo-def InvGlobalRecordCount-def
      by (simp add: add.commute)
    with gvu nvu have vuGII0: vacant-upto ?GII0 t
      by (simp add: vacant-upto-def)
    from bukGII1 uGII1 have vacant-upto ?κ t
       by (rule betaupright-upright-vut[of ?κ ?GII1]) (metis vuGII0 add.commute
sumGIIsk)
    with gvu have GlobVacantUpto (shd (stl s)) q t
      by (simp add: globk vacant-upto-def)
  }
  then show ?thesis
    using assms unfolding next-recvupd′-def
    by auto
qed

lemma spec-imp-SafeStickyGlobVacantUpto-aux:
  assumes
    alw (holds (λc. InvGlobVacantUptoImpliesNrec c)) s and
    alw (holds (λc. InvGlobalRecordCount c)) s and
    alw (holds (λc. InvGlobalIncomingInfoUpright c)) s and
    alw (holds (λc. InvGlobalInfoAtBetaUpright c)) s and
    alw next s
```

shows *alw SafeStickyGlobVacantUpto s*
using *assms* **apply** (*coinduction arbitrary*: *s*)
subgoal for *s*
unfolding *spec-def next-def SafeStickyGlobVacantUpto-def Let-def*
apply (*rule exI*[*of* - *s*])
apply *safe*
subgoal for *q t*
apply (*coinduction arbitrary*: *s rule*: *alw.coinduct*)
apply *clarsimp*
apply (*rule conjI*)
apply *blast*
proof −
fix *sb* :: (*'a*, *'b*) *configuration stream*
assume *a1*: *alw* (*holds InvGlobVacantUptoImpliesNrec*) *sb*
assume *a2*: *alw* (*holds InvGlobalRecordCount*) *sb*
assume *a3*: *alw* (*holds InvGlobalIncomingInfoUpright*) *sb*
assume *a4*: *alw* (*holds InvGlobalInfoAtBetaUpright*) *sb*
assume *a5*: *alw* (*λs. next-performop s ∨ next-sendupd s ∨ next-recvupd s ∨ shd* (*stl s*) = *shd s*) *sb*
assume *a6*: *GlobVacantUpto* (*shd sb*) *q t*
have *next-performop sb ∨ next-sendupd sb ∨ next-recvupd sb ∨ shd* (*stl sb*) = *shd sb*
using *a5* **by** *blast*
then have *GlobVacantUpto* (*shd* (*stl sb*)) *q t*
using *a6 a4 a3 a2 a1* **by** (*metis* (*no-types*) *alwD gvut1 gvut2 gvut3 holds.elims*(*2*))
then show *alw* (*holds InvGlobalRecordCount*) (*stl sb*) ∧ *alw* (*holds InvGlobalIncomingInfoUpright*) (*stl sb*) ∧ *alw* (*holds InvGlobalInfoAtBetaUpright*) (*stl sb*) ∧ *alw* (*λs. next-performop s ∨ next-sendupd s ∨ next-recvupd s ∨ shd* (*stl s*) = *shd s*) (*stl sb*) ∧ *GlobVacantUpto* (*shd* (*stl sb*)) *q t*
using *a5 a4 a3 a2* **by** *blast*
qed
apply *blast*
done
done

lemma *spec-imp-SafeStickyGlobVacantUpto*: *spec s* ⟹ *alw SafeStickyGlobVacantUpto s*
apply (*rule spec-imp-SafeStickyGlobVacantUpto-aux*)
apply (*simp add*: *spec-imp-inv1*)
apply (*simp add*: *alw-InvGlobalRecordCount*)
apply (*simp add*: *alw-InvGlobalIncomingInfoUpright*)
apply (*simp add*: *alw-InvGlobalInfoAtBetaUpright*)
apply (*simp add*: *spec-def*)
done

definition *SafeGlobMono* **where**
SafeGlobMono c0 c1 = (∀ *p t. GlobVacantUpto c0 p t* ⟶ *GlobVacantUpto c1 p t*)

lemma *alw-SafeGlobMono*: *spec s* $\Longrightarrow$ *alw* (*relates SafeGlobMono*) *s*
apply (*drule spec-imp-SafeStickyGlobVacantUpto*)
apply (*erule alw-mono*)
apply (*fastforce simp*: *SafeStickyGlobVacantUpto-def SafeGlobMono-def relates-def*)
done

4 Exchange Protocol

4.1 Specification

record ('*p*, '*t*) *configuration* =
c-temp :: '*p* $\Rightarrow$ '*t zmultiset*
c-msg :: '*p* $\Rightarrow$ '*p* $\Rightarrow$ '*t zmultiset list*
c-glob :: '*p* $\Rightarrow$ '*t zmultiset*
c-caps :: '*p* $\Rightarrow$ '*t zmultiset*
c-data-msg :: ('*p* $\times$ '*t*) *multiset*

Description of the configuration: *c-msg c p q* global, all progress messages currently in-flight from p to q *c-data-msg c* global, capabilities carried by in-flight data messages *c-temp c p* local, aggregated progress updates of worker p that haven't been sent yet *c-glob c p* local, worker p's conservative approximation of all capabilities in the system *c-caps c p* local, worker p's capabilities

global = state of the whole system to which no worker has access local = state that is kept locally by each worker and which it can access

type-synonym ('*p*, '*t*) *computation* = ('*p*, '*t*) *configuration stream*

context *order* **begin**

abbreviation *timestamps M* $\equiv$ {# *t*. (*x*,*t*) $\in\#_z$ *M* #}

definition *vacant-upto* :: '*a zmultiset* $\Rightarrow$ '*a* $\Rightarrow$ *bool* **where**
vacant-upto a t $\equiv$ ($\forall$ *s*. *s* $\leq$ *t* $\longrightarrow$ *zcount a s* = *0*)

definition *nonpos-upto* :: '*a zmultiset* $\Rightarrow$ '*a* $\Rightarrow$ *bool* **where**
nonpos-upto a t = ($\forall$ *s*. *s* $\leq$ *t* $\longrightarrow$ *zcount a s* $\leq$ *0*)

definition *supported* :: '*a zmultiset* $\Rightarrow$ '*a* $\Rightarrow$ *bool* **where**
supported a t $\equiv$ ($\exists$ *s*. *s* $<$ *t* $\wedge$ *zcount a s* $<$ *0*)

definition *supported-strong* :: '*a zmultiset* $\Rightarrow$ '*a* $\Rightarrow$ *bool* **where**
supported-strong a t $\equiv$ ($\exists$ *s*. *s* $<$ *t* $\wedge$ *zcount a s* $<$ *0* $\wedge$ *nonpos-upto a s*)

definition *justified* **where**
justified C M = ($\forall$ *t*. *0* $<$ *zcount M t* $\longrightarrow$ *supported M t* $\vee$ ($\exists$ *t'*$<$*t*. *0* $<$ *zcount C t'*) $\vee$ *zcount M t* $<$ *zcount C t*)

lemma *justified-alt*:
justified C M = (∀ *t*. *0* < *zcount M t* ⟶ *supported-strong M t* ∨ (∃ *t'*<*t*. *0* < *zcount C t'*) ∨ *zcount M t* < *zcount C t*)
unfolding *justified-def supported-def supported-strong-def*
apply (*rule iffI*)
apply *clarsimp*
apply (*drule order-zmset-exists-foundation'*)
apply *clarsimp*
subgoal for *t s*
apply (*drule spec*[*of* - *s*])
apply *safe*
apply (*meson le-less-trans less-le-trans nonpos-upto-def*)
using *order.strict-trans2* **apply** *blast*
using *order.order-iff-strict* **apply** *auto*
done
apply *blast*
done

definition *justified-with* **where**
justified-with C M N =
(∀ *t*. *0* < *zcount M t* ⟶
(∃ *s*<*t*. (*zcount M s* < *0* ∨ *zcount N s* < *0*)) ∨
(∃ *s*<*t*. *0* < *zcount C s*) ∨
zcount (*M*+*N*) *t* < *zcount C t*)

lemma *justified-with-alt*: *justified-with C M N* =
(∀ *t*. *0* < *zcount M t* ⟶
(∃ *s*<*t*. (*zcount M s* < *0* ∨ *zcount N s* < *0*) ∧ (∀ *s'*<*s*. *zcount M s'* ≤ *0*)) ∨
(∃ *s*<*t*. *0* < *zcount C s*) ∨
zcount (*M*+*N*) *t* < *zcount C t*)
unfolding *justified-with-def*
apply (*rule iffI*)
apply *clarsimp*
apply (*drule order-zmset-exists-foundation'*)
apply *clarsimp*
subgoal for *t s*
apply (*drule spec*[*of* - *s*])
apply *safe*
using *order.strict-trans order.strict-trans2* **apply** *blast+*
apply (*simp add*: *not-less not-le*)
apply (*smt* (*verit*, *best*) *antisym-conv2*)
done
apply *blast*
done

definition *PositiveImplies* **where**
PositiveImplies v w ≡ ∀ *t*. *zcount v t* > *0* ⟶ *zcount w t* > *0*

— A worker can mint capabilities greater or equal to any owned capability
definition *minting-self* **where**
minting-self C M = (∀ *t*∈#*M*. ∃ *t′*≤*t*. *0* < *zcount C t′*)

— Sending messages mints a capability at a strictly greater pointstamp
definition *minting-msg* **where**
minting-msg C M = (∀ (*p*,*t*)∈#*M*. ∃ *t′*<*t*. *0* < *zcount C t′*)

definition *records* **where**
records c = (∑ *p*∈*UNIV*. *c-caps c p*) + *timestamps* (*zmset-of* (*c-data-msg c*))

definition *InfoAt* **where**
InfoAt c k p q = (*if* *0* ≤ *k* ∧ *k* < *length* (*c-msg c p q*) *then* (*c-msg c p q*) ! *k* *else* {#}$_z$)

definition *IncomingInfo* :: ('*p*, '*a*) *configuration* ⇒ *nat* ⇒ '*p* ⇒ '*p* ⇒ '*a zmultiset* **where**
IncomingInfo c k p q ≡ *sum-list* (*drop k* (*c-msg c p q*)) + *c-temp c p*

definition *GlobalIncomingInfo* :: ('*p* :: *finite*, '*a*) *configuration* ⇒ *nat* ⇒ '*p* ⇒ '*p* ⇒ '*a zmultiset* **where**
GlobalIncomingInfo c k p q ≡ ∑ *p′* ∈ *UNIV*. *IncomingInfo c* (*if* *p′* = *p* *then* *k* *else* *0*) *p′ q*

abbreviation *GlobalIncomingInfoAt* **where**
GlobalIncomingInfoAt c q ≡ *GlobalIncomingInfo c 0 q q*

definition *init-config* :: ('*p* :: *finite*, '*a*) *configuration* ⇒ *bool* **where**
init-config c ≡
(∀ *p*. *c-temp c p* = {#}$_z$) ∧
(∀ *p1 p2*. *c-msg c p1 p2* = []) ∧
— Capabilities have non-negative multiplicities
(∀ *p t*. *0* ≤ *zcount* (*c-caps c p*) *t*) ∧
— The pointstamps in glob are exactly those in *records*
(∀ *p*. *c-glob c p* = *records c*) ∧
— All capabilities are being tracked
c-data-msg c = {#}

definition *next-recvcap′* :: ('*p* :: *finite*, '*a*) *configuration* ⇒ ('*p*, '*a*) *configuration* ⇒ '*p* ⇒ '*a* ⇒ *bool* **where**
next-recvcap′ c0 c1 p t = (
(*p*,*t*) ∈# *c-data-msg c0*
∧ *c1* = *c0*(|*c-caps* := (*c-caps c0*)(*p* := *c-caps c0 p* + {#*t*#}$_z$),
c-data-msg := *c-data-msg c0* − {#(*p*,*t*)#}|))

abbreviation *next-recvcap* **where**
next-recvcap c0 c1 ≡ ∃ *p t*. *next-recvcap′ c0 c1 p t*

Can minting of capabilities be described as a refinement of the Abadi model? Short answer: No, not in general. Long answer: Could slightly modify Abadi model, such that a capability always comes with a multiplicity $2^6 4$ (or similar, could be parametrized over arbitrarily large constant). In that case minting new capabilities can be described as an upright change, dropping one of the capabilities, to make the change upright. This only works as long as no capability is required more than the constant number of times. Issues: - Not fully general, due to the arbitrary constant - Not clear whether refinement proofs would be easier than simply altering the model to support the operations

Rationale for the condition on *c-caps c0 p*: In Abadi, the operation *next-performop′* has the premise $\forall\, t.\ int\ (count\ \Delta neg\ t) \leq zcount\ (records\ c0)\ t$, (records corresponds to the global field *nrec* in that model) which means the processor performing the transition must verify that this condition is met. Since *records c* is "global" state, which no processor can know, an implementation of this protocol has to include some other protocol or reasoning for when it is safe to do this transition.

Naively using a processor's *c-glob c p* to approximate *records c* and justify transitions can cause a race condition, where a processor drops a pointstamp, e.g., $\Delta neg = \{\#t\#\}$, after which $zcount\ (records\ c)\ t = 0$ but other processors might still use the pointstamp to justify the creation of pointstamps that violate the safety property.

Instead we model ownership of pointstamps, calling "owned pointstamps" **capabilities**, which are tracked in *c-caps c*. In place of *nrec* we define *records c*, which is the sum of all capabilities, as well as *c-data-msg c*, which contains the capabilities carried by data messages. Since $\forall\, p\ t.\ zcount\ (c\text{-}caps\ c\ p)\ t \leq zcount\ (records\ c)\ t$, our condition $\forall\, t.\ int\ (count\ \Delta neg\ t) \leq zcount\ (c\text{-}caps\ c0\ p)\ t$ implies the one on *nrec* in Abadi's model.

Conditions in performop:

The performop transition takes three msets of pointstamps, Δneg, $\Delta mint\text{-}msg$, and $\Delta mint\text{-}self$ Δneg contains dropped capabilities (a subset of *c-caps*) $\Delta mint\text{-}msg$ contains pairs $(p,\ t)$, where a data message is sent (i.e. capability added to the pool), creating a capability at t, owned by p $\Delta mint\text{-}self$ contains pointstamps minted and owned by worker p

Δneg in combination with $\Delta mint\text{-}msg$ also allows any upright updates to be made as in the Abadi model, meaning this definition allows strictly more behaviors.

The $\Delta mint\text{-}msg \neq \{\#\} \vee zmset\text{-}of\ \Delta mint\text{-}self - zmset\text{-}of\ \Delta neg \neq \{\#\}_z$ condition ensures that no-ops aren't possible. However, it's still possible that the combined Δ is empty. E.g. a processor has capabilities 1 and 2, uses cap 1 to send a message, minting capability 2. Simultaneously it drops a capability 2 (for unrelated reasons), cancelling out the overall change but

shifting a capability to the pool, possibly with a different owner than itself.

definition *next-performop′* :: (*′p*::*finite*, *′a*) *configuration* ⇒ (*′p*, *′a*) *configuration* ⇒ *′p* ⇒ *′a multiset* ⇒ (*′p* × *′a*) *multiset* ⇒ *′a multiset* ⇒ *bool* **where**
next-performop′ c0 c1 p Δ*neg* Δ*mint-msg* Δ*mint-self* =
— Δ*pos* contains all positive changes, Δ the combined positive and negative changes
(*let* Δ*pos* = *timestamps* (*zmset-of* Δ*mint-msg*) + *zmset-of* Δ*mint-self*;
Δ = Δ*pos* − *zmset-of* Δ*neg*
in
(Δ*mint-msg* ≠ {#} ∨ *zmset-of* Δ*mint-self* − *zmset-of* Δ*neg* ≠ {#}$_z$)
∧ (∀ *t*. *int* (*count* Δ*neg t*) ≤ *zcount* (*c-caps c0 p*) *t*)
— Pointstamps added in Δ*mint-self* are minted at p
∧ *minting-self* (*c-caps c0 p*) Δ*mint-self*
— Pointstamps added in Δ*mint-msg* correspond to sent data messages
∧ *minting-msg* (*c-caps c0 p*) Δ*mint-msg*
— Worker immediately knows about dropped and minted capabilities
∧ *c1* = *c0*(|*c-caps* := (*c-caps c0*)(*p* := *c-caps c0 p* + *zmset-of* Δ*mint-self* − *zmset-of* Δ*neg*),
— Sending a data message creates a capability, once that message arrives. This is modelled as a pool of capabilities that may (will) appear at processors at some point.
c-data-msg := *c-data-msg c0* + Δ*mint-msg*,
c-temp := (*c-temp c0*)(*p* := *c-temp c0 p* + Δ)|))

abbreviation *next-performop* **where**
next-performop c0 c1 ≡ (∃ *p* Δ*neg* Δ*mint-msg* Δ*mint-self*. *next-performop′ c0 c1 p* Δ*neg* Δ*mint-msg* Δ*mint-self*)

definition *next-sendupd′* :: (*′p*::*finite*, *′a*) *configuration* ⇒ (*′p*, *′a*) *configuration* ⇒ *′p* ⇒ *′a set* ⇒ *bool* **where**
next-sendupd′ c0 c1 p tt =
(*let* γ = {#*t* ∈#$_z$ *c-temp c0 p*. *t* ∈ *tt*#} *in*
γ ≠ *0*
∧ *justified* (*c-caps c0 p*) (*c-temp c0 p* − γ)
∧ *c1* = *c0*(|*c-msg* := (*c-msg c0*)(*p* := λ*q*. *c-msg c0 p q* @ [γ]),
c-temp := (*c-temp c0*)(*p* := *c-temp c0 p* − γ)|))

abbreviation *next-sendupd* **where**
next-sendupd c0 c1 ≡ (∃ *p tt*. *next-sendupd′ c0 c1 p tt*)

definition *next-recvupd′* :: (*′p*::*finite*, *′a*) *configuration* ⇒ (*′p*, *′a*) *configuration* ⇒ *′p* ⇒ *′p* ⇒ *bool* **where**
next-recvupd′ c0 c1 p q ≡
c-msg c0 p q ≠ []
∧ *c1* = *c0*(|*c-msg* := (*c-msg c0*)(*p* := (*c-msg c0 p*)(*q* := *tl* (*c-msg c0 p q*))),
c-glob := (*c-glob c0*)(*q* := *c-glob c0 q* + *hd* (*c-msg c0 p q*))|)

abbreviation *next-recvupd* **where**
next-recvupd c0 c1 ≡ (∃ *p q*. *next-recvupd′ c0 c1 p q*)

definition *next′* **where**
next′ c0 c1 = (*next-performop c0 c1* ∨ *next-sendupd c0 c1* ∨ *next-recvupd c0 c1* ∨ *next-recvcap c0 c1* ∨ *c1* = *c0*)

abbreviation *next* **where**
next s ≡ *next′* (*shd s*) (*shd* (*stl s*))

definition *spec* :: (′*p* :: *finite*, ′*a*) *computation* ⇒ *bool* **where**
spec s ≡ *holds init-config s* ∧ *alw next s*

abbreviation *GlobVacantUpto* **where**
GlobVacantUpto c q t ≡ *vacant-upto* (*c-glob c q*) *t*

abbreviation *GlobNonposUpto* **where**
GlobNonposUpto c q t ≡ *nonpos-upto* (*c-glob c q*) *t*

abbreviation *RecordsVacantUpto* **where**
RecordsVacantUpto c t ≡ *vacant-upto* (*records c*) *t*

definition *SafeGlobVacantUptoImpliesStickyNrec* :: (′*p* :: *finite*, ′*a*) *computation* ⇒ *bool* **where**
SafeGlobVacantUptoImpliesStickyNrec s =
(*let c* = *shd s in* ∀ *t q*. *GlobVacantUpto c q t* ⟶ *alw* (*holds* (λ*c*. *RecordsVacantUpto c t*)) *s*)

4.2 Auxiliary Lemmas

lemma *finite-induct-select* [*consumes 1*, *case-names empty select*]:
assumes *finite S*
and *empty*: *P* {}
and *select*: ⋀*T*. *finite T* ⟹ *T* ⊂ *S* ⟹ *P T* ⟹ ∃ *s*∈*S* − *T*. *P* (*insert s T*)
shows *P S*
proof −
from *assms*(*1*) **have** *P S* ∧ *finite S*
by (*induct S rule*: *finite-induct-select*) (*auto intro*: *empty select*)
then show *?thesis* **by** *blast*
qed

lemma *finite-induct-decompose-sum*:
fixes *f* :: ′*c* ⇒ (′*b* :: *comm-monoid-add*)
assumes *finite X*
and *x*∈*X*
and *A* (*f x*)
and ∀ *Z*. *B* (*sum f Z*)
and ⋀*x Z*. *A* (*f x*) ⟹ *B* (*sum f Z*) ⟹ *A* (*f x* + *sum f Z*)
and ⋀*x Z*. *B* (*f x*) ⟹ *A* (*sum f Z*) ⟹ *A* (*f x* + *sum f Z*)
shows *A* (∑ *x*∈*X*. *f x*)
using *assms*(*1*,*2*,*3*)

apply (*induct X rule*: *finite-induct-select*)
apply *simp*
apply (*simp add*: *sum.insert-remove*)
subgoal for *T*
apply (*cases* $x \in T$; *simp add*: *assms*(*3*))
apply (*drule psubset-imp-ex-mem*)
apply *clarsimp*
subgoal for *z*
apply (*rule bexI*[*of* - *z*])
apply (*rule conjI*)
apply *clarsimp*
apply (*rule assms*(*6*)[*of z T*])
apply (*rule assms*(*4*)[*THEN spec*, *of* {*z*}, *simplified*])
apply *simp*
apply *simp*
done
apply *clarsimp*
apply (*drule bspec*[*of* - - *x*])
apply *safe*
apply (*rule assms*(*2*))
using *assms*(*4*) *assms*(*5*) **apply** *blast*
done
done

lemma *minting-msg-add-records*: *minting-msg C1 M* $\Longrightarrow \forall t.\ 0 \leq$ *zcount C2 t* $\Longrightarrow$ *minting-msg* (*C1+C2*) *M*
by (*auto simp*: *minting-msg-def intro*: *add-strict-increasing dest*!: *bspec*)

lemma *add-less*: (*a*::*int*) $<$ *c* $\Longrightarrow$ *b* $\leq$ *0* $\Longrightarrow$ *a* + *b* $<$ *c*
by *linarith*

lemma *disj3-split*: *P* $\vee$ *Q* $\vee$ *R* $\Longrightarrow$ (*P* $\Longrightarrow$ *thesis*) $\Longrightarrow$ ($\neg$ *P* $\wedge$ *Q* $\Longrightarrow$ *thesis*) $\Longrightarrow$ ($\neg$ *P* $\Longrightarrow$ $\neg$ *Q* $\Longrightarrow$ *R* $\Longrightarrow$ *thesis*) $\Longrightarrow$ *thesis*
by *blast*

lemma *filter-zmset-conclude-predicate*: *0* $<$ *zcount* {# *x* $\in\#_z$ *M*. *P x* #} *x* $\Longrightarrow$ *0* $<$ *zcount M x* $\Longrightarrow$ *P x*
by (*auto split*: *if-splits*)

lemma *alw-holds2*: *alw* (*holds P*) *ss* = (*P* (*shd ss*) $\wedge$ *alw* (*holds P*) (*stl ss*))
by (*meson alw.simps holds.elims*(*2*) *holds.elims*(*3*))

lemma *zmset-of-remove1-mset*: *x* $\in\#$ *M* $\Longrightarrow$ *zmset-of* (*remove1-mset x M*) = *zmset-of M* $-$ {#*x*#}$_z$
by (*induct M*) *auto*

lemma *timestamps-zmset-of-pair-image*[*simp*]: *timestamps* (*zmset-of* {# (*c*,*t*). *t* $\in\#$ *M* #}) = *zmset-of M*
by (*induct M*) *auto*

lemma *timestamps-image-zmset-fst*[*simp*]: *timestamps* {# (*f x*, *t*). (*x*, *t*) ∈#$_z$ *M* #} = *timestamps M*
apply *transfer*
apply (*clarsimp simp*: *equiv-zmset-def*)
apply (*metis* (*no-types*, *lifting*) *case-prod-unfold image-mset-cong prod.collapse prod.inject*)
done

lemma *lift-invariant-to-spec*:
assumes (⋀*c*. *init-config c* ⟹ *P c*)
and (⋀*s*. *holds P s* ⟹ *next s* ⟹ *nxt* (*holds P*) *s*)
shows *spec s* ⟹ *alw* (*holds P*) *s*
unfolding *spec-def*
apply (*elim conjE*)
apply (*coinduction arbitrary*: *s*)
apply *clarsimp*
apply (*intro conjI assms*(*1*))
apply *safe*
subgoal
proof −
fix *sa* :: (′*b*, ′*a*) *configuration stream*
assume *a1*: *init-config* (*shd sa*)
assume *a2*: *alw next sa*
assume ¬ *alw* (*holds P*) (*stl sa*)
then have ¬ *alw* (λ*s*. *holds P s* ⟶ *nxt* (*holds P*) *s*) *sa*
using *a1* **by** (*metis* (*no-types*) *alw.cases alw-invar assms*(*1*) *holds.elims*(*3*))
then show *init-config* (*shd* (*stl sa*))
using *a2* **by** (*metis* (*lifting*) *alw-iff-sdrop assms*(*2*))
qed
apply *auto*
done

lemma *timestamps-sum-distrib*[*simp*]: (∑ *p* ∈ *A*. *timestamps* (*f p*)) = *timestamps* (∑ *p* ∈ *A*. *f p*)
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *timestamps-zmset-of*[*simp*]: *timestamps* (*zmset-of M*) = *zmset-of* {# *t*. (*p*,*t*) ∈# *M* #}
by (*induct M*) *auto*

lemma *vacant-upto-add*: *vacant-upto a t* ⟹ *vacant-upto b t* ⟹ *vacant-upto* (*a*+*b*) *t*
by (*simp add*: *vacant-upto-def*)

lemma *nonpos-upto-add*: *nonpos-upto a t* ⟹ *nonpos-upto b t* ⟹ *nonpos-upto* (*a*+*b*) *t*
by (*auto intro*: *add-nonpos-nonpos simp*: *nonpos-upto-def*)

lemma *nonzero-lt-gtD*: $(n::-::linorder) \neq 0 \Longrightarrow 0 < n \vee n < 0$
by *auto*

lemma *zero-lt-diff*: $(0::int) < a - b \Longrightarrow b \geq 0 \Longrightarrow 0 < a$
by *auto*

lemma *zero-lt-add-disj*: $0 < (a::int) + b \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 < a \vee 0 < b$
by *auto*

4.2.1 Transition lemmas

lemma *next-performopD*:
assumes *next-performop′ c0 c1 p Δneg Δmint-msg Δmint-self*
shows
$\Delta mint\text{-}msg \neq \{\#\} \vee zmset\text{-}of\ \Delta mint\text{-}self - zmset\text{-}of\ \Delta neg \neq \{\#\}_z$
$\forall t.\ int\ (count\ \Delta neg\ t) \leq zcount\ (c\text{-}caps\ c0\ p)\ t$
minting-self $(c\text{-}caps\ c0\ p)\ \Delta mint\text{-}self$
minting-msg $(c\text{-}caps\ c0\ p)\ \Delta mint\text{-}msg$
$c\text{-}temp\ c1 = (c\text{-}temp\ c0)(p := c\text{-}temp\ c0\ p + (timestamps\ (zmset\text{-}of\ \Delta mint\text{-}msg) + zmset\text{-}of\ \Delta mint\text{-}self - zmset\text{-}of\ \Delta neg))$
$c\text{-}msg\ c1 = c\text{-}msg\ c0$
$c\text{-}glob\ c1 = c\text{-}glob\ c0$
$c\text{-}data\text{-}msg\ c1 = c\text{-}data\text{-}msg\ c0 + \Delta mint\text{-}msg$
$c\text{-}caps\ c1 = (c\text{-}caps\ c0)(p := c\text{-}caps\ c0\ p + (zmset\text{-}of\ \Delta mint\text{-}self - zmset\text{-}of\ \Delta neg))$
using *assms* **by** (*simp-all add*: *next-performop′-def Let-def algebra-simps*)

lemma *next-performop-complexD*:
assumes *next-performop′ c0 c1 p Δneg Δmint-msg Δmint-self*
shows
$records\ c1 = records\ c0 + (timestamps\ (zmset\text{-}of\ \Delta mint\text{-}msg) + zmset\text{-}of\ \Delta mint\text{-}self - zmset\text{-}of\ \Delta neg)$
$GlobalIncomingInfoAt\ c1\ q = GlobalIncomingInfoAt\ c0\ q + (timestamps\ (zmset\text{-}of\ \Delta mint\text{-}msg) + zmset\text{-}of\ \Delta mint\text{-}self - zmset\text{-}of\ \Delta neg)$
$IncomingInfo\ c1\ k\ p'\ q = (if\ p' = p\ then\ IncomingInfo\ c0\ k\ p'\ q + (timestamps\ (zmset\text{-}of\ \Delta mint\text{-}msg) + zmset\text{-}of\ \Delta mint\text{-}self - zmset\text{-}of\ \Delta neg)\ else\ IncomingInfo\ c0\ k\ p'\ q)$
$\forall t'<t.\ zcount\ (c\text{-}caps\ c0\ p)\ t' = 0 \Longrightarrow zcount\ (timestamps\ (zmset\text{-}of\ \Delta mint\text{-}msg))\ t = 0$
$InfoAt\ c1\ k\ p'\ q = InfoAt\ c0\ k\ p'\ q$
proof −
let *?Δ* = $timestamps\ (zmset\text{-}of\ \Delta mint\text{-}msg) + zmset\text{-}of\ \Delta mint\text{-}self - zmset\text{-}of\ \Delta neg$
note *change* = *next-performopD*[*OF assms*(*1*)]
show $records\ c1 = records\ c0 + ?\Delta$
unfolding *records-def change*
apply *simp*
apply (*subst add-diff-eq*[*symmetric*])

apply (*subst sum-if-distrib-add*)
apply (*simp-all add*: *algebra-simps zmset-of-plus*)
done
show *IncomingInfo c1 k p′ q* = (*if p′* = *p*
then IncomingInfo c0 k p′ q + (*timestamps* (*zmset-of* Δ*mint-msg*) + *zmset-of* Δ*mint-self* − *zmset-of* Δ*neg*)
else IncomingInfo c0 k p′ q)
unfolding *IncomingInfo-def change*
by (*auto simp*: *algebra-simps*)
show *GlobalIncomingInfoAt c1 q* = *GlobalIncomingInfoAt c0 q* + *?*Δ **for** *q*
unfolding *GlobalIncomingInfo-def IncomingInfo-def*
by (*rule Sum-eq-pick-changed-elem*[**where** *m* = *p*]) (*simp-all add*: *change algebra-simps*)
show ∀ *t′*<*t*. *zcount* (*c-caps c0 p*) *t′* = *0* ⟹ *zcount* (*timestamps* (*zmset-of* Δ*mint-msg*)) *t* = *0* **for** *t*
by (*rule ccontr*) (*clarsimp dest*!: *image-zmset-pre change*(*4*)[*unfolded minting-msg-def*, *rule-format*])
show *InfoAt c1 k p′ q* = *InfoAt c0 k p′ q*
unfolding *InfoAt-def change* **by** *simp*
qed

lemma *next-sendupdD*:
assumes *next-sendupd′ c0 c1 p tt*
shows
{#*t* ∈#*z c-temp c0 p*. *t* ∈ *tt*#} ≠ {#}*z*
justified (*c-caps c0 p*) (*c-temp c0 p* − {#*t* ∈#*z c-temp c0 p*. *t* ∈ *tt*#})
c-temp c1 p′ = (*if p′* = *p then c-temp c0 p* − {#*t* ∈#*z c-temp c0 p*. *t* ∈ *tt*#} *else c-temp c0 p′*)
c-msg c1 = (λ*p′ q*. *if p′* = *p then c-msg c0 p q* @ [{#*t* ∈#*z c-temp c0 p*. *t* ∈ *tt*#}] *else c-msg c0 p′ q*)
c-glob c1 = *c-glob c0*
c-caps c1 = *c-caps c0*
c-data-msg c1 = *c-data-msg c0*
using *assms* **by** (*simp-all add*: *next-sendupd′-def Let-def fun-eq-iff*)

lemma *next-sendupd-complexD*:
assumes *next-sendupd′ c0 c1 p tt*
shows
records c1 = *records c0*
IncomingInfo c1 0 = *IncomingInfo c0 0*
IncomingInfo c1 k p′ q = (*if p′* = *p* ∧ *length* (*c-msg c0 p q*) < *k*
then IncomingInfo c0 k p′ q − {#*t* ∈#*z c-temp c0 p′*. *t* ∈ *tt*#}
else IncomingInfo c0 k p′ q)
k ≤ *length* (*c-msg c0 p q*) ⟹ *IncomingInfo c1 k p′ q* = *IncomingInfo c0 k p′ q*
length (*c-msg c0 p q*) < *k* ⟹
IncomingInfo c1 k p′ q = (*if p′* = *p*
then IncomingInfo c0 k p′ q − {#*t* ∈#*z c-temp c0 p′*. *t* ∈ *tt*#}

else IncomingInfo c0 k p′ q)
GlobalIncomingInfoAt c1 q = GlobalIncomingInfoAt c0 q
InfoAt c1 k p′ q = (*if p′ = p ∧ k = length* (*c-msg c0 p q*) *then* {#*t* ∈#$_z$ *c-temp c0 p′*. *t* ∈ *tt*#} *else InfoAt c0 k p′ q*)
proof −
note *change* = *next-sendupdD*[*OF assms*]
show *records c1* = *records c0*
by (*simp add*: *records-def change*)
show *ii*: *IncomingInfo c1 k p′ q* = (*if p′ = p ∧ length* (*c-msg c0 p q*) < *k*
then IncomingInfo c0 k p′ q − {#*t* ∈#$_z$ *c-temp c0 p′*. *t* ∈ *tt*#}
else IncomingInfo c0 k p′ q)
by (*simp add*: *algebra-simps IncomingInfo-def change*)
then show *k* ≤ *length* (*c-msg c0 p q*) ⟹ *IncomingInfo c1 k p′ q* = *IncomingInfo c0 k p′ q*
by *auto*
from *ii* **show** *length* (*c-msg c0 p q*) < *k* ⟹
IncomingInfo c1 k p′ q = (*if p′ = p*
then IncomingInfo c0 k p′ q − {#*t* ∈#$_z$ *c-temp c0 p′*. *t* ∈ *tt*#}
else IncomingInfo c0 k p′ q)
by *auto*
have *IncomingInfo c1 0 p q* = *IncomingInfo c0 0 p q* **for** *p q*
by (*simp add*: *algebra-simps IncomingInfo-def change*)
then show *IncomingInfo c1 0* = *IncomingInfo c0 0*
by *auto*
then show *GlobalIncomingInfoAt c1 q* = *GlobalIncomingInfoAt c0 q*
unfolding *GlobalIncomingInfo-def* **by** *auto*
show *InfoAt c1 k p′ q* = (*if p′ = p ∧ k = length* (*c-msg c0 p q*) *then* {#*t* ∈#$_z$ *c-temp c0 p′*. *t* ∈ *tt*#} *else InfoAt c0 k p′ q*)
unfolding *InfoAt-def change*
by (*auto simp*: *nth-append*)
qed

lemma *next-recvupdD*:
assumes *next-recvupd′ c0 c1 p q*
shows
c-msg c0 p q ≠ []
c-temp c1 = *c-temp c0*
c-msg c1 = (λ*p′ q′*. *if p′ = p ∧ q′ = q then tl* (*c-msg c0 p q*) *else c-msg c0 p′ q′*)
c-glob c1 = (*c-glob c0*)(*q* := *c-glob c0 q* + *hd* (*c-msg c0 p q*))
c-caps c1 = *c-caps c0*
c-data-msg c1 = *c-data-msg c0*
using *assms* **by** (*simp-all add*: *next-recvupd′-def fun-eq-iff*)

lemma *next-recvupd-complexD*:
assumes *next-recvupd′ c0 c1 p q*
shows
records c1 = *records c0*

IncomingInfo c1 0 p′ q′ = (if p′ = p ∧ q′ = q then IncomingInfo c0 0 p′ q′ − hd (c-msg c0 p q) else IncomingInfo c0 0 p′ q′)
IncomingInfo c1 k p′ q′ = (if p′ = p ∧ q′ = q
then IncomingInfo c0 (k+1) p′ q′
else IncomingInfo c0 k p′ q′)
GlobalIncomingInfoAt c1 q′ = (if q′ = q then GlobalIncomingInfoAt c0 q′ − hd (c-msg c0 p q) else GlobalIncomingInfoAt c0 q′)
InfoAt c1 k p q = InfoAt c0 (k+1) p q
InfoAt c1 k p′ q′ = (if p′ = p ∧ q′ = q then InfoAt c0 (k+1) p q else InfoAt c0 k p′ q′)
proof −
note *change = next-recvupdD*[*OF assms*]
show *records c1 = records c0*
by (*simp add*: *records-def change*)
show *ii*: *IncomingInfo c1 0 p′ q′ = (if p′ = p ∧ q′ = q then IncomingInfo c0 0 p′ q′ − hd (c-msg c0 p q) else IncomingInfo c0 0 p′ q′)* **for** *p′ q′*
by (*auto simp*: *IncomingInfo-def change algebra-simps sum-list-hd-tl*)
show *IncomingInfo c1 k p′ q′ = (if p′ = p ∧ q′ = q*
then IncomingInfo c0 (k+1) p′ q′
else IncomingInfo c0 k p′ q′)
by (*auto simp*: *IncomingInfo-def change algebra-simps sum-list-hd-tl drop-Suc*)
show *GlobalIncomingInfoAt c1 q′ = (if q′ = q then GlobalIncomingInfoAt c0 q′ − hd (c-msg c0 p q) else GlobalIncomingInfoAt c0 q′)*
unfolding *GlobalIncomingInfo-def*
apply (*cases q′=q*)
apply *simp*
apply (*subst diff-conv-add-uminus*)
apply (*intro Sum-eq-pick-changed-elem*[**where** *m = p*])
apply (*simp-all add*: *ii*)
done
show *InfoAt c1 k p q = InfoAt c0 (k+1) p q*
unfolding *InfoAt-def change*
by (*auto simp*: *nth-tl*)
show *InfoAt c1 k p′ q′ = (if p′ = p ∧ q′ = q then InfoAt c0 (k+1) p q else InfoAt c0 k p′ q′)*
unfolding *InfoAt-def change*
by (*auto simp*: *nth-tl*)
qed

lemma *next-recvcapD*:
assumes *next-recvcap′ c0 c1 p t*
shows
(p,t) ∈# c-data-msg c0
c-temp c1 = c-temp c0
c-msg c1 = c-msg c0
c-glob c1 = c-glob c0
c-caps c1 = (c-caps c0)(p := c-caps c0 p + {#t#}$_z$)
c-data-msg c1 = c-data-msg c0 − {#(p,t)#}
using *assms* **by** (*simp-all add*: *next-recvcap′-def*)

lemma *next-recvcap-complexD*:
assumes *next-recvcap′ c0 c1 p t*
shows
records c1 = records c0
IncomingInfo c1 = IncomingInfo c0
GlobalIncomingInfo c1 = GlobalIncomingInfo c0
InfoAt c1 k p′ q = InfoAt c0 k p′ q
proof −
note *change = next-recvcapD*[*OF assms*]
show *records c1 = records c0*
unfolding *records-def change fun-upd-apply*
apply (*subst sum-if-distrib-add*)
using *change*(*1*) **apply** (*simp-all add*: *zmset-of-remove1-mset algebra-simps records-def change*)
done
show *IncomingInfo c1 = IncomingInfo c0*
unfolding *IncomingInfo-def change* **by** *simp*
then show *GlobalIncomingInfo c1 = GlobalIncomingInfo c0*
unfolding *GlobalIncomingInfo-def* **by** *simp*
show *InfoAt c1 k p′ q = InfoAt c0 k p′ q*
unfolding *InfoAt-def change* **by** *simp*
qed

lemma *ex-next-recvupd*:
assumes *c-msg c0 p q* ≠ []
shows ∃ *c1. next-recvupd′ c0 c1 p q*
using *assms* **unfolding** *next-recvupd′-def*
by (*intro*
exI[*of - c0*(|*c-msg* := (λ*p′ q′. if p′ = p* ∧ *q′ = q then tl* (*c-msg c0 p q*) *else c-msg c0 p′ q′*),
c-glob := (λ*q′. if q′ = q then c-glob c0 q* + *hd* (*c-msg c0 p q*) *else c-glob c0 q′*)|)])
(*auto simp*: *fun-eq-iff*)

4.2.2 Facts about *justified*'ness

lemma *justified-empty*[*simp*]: *justified* {#}$_z$ {#}$_z$
by (*simp add*: *justified-def*)

It's sufficient to show justified for least pointstamps in M.

lemma *justified-leastI*:
assumes ∀ *t. 0 < zcount M t* ⟶ (∀ *t′*<*t. zcount M t′* ≤ *0*) ⟶ *supported-strong M t* ∨ (∃ *t′*<*t. 0 < zcount C t′*) ∨ (*zcount M t < zcount C t*)
shows *justified C M*
unfolding *justified-alt supported-strong-def*
apply (*intro allI impI*)
apply (*drule order-zmset-exists-foundation*)
apply (*elim exE conjE*)

```
  subgoal for t' s
    apply (drule assms(1)[unfolded supported-strong-def, rule-format])
     apply (auto intro: ccontr) []
    apply (elim disj3-split)
      apply (rule disjI1)
    using order.strict-trans2 apply blast
     apply (rule disjI2, rule disjI1)
    using order.strict-trans2 apply blast
    apply (clarsimp simp: nonpos-upto-def)
    apply (metis le-less-linear linear le-imp-less-or-eq preorder-class.le-less-trans)
    done
  done

lemma justified-add:
  assumes justified C1 M1
    and   justified C2 M2
    and   ∀ t. 0 ≤ zcount C1 t
    and   ∀ t. 0 ≤ zcount C2 t
  shows   justified (C1+C2) (M1+M2)
  apply (rule justified-leastI)
  apply (intro allI impI)
  subgoal for t
    apply (cases 0 < zcount M1 t)
    subgoal
     apply (drule assms(1)[unfolded justified-alt supported-strong-def, rule-format])
      apply (elim disj3-split)
      subgoal
        apply (elim exE conjE)
        apply (drule order-zmset-exists-foundation-neg)
        apply (elim exE conjE)
        subgoal for s s'
          apply (cases zcount (M1 + M2) s' < 0)
          subgoal
            apply (rule disjI1)
          apply (auto intro!: exI[of - s'] simp: nonpos-upto-def supported-strong-def)
[]
            done
          subgoal
            apply (subst (asm) not-less)
            apply (cases 0 < zcount M2 s')
             prefer 2
            subgoal by auto
            subgoal
                  apply (drule assms(2)[unfolded justified-alt supported-strong-def,
rule-format])
              apply (elim disj3-split)
              subgoal
                apply (rule disjI1)
                apply (elim exE)
```

subgoal for s''
by (*auto intro*!: *exI*[*of* - s''] *simp*: *nonpos-upto-def supported-strong-def add-nonpos-neg*)
done
subgoal
apply (*rule disjI2*, *rule disjI1*)
apply (*elim exE conjE*)
subgoal for s''
using *assms*(*3*) **by** (*auto simp*: *add-nonneg-pos intro*!: *exI*[*of* - s''])
done
subgoal
by (*metis add.right-neutral add-strict-increasing2 assms*(*3*) *less-add-same-cancel1 order.strict-trans1 pos-add-strict zcount-union*)
done
done
done
done
subgoal
by (*metis add.commute add-mono-thms-linordered-field*(*4*) *assms*(*4*) *add-0 zcount-union*)
subgoal
apply (*cases supported-strong M2 t*)
subgoal
apply (*rule disjI1*)
using *assms*(*1*)[*unfolded justified-alt*]
apply (*subst supported-strong-def*)
apply (*subst* (*asm*) *supported-strong-def*)
apply (*elim exE conjE*)
unfolding *not-ex*
subgoal for s
apply *clarsimp*
apply (*rule exI*[*of* - s])
apply (*intro conjI*)
apply *blast*
apply (*rule add-nonpos-neg*)
apply (*metis assms*(*3*) *le-less-linear less-trans order-class.le-less supported-strong-def*)
apply *simp*
apply (*clarsimp simp*: *nonpos-upto-def*)
done
done
subgoal
apply (*cases* $\exists\, t'{<}t.\ 0 < zcount\ C2\ t'$)
subgoal
by (*metis add-cancel-left-left assms*(*3*) *order-class.order.not-eq-order-implies-strict zcount-union*)
subgoal
apply (*intro disjI2*)
apply *clarsimp*

```
          using assms(2)[unfolded justified-alt, rule-format, of t]
        apply (metis add.commute add-cancel-left-right add-mono-thms-linordered-field(5)
add-strict-increasing2 assms(4) nonzero-lt-gtD)
          done
         done
        done
       done
     subgoal
      apply (cases 0 < zcount M2 t)
       prefer 2
      subgoal by auto
      subgoal
     apply (drule assms(2)[unfolded justified-alt supported-strong-def, rule-format])
       apply (elim disj3-split)
       subgoal
         apply (elim exE conjE)
         apply (drule order-zmset-exists-foundation-neg)
         apply (elim exE conjE)
         subgoal for s s'
           apply (cases zcount (M1 + M2) s' < 0)
           subgoal
             apply (rule disjI1)
          apply (auto intro!: exI[of - s'] simp: nonpos-upto-def supported-strong-def)
[]
             done
           subgoal
             apply (subst (asm) not-less)
             apply (cases 0 < zcount M1 s')
              prefer 2
             subgoal by auto
             subgoal
                  apply (drule assms(1)[unfolded justified-alt supported-strong-def,
rule-format])
               apply (elim disj3-split)
               subgoal
                 apply (rule disjI1)
                 apply (elim exE)
                 subgoal for s''
               by (auto intro!: exI[of - s''] simp: nonpos-upto-def supported-strong-def
add-neg-nonpos)
                 done
               subgoal
                 apply (rule disjI2, rule disjI1)
                 apply (elim exE conjE)
                 subgoal for s''
                  using assms(4) by (auto simp: add-pos-nonneg intro!: exI[of - s''])
                 done
               subgoal
                 apply (rule disjI2, rule disjI1, rule exI[of - s'], rule conjI)
```

```
            using assms(4) by (auto intro!: add-pos-nonneg)
           done
         done
       done
     done
   subgoal
    by (metis add-mono-thms-linordered-field(4) assms(3) add-0 zcount-union)
   subgoal
     apply (cases supported-strong M1 t)
     subgoal
       apply (rule disjI1)
       apply (simp only: supported-strong-def)
       apply (elim exE)
       subgoal for s
         apply (clarsimp simp: nonpos-upto-def intro!: exI[of - s])
      using assms(2)[unfolded justified-alt nonpos-upto-def supported-strong-def,
rule-format, of s]
             assms(4)[rule-format, of s]
         apply (smt (verit, best) dual-order.strict-trans)
         done
       done
     subgoal
       apply (cases ∃ t'<t. 0 < zcount C1 t')
       subgoal
      by (metis add.commute add-cancel-left-left assms(4) order-class.order.not-eq-order-implies-strict
zcount-union)
       subgoal
         apply (intro disjI2)
            apply (metis add.commute add-strict-increasing2 assms(3) not-le
sublist-order.add-less zcount-union)
         done
       done
     done
   done
 done

lemma justified-sum:
  assumes ∀ p∈P. justified (f p) (g p)
    and   ∀ p∈P. ∀ t. 0 ≤ zcount (f p) t
  shows   justified (∑ p∈P. f p) (∑ p∈P. g p)
  using assms
  by (induct P rule: infinite-finite-induct)
    (auto intro!: justified-add sum-nonneg simp: zcount-sum)

lemma justified-add-records:
  assumes justified C M
    and   ∀ t. 0 ≤ zcount C' t
```

```
  shows   justified (C+C') M
  using assms unfolding justified-def
  apply (clarsimp intro: add-pos-nonneg)
  apply (metis add.commute add-strict-increasing2 assms(2))
  done

lemma justified-add-zmset-records:
  assumes justified C M
  shows   justified (add-zmset t C) M
  using assms
  apply (subst add-zmset-add-single)
  apply (rule justified-add-records)
   apply simp-all
  done

lemma justified-diff:
  assumes justified C M
    and   ∀ t. 0 ≤ zcount C t
    and   ∀ t. count Δ t ≤ zcount C t
  shows   justified (C − zmset-of Δ) (M − zmset-of Δ)
proof (intro justified-leastI allI impI)
  fix t
  assume least: ∀ t'<t. zcount (M − zmset-of Δ) t' ≤ 0
  assume 0 < zcount (M − zmset-of Δ) t
  then have Mt: 0 < zcount M t
    by auto
  note assms(1)[unfolded justified-alt, rule-format, OF Mt]
  then consider supported-strong M t | ∃ t'<t. 0 < zcount C t' | zcount M t <
zcount C t
    by blast
  then show supported-strong (M − zmset-of Δ) t ∨ (∃ t'<t. 0 < zcount (C −
zmset-of Δ) t') ∨ zcount (M − zmset-of Δ) t < zcount (C − zmset-of Δ) t
  proof cases
    case 1
    then show ?thesis
      unfolding supported-strong-def
      apply (elim exE)
      subgoal for s
        by (auto intro!: disjI1 exI[of - s] simp: nonpos-upto-def)
      done
  next
    case 2
    then obtain s where s: s < t 0 < zcount C s ∀ s'<s. zcount C s' = 0
      apply atomize-elim
      apply (elim exE conjE)
      apply (drule order-zmset-exists-foundation')
      apply (elim exE conjE)
      subgoal for t' s
        apply (rule exI[of - s])
```

```
      apply (intro conjI)
       apply auto [2]
      apply (intro allI impI)
      subgoal for s'
       using assms(2)[rule-format, of s']
       apply auto
       done
      done
    done
  then consider
    0 < zcount (C − zmset-of Δ) s |
    zcount (C − zmset-of Δ) s = 0 zcount (M − zmset-of Δ) s < 0 |
    zcount (C − zmset-of Δ) s = 0 zcount (M − zmset-of Δ) s = 0 |
    zcount (C − zmset-of Δ) s = 0 zcount (M − zmset-of Δ) s > 0
    using assms(3)[rule-format, of s] by atomize-elim auto
  then show ?thesis
  proof cases
    case 1
    then show ?thesis
     using s by auto
  next
    case 2
    then show ?thesis
     using s least by (auto simp: nonpos-upto-def supported-strong-def)
  next
    case 3
    note case3 = 3
    with s(2) have Ms: 0 < zcount M s
     by − (rule ccontr, auto simp: not-less)
    note assms(1)[unfolded justified-alt, rule-format, OF Ms]
     then consider supported-strong M s | ∃ t'<s. 0 < zcount C t' | zcount M s
< zcount C s
     using not-less by blast
    then show ?thesis
    proof cases
     case 1
     then show ?thesis
      unfolding supported-strong-def
      apply (elim exE conjE)
      subgoal for s'
       apply (intro disjI1 exI[of - s'])
       using s(1,2) apply (auto intro: exI[of - s'] simp: nonpos-upto-def)
       done
      done
    next
     case 2
     then show ?thesis
      using s(3) by auto
    next
```

```
      case 3
      from case3 have zcount C s = zcount M s
        by auto
      with 3 show ?thesis
        by linarith
    qed
  next
    case 4
    then show ?thesis
      using least s(1) by auto
  qed
next
  case 3
  then show ?thesis
    by auto
qed
qed

lemma justified-add-msg-delta:
  assumes justified C M
    and   minting-msg C Δ
    and   ∀t. 0 ≤ zcount C t
  shows   justified C (M + timestamps (zmset-of Δ))
proof (intro allI impI justified-leastI)
  fix t
  assume t: 0 < zcount (M + timestamps (zmset-of Δ)) t
  assume least: ∀t'<t. zcount (M + timestamps (zmset-of Δ)) t' ≤ 0
  have Δt: 0 < zcount (timestamps (zmset-of Δ)) t ⟹ supported-strong (M +
timestamps (zmset-of Δ)) t ∨ (∃t'<t. 0 < zcount C t') ∨ zcount (M + timestamps
(zmset-of Δ)) t < zcount C t
    by (auto dest: pos-image-zmset-obtain-pre[rotated] assms(2)[unfolded mint-
ing-msg-def, rule-format])
  { assume Δt: zcount (timestamps (zmset-of Δ)) t ≤ 0
    with t have Mt: 0 < zcount M t
      by auto
    note assms(1)[unfolded justified-alt, rule-format, OF Mt]
    then consider
      supported-strong M t ∀t'<t. zcount C t' = 0 zcount M t ≥ zcount C t |
      ∃t'<t. 0 < zcount C t' |
      zcount M t < zcount C t
      using not-less assms(3)
      by (metis order-class.order.not-eq-order-implies-strict)
    then have supported-strong (M + timestamps (zmset-of Δ)) t ∨ (∃t'<t. 0 <
zcount C t') ∨ zcount (M + timestamps (zmset-of Δ)) t < zcount C t
    proof cases
      case 1
      then obtain s where s: s < t zcount M s < 0 ∀s'<s. zcount M s' = 0
        unfolding supported-strong-def
        apply atomize-elim
```

```
      apply (elim exE conjE)
      apply (drule order-zmset-exists-foundation-neg')
      apply (elim exE conjE)
      subgoal for s s'
      by (auto intro!: exI[of - s'] simp: nonpos-upto-def order-class.order-antisym)
      done
    then show ?thesis
      apply (cases ∃ s'≤s. zcount (timestamps (zmset-of Δ)) s' > 0)
       apply (elim exE conjE)
      subgoal for s'
        apply (drule pos-image-zmset-obtain-pre[rotated])
         apply simp
        apply (elim exE conjE)
        apply simp
        apply (drule assms(2)[unfolded minting-msg-def, rule-format])
        apply (auto simp: supported-strong-def)
        done
      subgoal
        apply (intro disjI1 exI[of - s])
        unfolding not-less
          apply (metis (full-types) le-less-linear least eq-refl order.strict-trans1
nonpos-upto-def supported-strong-def sublist-order.add-less zcount-union)
        done
      done
  next
    case 2
    then show ?thesis by auto
  next
    case 3
    then show ?thesis
      using Δt by auto
  qed
 }
 then show supported-strong (M + timestamps (zmset-of Δ)) t ∨ (∃ t'<t. 0 <
zcount C t') ∨ zcount (M + timestamps (zmset-of Δ)) t < zcount C t
  apply (cases zcount (timestamps (zmset-of Δ)) t ≤ 0)
   apply blast
  apply (rule Δt)
  apply auto
  done
qed

lemma justified-add-same:
 assumes justified C M
   and   minting-self C Δ
   and   ∀ t. 0 ≤ zcount C t
 shows   justified (C + zmset-of Δ) (M + zmset-of Δ)
proof (intro allI impI justified-leastI)
 fix t
```

```
  assume t: 0 < zcount (M + zmset-of Δ) t
  assume least: ∀ t'<t. zcount (M + zmset-of Δ) t' ≤ 0
  from t consider
    0 < zcount M t |
    0 < zcount (zmset-of Δ) t zcount M t ≤ 0
    by atomize-elim (auto simp: not-less count-inI)
  then show supported-strong (M + zmset-of Δ) t ∨ (∃ t'<t. 0 < zcount (C +
zmset-of Δ) t') ∨ zcount (M + zmset-of Δ) t < zcount (C + zmset-of Δ) t
  proof cases
    case 1
    note assms(1)[unfolded justified-alt, rule-format, OF 1]
    then consider
      supported-strong M t |
      ∃ t'<t. 0 < zcount C t' |
      zcount M t < zcount C t
      by blast
    then show ?thesis
    proof cases
      case 1
      then show ?thesis
        unfolding supported-strong-def
        apply (elim exE conjE)
        subgoal for t'
          apply (cases ∃ s≤t'. zcount (zmset-of Δ) s > 0)
           apply (elim exE conjE)
          subgoal for s
            apply simp
            apply (drule assms(2)[unfolded minting-self-def, rule-format])
            apply (smt (z3) order.strict-trans1 of-nat-0-le-iff)
            done
          subgoal
            apply (intro disjI1 exI[of - t'] conjI)
              apply simp
             apply simp
                apply (metis add-cancel-right-left add-mono-thms-linordered-field(1)
count-eq-zero-iff order.order-iff-strict of-nat-eq-0-iff)
            using least nonpos-upto-def apply auto
            done
          done
        done
    next
      case 2
      then show ?thesis
        by auto
    next
      case 3
      then show ?thesis
        by auto
    qed
```

next
case *2*
then obtain t' **where** t': $t' \leq t\ 0 < zcount\ C\ t'$
using *assms(2)*[*unfolded minting-self-def*]
by *auto*
then show *?thesis*
apply (*cases* t'=t)
subgoal
using *2(2)* **by** *auto*
subgoal
apply (*rule disjI2*, *rule disjI1*)
using *assms(3) order.not-eq-order-implies-strict* **apply** *fastforce*
done
done
qed
qed

4.2.3 Facts about *justified-with*'ness

lemma *justified-with-add-records*:
assumes *justified-with C1 M N*
and $\forall t.\ 0 \leq zcount\ C2\ t$
shows *justified-with* (*C1*+*C2*) *M N*
unfolding *justified-with-def*
apply (*intro allI impI*)
subgoal for t
apply (*drule assms(1)*[*unfolded justified-with-def*, *rule-format*])
apply (*elim disjE*)
subgoal
by *blast*
subgoal
apply (*elim exE*)
subgoal for s
apply (*rule disjI2*, *rule disjI1*)
using *assms(2)*[*rule-format*, *of s*] **by** *auto*
done
subgoal
apply (*intro disjI2*)
using *assms(2)*[*rule-format*, *of t*]
by *auto*
done
done

lemma *justified-with-leastI*:
assumes
$(\forall t.\ 0 < zcount\ M\ t \longrightarrow (\forall t'<t.\ zcount\ M\ t' \leq 0) \longrightarrow$
$(\exists s<t.\ (zcount\ M\ s < 0 \vee zcount\ N\ s < 0) \wedge (\forall s'<s.\ zcount\ M\ s' \leq 0)) \vee$
$(\exists s<t.\ 0 < zcount\ C\ s) \vee$
$zcount\ (M+N)\ t < zcount\ C\ t)$

```
  shows   justified-with C M N
  unfolding justified-with-alt
proof (intro allI impI)
  fix t
  assume t: 0 < zcount M t
  from t obtain t' where t': t' ≤ t 0 < zcount M t' ∀ s<t'. zcount M s ≤ 0
    by atomize-elim (drule order-zmset-exists-foundation')
  note assms[rule-format, OF t'(2)]
  with t'(3) consider
    ∃ s<t'. (zcount M s < 0 ∨ zcount N s < 0) ∧ (∀ s'<s. zcount M s' ≤ 0) |
    ∃ s<t'. 0 < zcount C s |
    zcount (M+N) t' < zcount C t'
    using not-less by blast
  then show (∃ s<t. (zcount M s < 0 ∨ zcount N s < 0) ∧ (∀ s'<s. zcount M s'
≤ 0)) ∨ (∃ s<t. 0 < zcount C s) ∨ zcount (M+N) t < zcount C t
  proof cases
    case 1
    then show ?thesis
      using order.strict-trans2 t'(1) by blast
  next
    case 2
    then show ?thesis
      using order.strict-trans2 t'(1) by blast
  next
    case 3
    then consider
      zcount (M+N) t' < 0 |
      zcount C t' > 0
      by atomize-elim auto
    then show ?thesis
    proof cases
      case 1
      then have zcount N t' < 0
        using t'(2) by auto
      with t'(3) show ?thesis
        apply (cases t'=t)
        subgoal
          using 3(1) by blast
        subgoal
          using t'(1) by (auto intro: exI[of - t'])
        done
    next
      case 2
      then show ?thesis
        apply (cases t'=t)
        subgoal
          apply (intro disjI2)
          using 3(1) apply blast
          done
```

```
      subgoal
        apply (rule disjI2)
        apply (rule disjI1)
        using order.not-eq-order-implies-strict t'(1) apply blast
        done
      done
    qed
  qed
qed

lemma justified-with-add:
  assumes justified-with C1 M N1
    and   justified C1 N1
    and   justified C2 N2
    and   ∀ t. 0 ≤ zcount C1 t
    and   ∀ t. 0 ≤ zcount C2 t
  shows   justified-with (C1+C2) M (N1+N2)
proof (intro justified-with-leastI allI impI)
  fix t
  assume count-t: 0 < zcount M t
  assume least: ∀ t'<t. zcount M t' ≤ 0
  note assms(1)[unfolded justified-with-alt, rule-format, OF count-t]
  then consider
    ∃ s<t. zcount M s < 0 ∧ (∀ s'<s. zcount M s' ≤ 0) |
    ∃ s<t. zcount N1 s < 0 ∧ (∀ s'<s. zcount M s' ≤ 0) |
    ∃ s<t. 0 < zcount C1 s |
    zcount (M + N1) t < zcount C1 t
    by blast
  then show (∃ s<t. (zcount M s < 0 ∨ zcount (N1 + N2) s < 0) ∧ (∀ s'<s. zcount
M s' ≤ 0)) ∨
        (∃ s<t. 0 < zcount (C1 + C2) s) ∨ zcount (M + (N1 + N2)) t < zcount
(C1 + C2) t
  proof cases
    case 1
    then show ?thesis
      by blast
  next
    case 2
    then obtain s where s: s < t zcount N1 s < 0 ∀ s'<s. 0 ≤ zcount N1 s'
      apply atomize-elim
      apply (elim exE conjE)
      apply (drule order-zmset-exists-foundation-neg')
      using order.strict-trans order.strict-trans1 apply blast
      done
    then consider
      zcount (N1 + N2) s < 0 |
      0 < zcount N2 s zcount (N1 + N2) s ≥ 0
      by atomize-elim auto
    then show ?thesis
```

```
  proof cases
    case 1
    then show ?thesis
      using least order.strict-trans s(1) by blast
  next
    case 2
    note assms(3)[unfolded justified-alt, rule-format, OF 2(1)]
    then consider
      supported-strong N2 s ∀ t'<s. zcount C2 t' ≤ 0 zcount N2 s ≥ zcount C2 s |
      ∃ t'<s. 0 < zcount C2 t' |
      zcount N2 s < zcount C2 s
      using not-less by blast
    then show ?thesis
    proof cases
      case 1
      then obtain s' where s': s' < s zcount N2 s' < 0 nonpos-upto N2 s'
        unfolding supported-strong-def
        by blast
      from s'(2) have nonneg: 0 ≤ zcount (N1+N2) s' ⟹ 0 < zcount N1 s'
        by auto
      show ?thesis
        apply (cases zcount (N1 + N2) s' < 0)
        subgoal
          using least order.strict-trans s'(1) s(1) by (intro disjI1) blast
        subgoal
          unfolding not-less
          apply (drule nonneg)
               apply (drule assms(2)[unfolded justified-alt supported-strong-def,
rule-format])
          apply (elim disjE exE conjE)
          subgoal for u
            by (meson local.order.strict-trans not-less s'(1) s(3))
          subgoal for u
              by (metis add-strict-increasing assms(5) local.less-trans s'(1) s(1)
zcount-union)
          subgoal
             by (smt (verit, ccfv-threshold) assms(5) local.dual-order.strict-trans
s'(1) s(1) s(3) zcount-union)
          done
        done
    next
      case 2
      then show ?thesis
        apply (elim exE conjE)
        subgoal for s'
          apply (rule disjI2)
          apply (rule disjI1)
          using assms(4)[rule-format, of s'] s(1)
          apply (auto intro!: exI[of - s'])
```

```
          done
        done
    next
      case 3
      then show ?thesis
        by (smt (verit) 2(1) assms(4) s(1) zcount-union)
    qed
  qed
next
  case 3
  then show ?thesis
    using assms(5)
    apply −
    apply (rule disjI2)
    apply (rule disjI1)
    apply (metis add-strict-increasing zcount-union)
    done
next
  case 4
  then show ?thesis
  proof (cases zcount (M + (N1 + N2)) t < zcount (C1 + C2) t)
    case True
    then show ?thesis
      by blast
  next
    case False
    then have N2t: 0 < zcount N2 t
      using 4 assms(5)[rule-format, of t]
      unfolding not-less zcount-union
      by linarith
    then show ?thesis
      using assms(3)[unfolded justified-alt supported-strong-def, rule-format, OF
N2t]
      apply (elim exE conjE disjE)
      subgoal for s
        apply (cases 0 < zcount N1 s)
        subgoal
              apply (drule assms(2)[unfolded justified-alt supported-strong-def,
rule-format])
          apply (elim exE conjE disjE)
          subgoal for s'
            apply (rule disjI1)
            apply (rule exI[of - s'])
            apply (intro conjI)
              apply simp
         apply (metis add-cancel-right-right add-neg-neg order.strict-implies-order
nonpos-upto-def order-class.order.not-eq-order-implies-strict zcount-union)
            apply (meson least less-trans)
            done
```

```
        subgoal for s'
          by (metis add-pos-nonneg assms(5) less-trans zcount-union)
        subgoal
          apply (rule ccontr)
          apply (clarsimp simp: not-le not-less)
        apply (metis (no-types, opaque-lifting) add-cancel-right-right add-neg-neg
assms(4) assms(5) least less-trans not-less order-class.order.not-eq-order-implies-strict
pos-add-strict)
          done
        done
      subgoal
        apply (intro disjI1 exI[of - s])
        apply (intro disjI2 conjI)
          apply simp
         apply simp
        using least apply simp
        done
      done
    subgoal for s
      by (metis add.comm-neutral add-mono-thms-linordered-field(4) assms(4)
zcount-union)
    subgoal
      using 4 by auto
    done
  qed
 qed
qed

lemma justified-with-sum':
 assumes finite X X≠{}
   and   ∀x∈X. justified-with (C x) M (N x)
   and   ∀x∈X. justified (C x) (N x)
   and   ∀x∈X. ∀t. 0 ≤ zcount (C x) t
 shows   justified-with (∑x∈X. C x) M (∑x∈X. N x)
 using assms
proof (induct X rule: finite-induct)
 case empty
 then show ?case by simp
next
 case (insert x F)
 show ?case
   apply (cases F={})
   subgoal
     using insert(5) by simp
   subgoal
     apply (subst (1 2) sum.insert-remove)
     using insert(1) apply simp
     using insert(2) apply simp
     apply (rule justified-with-add)
```

```
      using insert(5) apply simp
      using insert(6) apply simp
        apply (rule justified-sum)
      using insert(6) apply simp
      using insert(7) apply simp
      using insert(7) apply simp
      apply (intro allI)
      unfolding zcount-sum
      apply (rule sum-nonneg)
      using insert(7) apply simp
      done
    done
qed

lemma justified-with-sum:
  assumes finite X X≠{}
    and   x ∈ X
    and   justified-with (C x) M (N x)
    and   ∀x∈X. justified (C x) (N x)
    and   ∀x∈X. ∀t. 0 ≤ zcount (C x) t
  shows   justified-with (∑x∈X. C x) M (∑x∈X. N x)
  using assms
proof (induct X rule: finite-induct)
  case empty
  then show ?case
    by simp
next
  case (insert y F)
  thm insert
  show ?case
    apply (cases F={})
    subgoal
      using insert by simp
    subgoal
      apply (subst (1 2) sum.insert-remove)
      using insert(1) apply simp
      using insert(2) apply simp
      apply (cases y=x)
      subgoal
        apply (rule justified-with-add)
        using insert(6) apply simp
        using insert(7) apply simp
          apply (rule justified-sum)
        using insert(7) apply simp
        using insert(8) apply simp
        using insert(8) apply simp
        apply (intro allI)
        unfolding zcount-sum
        apply (rule sum-nonneg)
```

```
      using insert(8) apply simp
      done
    subgoal
      apply (subst (1 2) add.commute)
      apply (rule justified-with-add)
        apply (rule insert(3))
          apply simp
      using insert(5) apply simp
      using insert(6) apply simp
      using insert(7) apply simp
      using insert(8) apply simp
       apply (rule justified-sum)
      using insert(7) apply simp
      using insert(8) apply simp
      using insert(7) apply simp
       apply (intro allI)
      unfolding zcount-sum
       apply (rule sum-nonneg)
      using insert(8) apply simp
      using insert(8) apply simp
      done
    done
  done
qed

lemma justified-with-add-same:
  assumes justified-with C M N
    and  ∀t. 0 ≤ zcount C t
  shows  justified-with (C + zmset-of Δ) M (N + zmset-of Δ)
  unfolding justified-with-def
proof (intro allI impI)
  fix t
  assume Mt: 0 < zcount M t
  note assms(1)[unfolded justified-with-alt, rule-format, OF Mt]
  with Mt consider
    ∃s<t. zcount M s < 0 ∧ (∀s'<s. zcount M s' ≤ 0) |
    ∃s<t. zcount N s < 0 ∧ (∀s'<s. zcount M s' ≤ 0) |
    ∃s<t. 0 < zcount C s |
    zcount (M + N) t < zcount C t
    using not-less by blast
  then show (∃s<t. (zcount M s < 0 ∨ zcount (N + zmset-of Δ) s < 0)) ∨
        (∃s<t. 0 < zcount (C + zmset-of Δ) s) ∨ zcount (M + (N + zmset-of
Δ)) t < zcount (C + zmset-of Δ) t
  proof cases
    case 1
    then show ?thesis
      by blast
  next
    case 2
```

```
    then show ?thesis
      by (metis add-less-same-cancel2 assms(2) not-less preorder-class.le-less-trans
zcount-union)
  next
    case 3
    then show ?thesis
      by fastforce
  next
    case 4
    then show ?thesis
      by auto
  qed
qed

lemma justified-with-add-msg-delta:
  assumes justified-with C M N
    and   minting-msg C Δ
    and   ∀ t. 0 ≤ zcount C t
  shows   justified-with C M (N + timestamps (zmset-of Δ))
  unfolding justified-with-def
proof (intro allI impI)
  fix t
  assume Mt: 0 < zcount M t
  note assms(1)[unfolded justified-with-alt, rule-format, OF Mt]
  with Mt consider
    ∃ s<t. zcount M s < 0 ∧ (∀ s'<s. zcount M s' ≤ 0) |
    ∃ s<t. zcount N s < 0 ∧ (∀ s'<s. zcount M s' ≤ 0) |
    ∃ s<t. 0 < zcount C s |
    zcount (M + N) t < zcount C t
    using not-less by blast
  then show (∃ s<t. (zcount M s < 0 ∨ zcount (N + timestamps (zmset-of Δ)) s
< 0)) ∨
        (∃ s<t. 0 < zcount C s) ∨ zcount (M + (N + timestamps (zmset-of Δ))) t
< zcount C t
  proof cases
    case 1
    then show ?thesis
      by blast
  next
    case 2
    then obtain s where s: s < t zcount N s < 0
      by blast
    then show ?thesis
    proof (cases ∃ (p,s')∈#Δ. s' ≤ s)
      case True
      then show ?thesis
        apply −
        apply clarsimp
        apply (drule assms(2)[unfolded minting-msg-def, rule-format])
```

```
      using order.strict-trans order.strict-trans1 s(1) not-less apply blast
      done
    next
      case False
      then have zcount (timestamps (zmset-of Δ)) s = 0
        by (force intro: ccontr dest: pos-image-zmset-obtain-pre[rotated])
      then show ?thesis
        by (metis plus-int-code(1) s(1,2) zcount-union)
    qed
  next
    case 3
    then show ?thesis
      by blast
  next
    case 4
    then show ?thesis
      apply (cases zcount (timestamps (zmset-of Δ)) t > 0)
        apply (auto dest: pos-image-zmset-obtain-pre[rotated] assms(2)[unfolded
minting-msg-def, rule-format]) []
      unfolding not-less apply auto
      done
  qed
qed

lemma justified-with-diff:
  assumes justified-with C M N
    and  ∀ t. 0 ≤ zcount C t
    and  ∀ t. count Δ t ≤ zcount C t
    and  justified C N
  shows  justified-with (C − zmset-of Δ) M (N − zmset-of Δ)
proof (intro allI impI justified-with-leastI)
  fix t
  assume Mt: 0 < zcount M t
  assume least: ∀ t'<t. zcount M t' ≤ 0
  note assms(1)[unfolded justified-with-alt, rule-format, OF Mt]
  with Mt consider
    ∃ s<t. zcount M s < 0 ∧ (∀ s'<s. zcount M s' ≤ 0) |
    ∃ s<t. zcount N s < 0 ∧ (∀ s'<s. zcount M s' ≤ 0) |
    ∃ s<t. 0 < zcount C s ∀ s<t. zcount M s ≥ 0 ∨ ¬ (∀ s'<s. zcount M s' ≤ 0)
∀ s<t. zcount N s ≥ 0 ∨ ¬ (∀ s'<s. zcount M s' ≤ 0) zcount (M + N) t ≥ zcount
C t  |
    zcount (M + N) t < zcount C t
    using not-less by blast
  then show (∃ s<t. (zcount M s < 0 ∨ zcount (N − zmset-of Δ) s < 0) ∧ (∀ s'<s.
zcount M s' ≤ 0)) ∨
        (∃ s<t. 0 < zcount (C − zmset-of Δ) s) ∨ zcount (M + (N − zmset-of
Δ)) t < zcount (C − zmset-of Δ) t
  proof cases
    case 1
```

```
    then show ?thesis
      by blast
  next
    case 2
    then show ?thesis
      using diff-add-cancel zcount-union zcount-zmset-of-nonneg by auto
  next
    case 3
    then obtain s where s: s < t 0 < zcount C s ∀ s'<s. zcount C s' = 0
      apply atomize-elim
      apply (elim exE conjE)
      apply (drule order-zmset-exists-foundation')
      apply (elim exE conjE)
      subgoal for t' s
        apply (rule exI[of - s])
        apply (intro conjI)
          apply auto [2]
        apply (intro allI impI)
        subgoal for s'
          using assms(2)[rule-format, of s'] by auto
        done
      done
    then consider
      0 < zcount (C − zmset-of Δ) s |
      zcount (C − zmset-of Δ) s = 0 zcount (N − zmset-of Δ) s < 0 |
      zcount (C − zmset-of Δ) s = 0 zcount (N − zmset-of Δ) s = 0 |
      zcount (C − zmset-of Δ) s = 0 zcount (N − zmset-of Δ) s > 0
      using assms(3)[rule-format, of s] by atomize-elim auto
    then show ?thesis
    proof cases
      case 1
      then show ?thesis
        using s by auto
    next
      case 2
      then show ?thesis
        using s least by (auto simp: nonpos-upto-def)
    next
      case 3
      note case3 = 3
      with s(2) have Ns: 0 < zcount N s
        by (auto intro: ccontr simp: not-less)
      note assms(4)[unfolded justified-alt, rule-format, OF Ns]
      then consider supported-strong N s | ∃ t'<s. 0 < zcount C t' | zcount N s <
zcount C s
        using not-less by blast
      then show ?thesis
      proof cases
        case 1
```

```
      then show ?thesis
        unfolding supported-strong-def
        apply (elim exE conjE)
        subgoal for s'
          using s(1,2) least by (auto intro: exI[of - s'] simp: nonpos-upto-def)
        done
      next
        case 2
        then show ?thesis
          using s(3) by auto
      next
        case 3
        from case3 have zcount C s = zcount N s
          by auto
        with 3 show ?thesis
          by linarith
      qed
    next
      case 4
      then have Ns: 0 < zcount N s
        by auto
      note assms(4)[unfolded justified-alt, rule-format, OF Ns]
      then consider supported-strong N s | ∃ t'<s. 0 < zcount C t' | zcount N s <
zcount C s
        using not-less by blast
      then show ?thesis
      proof cases
        case 1
        then show ?thesis
          unfolding supported-strong-def
          apply (elim exE conjE)
          subgoal for s'
            using s(1,2) least by (auto intro: exI[of - s'] simp: nonpos-upto-def)
          done
      next
        case 2
        then show ?thesis
          using s(3) by auto
      next
        case 3
        have zcount C s = zcount N s
          using 3 4(1,2) by auto
        with 3 show ?thesis
          by linarith
      qed
    qed
  next
    case 4
    then show ?thesis
```

```
    by auto
  qed
qed

lemma PositiveImplies-justified-with:
  assumes justified C (M+N)
    and   PositiveImplies M (M+N)
  shows   justified-with C M N
  unfolding justified-with-def
  apply (intro allI impI)
  apply (drule assms(2)[unfolded PositiveImplies-def, rule-format])
  apply (frule assms(1)[unfolded justified-alt supported-strong-def, rule-format])
  apply (elim disjE)
  subgoal for t
    apply (intro disjI1)
    apply (elim exE)
    subgoal for s
      apply (clarsimp intro!: exI[of - s])
      done
    done
  subgoal for t
    using less-imp-le by blast
  subgoal for t
    by (intro disjI2 exI[of - t]) auto
  done

lemma justified-with-add-zmset[simp]:
  assumes justified-with C M N
  shows   justified-with (add-zmset c C) M N
  using assms
  apply (subst add-zmset-add-single)
  apply (rule justified-with-add-records)
   apply simp-all
  done

lemma next-performop'-preserves-justified-with:
  assumes justified-with (c-caps c0 p) M N
    and   next-performop' c0 c1 p Δneg Δmint-msg Δmint-self
    and   ∀t. 0 ≤ zcount (c-caps c0 p) t
    and   justified (c-caps c0 p) N
  shows   justified-with (c-caps c0 p + zmset-of Δmint-self − zmset-of Δneg) M
(N + zmset-of Δmint-self + timestamps (zmset-of Δmint-msg) − zmset-of Δneg)
  apply (rule justified-with-diff)
     apply (rule justified-with-add-msg-delta)
       apply (rule justified-with-add-same)
  using assms(1) apply simp
  using assms(3) apply simp
     apply (rule minting-msg-add-records)
  using next-performopD(4)[OF assms(2)] apply simp
```

apply *simp*
using *assms*(3) **apply** *simp*
using *assms*(3) **apply** *simp*
using *next-performopD*(2)[*OF assms*(2)] **apply** (*simp add*: *add.commute add-increasing*)
apply (*rule justified-add-msg-delta*)
apply (*rule justified-add-same*)
using *assms*(4) **apply** *simp*
using *next-performopD*(3)[*OF assms*(2)] **apply** *simp*
using *assms*(3) **apply** *simp*
apply (*rule minting-msg-add-records*)
using *next-performopD*(4)[*OF assms*(2)] **apply** *simp*
apply *simp*
using *assms*(3) **apply** (*simp add*: *add.commute add-increasing*)
done

4.3 Invariants

4.3.1 InvRecordCount

InvRecordCount states that for every processor, its local approximation *c-glob c q* and the sum of all incoming progress updates *GlobalIncoming-InfoAt c q* together are equal to the sum of all capabilities in the system.

definition *InvRecordCount* **where**
InvRecordCount c ≡ ∀ *q*. *records c* = *GlobalIncomingInfoAt c q* + *c-glob c q*

lemma *init-config-implies-InvRecordCount*: *init-config c* ⟹ *InvRecordCount c*
by (*simp add*: *InvRecordCount-def init-config-def GlobalIncomingInfo-def IncomingInfo-def*)

lemma *performop-preserves-InvRecordCount*:
assumes *InvRecordCount c0*
and *next-performop′ c0 c1 p* Δ*neg* Δ*mint-msg* Δ*mint-self*
shows *InvRecordCount c1*
proof −
note *change* = *next-performopD*[*OF assms*(2)]
note *complex-change* = *next-performop-complexD*[*OF assms*(2)]
show *InvRecordCount c1*
unfolding *InvRecordCount-def complex-change*
by (*auto intro*: *assms*(1)[*unfolded InvRecordCount-def*, *rule-format*] *simp*: *change*)
qed

lemma *sendupd-preserves-InvRecordCount*:
assumes *InvRecordCount c0*
and *next-sendupd′ c0 c1 p tt*
shows *InvRecordCount c1*
proof −
note *change* = *next-sendupdD*[*OF assms*(2)]
note *complex-change* = *next-sendupd-complexD*[*OF assms*(2)]

from *assms*(*1*) **show** *InvRecordCount c1*
unfolding *InvRecordCount-def complex-change change*(*5*) **by** *assumption*
qed

lemma *recvupd-preserves-InvRecordCount*:
assumes *InvRecordCount c0*
and *next-recvupd′ c0 c1 p q*
shows *InvRecordCount c1*
proof −
note *change* = *next-recvupdD*[*OF assms*(*2*)]
note *complex-change* = *next-recvupd-complexD*[*OF assms*(*2*)]
show *InvRecordCount c1*
unfolding *InvRecordCount-def complex-change change*(*4*)
by (*auto simp*: *assms*(*1*)[*unfolded InvRecordCount-def*, *rule-format*])
qed

lemma *recvcap-preserves-InvRecordCount*:
assumes *InvRecordCount c0*
and *next-recvcap′ c0 c1 p t*
shows *InvRecordCount c1*
proof −
note *change* = *next-recvcapD*[*OF assms*(*2*)]
note *complex-change* = *next-recvcap-complexD*[*OF assms*(*2*)]
show *InvRecordCount c1*
unfolding *InvRecordCount-def complex-change change*(*4*)
by (*auto simp*: *assms*(*1*)[*unfolded InvRecordCount-def*, *rule-format*])
qed

lemma *next-preserves-InvRecordCount*: *InvRecordCount c0* ⟹ *next′ c0 c1* ⟹
InvRecordCount c1
unfolding *next′-def*
apply (*elim disjE*)
subgoal
using *performop-preserves-InvRecordCount* **by** *auto*
subgoal
using *sendupd-preserves-InvRecordCount* **by** *auto*
subgoal
using *recvupd-preserves-InvRecordCount* **by** *auto*
subgoal
using *recvcap-preserves-InvRecordCount* **by** *auto*
subgoal
by *simp*
done

lemma *alw-InvRecordCount*: *spec s* ⟹ *alw* (*holds InvRecordCount*) *s*
using *lift-invariant-to-spec init-config-implies-InvRecordCount next-preserves-InvRecordCount*
by (*metis* (*mono-tags*, *lifting*) *holds.elims*(*2*) *holds.elims*(*3*) *nxt.simps*)

4.3.2 InvCapsNonneg and InvRecordsNonneg

InvCapsNonneg states that elements in a processor's *c-caps c p* always have non-negative cardinality. InvRecordsNonneg lifts this result to *records c*

definition *InvCapsNonneg* :: (*'p* :: *finite*, *'a*) *configuration* ⇒ *bool* **where**
InvCapsNonneg c = (∀ *p t*. *0* ≤ *zcount* (*c-caps c p*) *t*)

definition *InvRecordsNonneg* **where**
InvRecordsNonneg c = (∀ *t*. *0* ≤ *zcount* (*records c*) *t*)

lemma *init-config-implies-InvCapsNonneg*: *init-config c* ⟹ *InvCapsNonneg c*
unfolding *init-config-def InvCapsNonneg-def* **by** *simp*

lemma *performop-preserves-InvCapsNonneg*:
assumes *InvCapsNonneg c0*
and *next-performop' c0 c1 p* Δ_m Δ_{p1} Δ_{p2}
shows *InvCapsNonneg c1*
using *assms* **unfolding** *InvCapsNonneg-def next-performop'-def Let-def*
by *clarsimp* (*metis add.right-neutral add-mono-thms-linordered-semiring*(*1*) *of-nat-0-le-iff*)

lemma *sendupd-performs-InvCapsNonneg*:
assumes *InvCapsNonneg c0*
and *next-sendupd' c0 c1 p tt*
shows *InvCapsNonneg c1*
using *assms* **by** (*simp add*: *InvCapsNonneg-def next-sendupd'-def Let-def*)

lemma *recvupd-preserves-InvCapsNonneg*:
assumes *InvCapsNonneg c0*
and *next-recvupd' c0 c1 p q*
shows *InvCapsNonneg c1*
using *assms* **unfolding** *InvCapsNonneg-def next-recvupd'-def*
by *simp*

lemma *recvcap-preserves-InvCapsNonneg*:
assumes *InvCapsNonneg c0*
and *next-recvcap' c0 c1 p t*
shows *InvCapsNonneg c1*
using *assms* **unfolding** *InvCapsNonneg-def next-recvcap'-def*
by *simp*

lemma *next-preserves-InvCapsNonneg*: *holds InvCapsNonneg s* ⟹ *next s* ⟹ *nxt* (*holds InvCapsNonneg*) *s*
unfolding *next'-def*
apply (*elim disjE*)
subgoal
using *performop-preserves-InvCapsNonneg* **by** *auto*
subgoal
using *sendupd-performs-InvCapsNonneg* **by** *auto*
subgoal

using *recvupd-preserves-InvCapsNonneg* **by** *auto*
subgoal
using *recvcap-preserves-InvCapsNonneg* **by** *auto*
subgoal
by *simp*
done

lemma *alw-InvCapsNonneg*: *spec s* $\Longrightarrow$ *alw* (*holds InvCapsNonneg*) *s*
using *lift-invariant-to-spec next-preserves-InvCapsNonneg init-config-implies-InvCapsNonneg*
by *blast*

lemma *alw-InvRecordsNonneg*: *spec s* $\Longrightarrow$ *alw* (*holds InvRecordsNonneg*) *s*
apply (*rule alw-mp*[**where** φ = *holds InvCapsNonneg*])
using *alw-InvCapsNonneg* **apply** *assumption*
apply (*rule all-imp-alw*)
unfolding *InvCapsNonneg-def InvRecordsNonneg-def records-def*
apply (*auto intro*!: *add-nonneg-nonneg sum-nonneg simp*: *zcount-sum*)
done

4.3.3 Resulting lemmas

lemma *pos-caps-pos-records*:
assumes *InvCapsNonneg c*
shows *0* < *zcount* (*c-caps c p*) *x* $\Longrightarrow$ *0* < *zcount* (*records c*) *x*
proof −
fix *x*
assume *0* < *zcount* (*c-caps c p*) *x*
then have *0* < *zcount* ($\sum$ *p*∈*UNIV*. *c-caps c p*) *x*
using *assms*(*1*)[*unfolded InvCapsNonneg-def*]
by (*auto intro*!: *sum-pos2 simp*: *zcount-sum*)
then show *0* < *zcount* (*records c*) *x*
unfolding *records-def* **by** *simp*
qed

4.3.4 SafeRecordsMono

The records in the system are monotonic, i.e. once *records c* contains no records up to some timestamp t, then it will stay that way forever.

definition *SafeRecordsMono* :: (′*p* :: *finite*, ′*a*) *computation* ⇒ *bool* **where**
SafeRecordsMono s = (∀ *t*. *RecordsVacantUpto* (*shd s*) *t* ⟶ *alw* (*holds* (λ*c*. *RecordsVacantUpto c t*)) *s*)

lemma *performop-preserves-RecordsVacantUpto*:
assumes *RecordsVacantUpto c0 t*
and *next-performop′ c0 c1 p* Δ*neg* Δ*mint-msg* Δ*mint-self*
and *InvRecordsNonneg c1*
and *InvCapsNonneg c0*
shows *RecordsVacantUpto c1 t*
proof −

```
  note InvRecordsNonneg = assms(3)[rule-format]
  { fix s
    let ?Δpos = timestamps (zmset-of Δmint-msg) + zmset-of Δmint-self
    let ?Δ = ?Δpos − zmset-of Δneg
    note change = next-performopD[OF assms(2)]
    note complex-change = next-performop-complexD[OF assms(2)]
    assume s: s ≤ t zcount (records c1) s ≠ 0
    then have s-pos: 0 < zcount (records c1) s
      using InvRecordsNonneg
       by (simp add: order-class.order.not-eq-order-implies-strict InvRecordsNon-
neg-def)
    have Δ-in-nrec: 0 < zcount ?Δ t ⟹ ∃ t'≤t. 0 < zcount (records c0) t' for t
      apply (subst (asm) zcount-diff)
      apply (subst (asm) zcount-union)
      apply (drule zero-lt-diff)
       apply simp
      apply (drule zero-lt-add-disj)
        apply simp
       apply simp
      apply (erule disjE)
      subgoal
        apply (drule pos-image-zmset-obtain-pre[rotated])
             apply (auto dest!: change(4)[unfolded minting-msg-def, rule-format]
pos-caps-pos-records[OF assms(4)] less-imp-le)
        done
      subgoal
       by (auto dest!: change(3)[unfolded minting-self-def, rule-format] pos-caps-pos-records[OF
assms(4)])
      done
    have nrec0s: zcount (records c0) s = 0
      by (rule assms(1)[unfolded vacant-upto-def, rule-format, OF s(1)])
    have zcount (records c1) s ≤ 0
      unfolding complex-change
      apply (subst zcount-union)
      apply (subst nrec0s)
      apply (subst add-0)
      apply (rule ccontr)
      unfolding not-le
      apply (drule Δ-in-nrec[of s])
       apply (meson assms(1) order-trans pos-zcount-in-zmset s(1) vacant-upto-def
zcount-ne-zero-iff)
      done
    with s-pos have False
      by linarith
  }
  note r = this
  from assms show ?thesis
    unfolding next-performop'-def Let-def vacant-upto-def
    apply clarify
```

```
    apply (rule ccontr)
    apply (rule r)
     apply auto
    done
qed

lemma next'-preserves-RecordsVacantUpto:
  fixes c0 :: ('p::finite, 'a) configuration
  shows InvCapsNonneg c0 ⟹ InvRecordsNonneg c1 ⟹ RecordsVacantUpto c0
t ⟹ next' c0 c1 ⟹ RecordsVacantUpto c1 t
  unfolding next'-def
  apply (elim disjE)
  subgoal
    by (auto intro: performop-preserves-RecordsVacantUpto)
  subgoal
    by (auto simp: next-sendupd'-def Let-def records-def)
  subgoal
    by (auto simp: next-recvupd'-def records-def)
  subgoal
    by (auto dest: next-recvcap-complexD)
  subgoal
    by simp
  done

lemma alw-next-implies-alw-SafeRecordsMono:
  alw next s ⟹ alw (holds InvCapsNonneg) s ⟹ alw (holds InvRecordsNonneg)
s ⟹ alw SafeRecordsMono s
  apply (coinduction arbitrary: s)
  subgoal for s
    unfolding spec-def next'-def SafeRecordsMono-def Let-def
    apply (rule exI[of - s])
    apply safe
    subgoal for t
      apply (coinduction arbitrary: s rule: alw.coinduct)
      apply clarsimp
      apply (rule conjI)
       apply blast
      apply (erule alw.cases)
      apply clarsimp
       apply (metis (no-types, lifting) next'-def next'-preserves-RecordsVacantUpto
alw-holds2)
      done
    apply blast
    done
  done

lemma alw-SafeRecordsMono: spec s ⟹ alw SafeRecordsMono s
   by (auto intro!: alw-next-implies-alw-SafeRecordsMono alw-InvRecordsNonneg
alw-InvCapsNonneg simp: spec-def)
```

4.3.5 InvJustifiedII and InvJustifiedGII

These two invariants state that any net-positive change in the sum of incoming progress updates is "justified" by one of several statements being true.

definition *InvJustifiedII* **where**
InvJustifiedII c = (∀ *k p q*. *justified* (*c-caps c p*) (*IncomingInfo c k p q*))

definition *InvJustifiedGII* **where**
InvJustifiedGII c = (∀ *k p q*. *justified* (*records c*) (*GlobalIncomingInfo c k p q*))

Given some zmset M justified wrt to *caps c0 p*, after a performop $M + \Delta$ is justified wrt to *c-caps c1 p*. This lemma captures the identical argument used for preservation of InvTempJustified and InvJustifiedII.

lemma *next-performop′-preserves-justified*:
assumes *justified* (*c-caps c0 p*) *M*
and *next-performop′ c0 c1 p* Δ*neg* Δ*mint-msg* Δ*mint-self*
and *InvCapsNonneg c0*
shows *justified* (*c-caps c1 p*) (*M* + (*timestamps* (*zmset-of* Δ*mint-msg*) + *zmset-of* Δ*mint-self* − *zmset-of* Δ*neg*))
proof −
let *?*Δ*pos* = *timestamps* (*zmset-of* Δ*mint-msg*) + *zmset-of* Δ*mint-self*
let *?*Δ = *?*Δ*pos* − *zmset-of* Δ*neg*
let *?M1* = *M* + *?*Δ
note *change* = *next-performopD*[*OF assms*(*2*)]
note *complex-change* = *next-performop-complexD*[*OF assms*(*2*)]
note *inv0* = *assms*(*1*)[*unfolded InvJustifiedII-def justified-alt*, *rule-format*]
{ fix *k q t*
assume *t*: *0* < *zcount ?M1 t*
assume *least*: ∀ *t′*<*t*. *zcount ?M1 t′* ≤ *0*
from *t* **consider** *0* < *zcount M t* | *zcount M t* ≤ *0 0* < *zcount ?*Δ *t*
by *atomize-elim* (*auto simp*: *complex-change*)
then have *supported-strong ?M1 t* ∨ (∃ *t′*<*t*. *0* < *zcount* (*c-caps c1 p*) *t′*) ∨ *zcount ?M1 t* < *zcount* (*c-caps c1 p*) *t*
proof *cases*
case *1* — *M* was already positive at *t* in *c0*
note *Mcount* = *1*
note *assms*(*1*)[*unfolded InvJustifiedII-def justified-alt*, *rule-format*, *OF Mcount*]
then consider
supported-strong M t |
¬ *supported-strong M t* ∃ *t′*<*t*. *0* < *zcount* (*c-caps c0 p*) *t′* |
∀ *t′*<*t*. *zcount* (*c-caps c0 p*) *t′* ≤ *0 zcount M t* < *zcount* (*c-caps c0 p*) *t*
by *atomize-elim auto*
then show *?thesis*
proof *cases*
case *1*
{ assume *nosupp*: ¬ *supported-strong ?M1 t*
assume ∀ *t′*<*t*. ¬ *0* < *zcount* (*c-caps c1 p*) *t′*

then have *nocaps*: $\forall t'<t.\ zcount\ (c\text{-}caps\ c1\ p)\ t' = 0$
using *InvCapsNonneg-def assms(2) assms(3) order-class.le-less perfor-mop-preserves-InvCapsNonneg* **by** *fastforce*
from *1* **obtain** *s* **where** *s*: $s < t\ zcount\ M\ s < 0\ \bigwedge s'.\ s' < s \Longrightarrow zcount\ M\ s' = 0$
unfolding *nonpos-upto-def supported-strong-def*
apply *atomize-elim*
apply (*elim exE conjE*)
apply (*drule order-zmset-exists-foundation-neg*)
apply (*elim exE*)
subgoal for - *s*
apply (*rule exI*[*of* - *s*])
apply (*rule conjI*)
using *le-less-trans* **apply** *blast*
using *less-imp-le order-trans order-class.order.not-eq-order-implies-strict* **apply** *blast*
done
done
have *count1s*: $0 \leq zcount\ ?M1\ s$
apply (*rule ccontr*)
apply (*subst* (*asm*) *not-le*)
using *nosupp*[*unfolded nonpos-upto-def supported-strong-def*, *simplified*, *rule-format*]
using *least order.strict-trans2 s(1)* **apply** *fastforce*
done
have Δinc: $0 < zcount\ ?\Delta\ s$
using *complex-change(3) count1s s(2)* **by** *auto*
have $0 < zcount\ (timestamps\ (zmset\text{-}of\ \Delta mint\text{-}msg))\ s$
using Δinc *change(2)*[*rule-format*, *of s*] *nocaps*[*rule-format*, *OF s(1)*]
unfolding *change(9) fun-upd-same zcount-union zcount-diff*
by *linarith*
then obtain *u* **where** *u*: $u < s\ 0 < zcount\ (c\text{-}caps\ c0\ p)\ u\ \forall u'<u.\ zcount\ (c\text{-}caps\ c0\ p)\ u' = 0$
apply *atomize-elim*
apply (*drule pos-image-zmset-obtain-pre*[*rotated*])
apply *simp*
apply *clarify*
subgoal for p'
apply *simp*
apply (*drule change(4)*[*unfolded minting-msg-def*, *rule-format*])
apply *simp*
apply (*elim exE conjE*)
apply (*drule order-zmset-exists-foundation*)
apply *clarsimp*
subgoal for s' *u*
apply (*rule exI*[*of* - *u*])
apply *clarsimp*
using *assms(3)*[*unfolded InvCapsNonneg-def*, *rule-format*]
apply (*metis le-less-trans order-class.order.not-eq-order-implies-strict*)

```
            done
          done
        done
      have count1u: zcount ?M1 u < 0
        using complex-change(4)[of u] nocaps[unfolded change(9) fun-upd-same]
order.strict-trans[OF u(1) s(1)] s(3)[OF u(1)] u(2) u(3)
        by auto
      then have nonpos-upto ?M1 u
        unfolding nonpos-upto-def
          using least order.strict-implies-order order.strict-trans1 s(1) u(1) by
blast
      then have supported-strong ?M1 t
        using count1u order.strict-trans s(1) u(1) supported-strong-def by blast
      with nosupp have False
        by blast
    }
    then show ?thesis
      by blast
  next
    case 2
    { assume nosupp: ¬ supported-strong ?M1 t
      assume ∀ t'<t. ¬ 0 < zcount (c-caps c1 p) t'
      then have nocaps: ∀ t'<t. zcount (c-caps c1 p) t' = 0
        using InvCapsNonneg-def assms(2) assms(3) order-class.le-less perfor-
mop-preserves-InvCapsNonneg by fastforce
       from 2(2) obtain s where s: s < t 0 < zcount (c-caps c0 p) s ∀ s'<s.
zcount (c-caps c0 p) s' = 0
        apply atomize-elim
        apply (elim exE conjE)
        apply (drule order-zmset-exists-foundation)
        apply (elim exE conjE)
        subgoal for - s
          apply (rule exI[of - s])
          apply (rule conjI, simp)
          apply (rule conjI, simp)
          apply (intro allI impI)
          apply (rule ccontr)
          subgoal
            using assms(3)[unfolded InvCapsNonneg-def, rule-format]
            apply auto
            by (metis order-class.le-less zcount-ne-zero-iff)
          done
        done
      have Δcounts:
        ⋀s. s < t ⟹ count Δneg s = zcount (c-caps c0 p) s
        ⋀s. s < t ⟹ count Δmint-self s = 0
        ⋀p s'. s' ≤ s ⟹ count Δmint-msg (p,s') = 0
        subgoal for s'
        using change(2) nocaps s(1) order-class.order.not-eq-order-implies-strict
```

```
          by (fastforce simp: change(9))
        subgoal for s'
          using nocaps[rule-format, of s']
         by (simp add: change(9) ‹⋀s'. s' < t ⟹ int (count Δneg s') = zcount
(c-caps c0 p) s'›)
        subgoal for p s'
          using change(4)[unfolded minting-msg-def, rule-format, of (p,s')] s(3)
          by (force intro: ccontr)
        done
      have caps-le-ii0: zcount (c-caps c0 p) s ≤ zcount M s
      proof (rule ccontr)
        assume nle: ¬ zcount (c-caps c0 p) s ≤ zcount M s
        have zcount ?M1 s < 0
          unfolding complex-change(3)
            using complex-change(4) nle s(3) by (auto simp: Δcounts(1,2)[OF
s(1)])
        then show False
          using s(1) least
              by (force dest!: nosupp[unfolded supported-strong-def, simplified,
rule-format] simp: nonpos-upto-def)
      qed
      with s(2) have count0s: 0 < zcount M s
        by auto
      have False
        using inv0[OF count0s]
        apply (elim disj3-split)
        using 2(1) order.strict-trans s(1) supported-strong-def apply blast
        using s(3) apply auto []
        using caps-le-ii0 apply linarith
        done
    }
    then show ?thesis
      by blast
  next
    case 3
    then show ?thesis
      apply (intro disjI2)
      unfolding complex-change(3) change(9)
      apply (simp only: if-P zcount-union zcount-diff)
      apply (subst complex-change(4)[of t])
      using assms(3)[unfolded InvCapsNonneg-def, rule-format]
      apply (simp-all add: antisym)
      done
  qed
 next — Adding Δ made M positive at t in c1
  case 2
  { assume nosupp: ¬ supported-strong ?M1 t
    assume ∀t'<t. ¬ 0 < zcount (c-caps c1 p) t'
    then have nocaps: ∀t'<t. zcount (c-caps c1 p) t' = 0
```

using *InvCapsNonneg-def assms(2) assms(3) order-class.le-less perfor-mop-preserves-InvCapsNonneg* **by** *metis*
assume ¬ *zcount ?M1 t < zcount* (*c-caps c1 p*) *t*
then have *caps-le*: *zcount* (*c-caps c1 p*) *t* ≤ *zcount ?M1 t*
by *linarith*
from *2* **have** *count* Δ*neg t < zcount ?*Δ*pos t*
by *auto*
then have *0 < count* Δ*mint-self t* ∨ *0 < zcount* (*timestamps* (*zmset-of* Δ*mint-msg*)) *t*
by (*metis 2*(*2*) *add.commute add.left-neutral not-gr-zero of-nat-0 zero-lt-diff zcount-diff zcount-of-mset zcount-union zcount-zmset-of-nonneg*)
then obtain *s* **where** *s*: *s* ≤ *t 0 < zcount* (*c-caps c0 p*) *s* ∀ *s'*<*s. zcount* (*c-caps c0 p*) *s'* = *0*
apply *atomize-elim*
apply (*elim disjE*)
subgoal
apply *simp*
apply (*drule change*(*3*)[*unfolded minting-self-def*, *rule-format*])
apply (*elim exE conjE*)
apply (*drule order-zmset-exists-foundation*)
apply *clarsimp*
subgoal for *s' u*
apply (*rule exI*[*of - u*])
apply *clarsimp*
using *assms*(*3*)[*unfolded InvCapsNonneg-def*, *rule-format*]
apply (*metis order.trans order-class.order.not-eq-order-implies-strict*)
done
done
subgoal
apply (*drule pos-image-zmset-obtain-pre*[*rotated*])
apply *simp*
apply *clarify*
subgoal for *p'*
apply *simp*
apply (*drule change*(*4*)[*unfolded minting-msg-def*, *rule-format*])
apply *simp*
apply (*elim exE conjE*)
apply (*drule order-zmset-exists-foundation*)
apply *clarsimp*
subgoal for *s' u*
apply (*rule exI*[*of - u*])
apply *clarsimp*
using *assms*(*3*)[*unfolded InvCapsNonneg-def*, *rule-format*]
apply (*metis order.strict-trans1 less-imp-le order-class.order.not-eq-order-implies-strict*)
done
done
done
done
have Δ*counts*:

```
        ⋀s. s < t ⟹ count Δneg s = zcount (c-caps c0 p) s
        ⋀s. s < t ⟹ count Δmint-self s = 0
        ⋀p s'. s' ≤ s ⟹ count Δmint-msg (p,s') = 0
        subgoal for s'
         using change(2) nocaps s(1) order-class.order.not-eq-order-implies-strict
         by (fastforce simp: change(9))
        subgoal for s'
         using nocaps[rule-format, of s']
         by (simp add: change(9) ‹⋀s'. s' < t ⟹ int (count Δneg s') = zcount
(c-caps c0 p) s'›)
        subgoal for p s'
         using change(4)[unfolded minting-msg-def, rule-format, of (p,s')] s(3)
         by (force intro: ccontr)
        done
       { assume less: s < t
        have caps-le-ii0: zcount (c-caps c0 p) s ≤ zcount M s
        proof (rule ccontr)
         assume nle: ¬ zcount (c-caps c0 p) s ≤ zcount M s
         have zcount ?M1 s < 0
          unfolding complex-change(3)
           using complex-change(4) nle s(3) by (auto simp: Δcounts(1,2)[OF
less])
         then show False
          using less least order.strict-trans2
              by (force dest!: nosupp[unfolded supported-strong-def, simplified,
rule-format] simp: nonpos-upto-def)
        qed
        with s(2) have count0s: 0 < zcount M s
         by auto
        have False
         using inv0[OF count0s]
         apply (elim disj3-split)
         subgoal
         proof −
          assume supported-strong M s
          then obtain u where u: u < s zcount M u < 0
           unfolding supported-strong-def
           by blast
          have 0 ≤ zcount ?M1 u
              using least nosupp[unfolded supported-strong-def nonpos-upto-def,
simplified, rule-format, of u] order.strict-trans[OF u(1) less]
           by fastforce
          then have 0 < zcount ?Δpos u
           using Δcounts(1)[of u] s(3) u(1) u(2) less by force
         then have 0 < count Δmint-self u ∨ 0 < zcount (timestamps (zmset-of
Δmint-msg)) u
          using gr0I by fastforce
         then obtain u' where u' ≤ u 0 < zcount (c-caps c0 p) u'
          apply atomize-elim
```

```
            apply (elim disjE)
            subgoal
              apply simp
              apply (drule change(3)[unfolded minting-self-def, rule-format])
              using s(1) s(2) apply blast
              done
            subgoal
              apply (drule pos-image-zmset-obtain-pre[rotated])
               apply simp
              apply clarify
              subgoal for p'
                apply simp
                apply (drule change(4)[unfolded minting-msg-def, rule-format])
                using order.strict-iff-order apply auto
                done
              done
            done
          then show False
            using s(3) u(1) by auto
        qed
        using s(3) apply auto []
        using caps-le-ii0 apply linarith
        done
    }
    moreover
    { assume eq: s = t
      have count0t: 0 < zcount M t
        using eq caps-le change(9) complex-change(3,4) s(2,3) by auto
      have False
        using 2(1) count0t by auto
    }
    ultimately have False
      using order.not-eq-order-implies-strict s(1) by blast
  }
  then show ?thesis
    by blast
 qed
}
then show justified (c-caps c1 p) ?M1
  by (intro justified-leastI) blast
qed

lemma InvJustifiedII-implies-InvJustifiedGII:
  assumes InvJustifiedII c
    and   InvCapsNonneg c
  shows   InvJustifiedGII c
  using assms
  unfolding
    InvJustifiedGII-def
```

```
    InvJustifiedII-def
    GlobalIncomingInfo-def
    records-def
    InvCapsNonneg-def
  by (auto intro!: justified-add-records justified-sum)

lemma init-config-implies-InvJustifiedII: init-config c ⟹ InvJustifiedII c
  by (simp add: init-config-def InvJustifiedII-def justified-alt IncomingInfo-def)

lemma performop-preserves-InvJustifiedII:
  assumes InvJustifiedII c0
    and   next-performop' c0 c1 p Δneg Δmint-msg Δmint-self
    and   InvCapsNonneg c0
  shows   InvJustifiedII c1
  unfolding InvJustifiedII-def
  apply clarify
  subgoal for k p' q
    apply (cases p'=p)
    subgoal
      unfolding next-performop-complexD[OF assms(2)]
      apply (simp only: if-P)
      apply (rule next-performop'-preserves-justified[
            OF assms(1)[unfolded InvJustifiedII-def, rule-format, of p k q],
            OF assms(2,3)])
      done
    subgoal
      unfolding
        next-performopD[OF assms(2)]
        next-performop-complexD[OF assms(2)]
      using assms(1)[unfolded InvJustifiedII-def]
      by simp
    done
  done

lemma sendupd-preserves-InvJustifiedII:
  assumes InvJustifiedII c0
    and   next-sendupd' c0 c1 p tt
  shows   InvJustifiedII c1
 unfolding InvJustifiedII-def justified-alt supported-strong-def next-sendupdD(6)[OF
assms(2)]
  apply (intro allI)
  subgoal for k p' q t
    apply (cases k ≤ length (c-msg c0 p q))
    subgoal
      apply (drule next-sendupd-complexD(4)[OF assms(2), of - - p'])
    apply (auto dest: assms(1)[unfolded InvJustifiedII-def justified-alt supported-strong-def,
rule-format])
      done
    subgoal
```

apply (*subst* (*asm*) *not-le*)
apply (*subst* (*1 2 3 4*) *next-sendupd-complexD*(*5*)[*OF assms*(*2*), *of - - p'*])
apply *simp*
apply *simp*
apply (*cases p'=p*)
subgoal
apply *rule*
apply (*subst* (*asm*) *if-P*)
apply *simp*
apply (*subst* (*1 2 3*) *if-P*)
apply *simp*
apply *simp*
unfolding *IncomingInfo-def*
apply (*subst* (*asm*) *drop-all*)
apply *simp*
apply (*subst drop-all*)
apply *simp*
apply (*simp del*: *zcount-diff*)
— The justified condition ensures that anything remaining in temp satisfies this invariant
apply (*drule next-sendupdD*(*2*)[*OF assms*(*2*), *unfolded justified-alt supported-strong-def*, *rule-format*])
apply *simp*
done
subgoal
by (*auto intro*!: *assms*(*1*)[*unfolded InvJustifiedII-def justified-alt supported-strong-def*, *rule-format*])
done
done
done

lemma *recvupd-preserves-InvJustifiedII*:
assumes *InvJustifiedII c0*
and *next-recvupd' c0 c1 p q*
shows *InvJustifiedII c1*
using *assms*(*1*)
unfolding
InvJustifiedII-def
next-recvupd-complexD[*OF assms*(*2*)]
next-recvupdD[*OF assms*(*2*)]
by *auto*

lemma *recvcap-preserves-InvJustifiedII*:
assumes *InvJustifiedII c0*
and *next-recvcap' c0 c1 p t*
shows *InvJustifiedII c1*
unfolding *InvJustifiedII-def justified-alt supported-strong-def next-recvcap-complexD*[*OF assms*(*2*)] *next-recvcapD*(*5*)[*OF assms*(*2*)]
by (*auto dest*!: *assms*(*1*)[*unfolded InvJustifiedII-def justified-alt supported-strong-def*,

rule-format])

lemma *next′-preserves-InvJustifiedII*:
InvCapsNonneg c0 $\Longrightarrow$ *InvJustifiedII c0* $\Longrightarrow$ *next′ c0 c1* $\Longrightarrow$ *InvJustifiedII c1*
using
next′-def
performop-preserves-InvJustifiedII
recvcap-preserves-InvJustifiedII
recvupd-preserves-InvJustifiedII
sendupd-preserves-InvJustifiedII
by *blast*

lemma *alw-InvJustifiedII*: *spec s* $\Longrightarrow$ *alw* (*holds InvJustifiedII*) *s*
apply (*frule alw-InvCapsNonneg*)
unfolding *spec-def*
apply (*elim conjE*)
apply (*subst* (*asm*) *holds.simps*)
apply (*drule init-config-implies-InvJustifiedII*)
apply (*coinduction arbitrary*: *s rule*: *alw.coinduct*)
apply (*subst* (*asm*) (*1 2*) *alw-nxt*)
apply *clarsimp*
using *next′-preserves-InvJustifiedII* **apply** *blast*
done

lemma *alw-InvJustifiedGII*: *spec s* $\Longrightarrow$ *alw* (*holds InvJustifiedGII*) *s*
apply (*frule alw-InvJustifiedII*)
apply (*drule alw-InvCapsNonneg*)
apply (*simp add*: *alw-iff-sdrop InvJustifiedII-implies-InvJustifiedGII*)
done

4.3.6 InvTempJustified

definition *InvTempJustified* **where**
InvTempJustified c = ($\forall$ *p*. *justified* (*c-caps c p*) (*c-temp c p*))

lemma *init-config-implies-InvTempJustified*: *init-config c* $\Longrightarrow$ *InvTempJustified c*
unfolding *init-config-def InvTempJustified-def*
using *justified-add-records*[*OF justified-empty*]
by *auto*

lemma *recvcap-preserves-InvTempJustified*:
assumes *InvTempJustified c0*
and *next-recvcap′ c0 c1 p t*
shows *InvTempJustified c1*
using *assms*(*1*)[*unfolded InvTempJustified-def*, *rule-format*, *of p*] *assms*(*1*)
unfolding *next-recvcapD*[*OF assms*(*2*)] *InvTempJustified-def*
by − (*frule justified-add-records*[*of* - - {#*t*#}$_z$], *auto*)

lemma *recvupd-preserves-InvTempJustified*:

assumes *InvTempJustified c0*
and *next-recvupd′ c0 c1 p t*
shows *InvTempJustified c1*
using *assms*(*1*)
unfolding *next-recvupdD*[*OF assms*(*2*)] *InvTempJustified-def*
by *assumption*

lemma *sendupd-preserves-InvTempJustified*:
assumes *InvTempJustified c0*
and *next-sendupd′ c0 c1 p tt*
shows *InvTempJustified c1*
using *assms*(*1*)
unfolding *next-sendupdD*[*OF assms*(*2*)] *InvTempJustified-def*
using *next-sendupdD*(*2*)[*OF assms*(*2*)]
by *auto*

lemma *performop-preserves-InvTempJustified*:
assumes *InvTempJustified c0*
and *next-performop′ c0 c1 p* Δ*neg* Δ*mint-msg* Δ*mint-self*
and *InvCapsNonneg c0*
shows *InvTempJustified c1*
unfolding *InvTempJustified-def*
apply *clarify*
subgoal for *p′*
apply (*cases p′=p*)
subgoal
unfolding *next-performopD*(*5*)[*OF assms*(*2*)] *fun-upd-apply*
apply (*simp only*: *if-P*)
apply (*rule next-performop′-preserves-justified*[
OF assms(*1*)[*unfolded InvTempJustified-def*, *rule-format*, *of p*],
OF assms(*2*,*3*)])
done
subgoal
unfolding
next-performopD[*OF assms*(*2*)]
next-performop-complexD[*OF assms*(*2*)]
using *assms*(*1*)[*unfolded InvTempJustified-def*]
by *simp*
done
done

lemma *next′-preserves-InvTempJustified*:
InvCapsNonneg c0 ⟹ *InvTempJustified c0* ⟹ *next′ c0 c1* ⟹ *InvTempJustified c1*
using
next′-def
performop-preserves-InvTempJustified
recvcap-preserves-InvTempJustified
recvupd-preserves-InvTempJustified

sendupd-preserves-InvTempJustified
by *blast*

lemma *alw-InvTempJustified*: *spec s* $\Longrightarrow$ *alw* (*holds InvTempJustified*) *s*
apply (*frule alw-InvCapsNonneg*)
unfolding *spec-def*
apply (*elim conjE*)
apply (*subst* (*asm*) *holds.simps*)
apply (*drule init-config-implies-InvTempJustified*)
apply (*coinduction arbitrary*: *s rule*: *alw.coinduct*)
apply (*subst* (*asm*) (*1 2*) *alw-nxt*)
apply *clarsimp*
using *next'-preserves-InvTempJustified* **apply** *blast*
done

4.3.7 InvGlobNonposImpRecordsNonpos

InvGlobNonposImpRecordsNonpos states that each processor's *c-glob c q* is a conservative approximation of *records c*.

definition *InvGlobNonposImpRecordsNonpos* :: (*'p* :: *finite*, *'a*) *configuration* $\Rightarrow$ *bool* **where**
InvGlobNonposImpRecordsNonpos c = ($\forall$ *t q*. *nonpos-upto* (*c-glob c q*) *t* $\longrightarrow$ *nonpos-upto* (*records c*) *t*)

definition *InvGlobVacantImpRecordsVacant* :: (*'p* :: *finite*, *'a*) *configuration* $\Rightarrow$ *bool* **where**
InvGlobVacantImpRecordsVacant c = ($\forall$ *t q*. *GlobVacantUpto c q t* $\longrightarrow$ *RecordsVacantUpto c t*)

lemma *invs-imp-InvGlobNonposImpRecordsNonpos*:
assumes *InvJustifiedGII c*
and *InvRecordCount c*
and *InvRecordsNonneg c*
shows *InvGlobNonposImpRecordsNonpos c*
unfolding *InvGlobNonposImpRecordsNonpos-def*
apply (*rule ccontr*)
apply (*clarsimp simp*: *nonpos-upto-def*)
subgoal for *t q u*
proof −
let *?GII* = *GlobalIncomingInfoAt c q*
assume *gvu*: $\forall$ *sa*$\leq$*t*. *zcount* (*c-glob c q*) *sa* $\leq$ *0*
assume *uleqt*: *u* $\leq$ *t*
assume $\neg$ *zcount* (*records c*) *u* $\leq$ *0*
— u is pointstamp that violates *InvGlobNonposImpRecordsNonpos*
with *assms*(*2*) **have** *u*: *0* < *zcount* (*records c*) *u*
by *linarith*
— u' is the least pointstamp with positive *records*
with *uleqt* **obtain** *u'* **where** *u'*: *0* < *zcount* (*records c*) *u'* $\forall$ *u*. *0* < *zcount* (*records c*) *u* $\longrightarrow$ $\neg$ *u* < *u' u'* $\leq$ *t*

```
    using order-zmset-exists-foundation[OF u] by auto
      — from the records count we know that GII also has positive count
   from u'(1,3) assms(2) gvu have pos-gii: 0 < zcount ?GII u'
    by (simp add: InvRecordCount-def) (smt (verit) zcount-union)
      — Case distinction on which part of the partition GII's u is in
   { — Original proof from Abadi paper, change is justified by uprightness
    assume supported-strong ?GII u'
      — uprightness gives us a lesser pointstamp with negative count in GII..
    then obtain v where v: v ≤ u' zcount ?GII v < 0
      using order.strict-implies-order supported-strong-def by blast
        — ..which is also negative in records..
    with u'(3) v(1) assms(2) have zcount (records c) v < 0
       by (smt (verit, ccfv-threshold) InvRecordCount-def gvu local.order-trans
zcount-union)
        — ..contradicting InvNrecNonneg
    with assms(3) have False
      unfolding InvRecordsNonneg-def
      using order-class.leD by blast
   }
   moreover
   { — Change is justified by strictly lesser pointstamp in records
    assume ∃t'<u'. 0 < zcount (records c) t'
      — v is a strictly lesser positive count in records..
    then obtain v where v: v < u' 0 < zcount (records c) v
      by auto
        — ..which contradicts the fact that we obtained u' as the least, positive
pointstamp in records
    with u'(2) have False
      by auto
   }
   moreover
   { — Change is justified by records
    assume zcount ?GII u' < zcount (records c) u'
    then have 0 < zcount (c-glob c q) u'
      by (simp add: assms(2)[unfolded InvRecordCount-def, rule-format, of q])
    then have False
      using gvu u'(3) by auto
   }
   ultimately show False
     using pos-gii assms(1)[unfolded InvJustifiedGII-def justified-alt, rule-format,
OF pos-gii]
    by auto
  qed
  done
```

InvGlobVacantImpRecordsVacant is the one proved in the Abadi paper. We prove InvGlobNonposImpRecordsNonpos, which implies this.

```
lemma invs-imp-InvGlobVacantImpRecordsVacant:
  assumes InvJustifiedGII c
```

and *InvRecordCount c*
and *InvRecordsNonneg c*
shows *InvGlobVacantImpRecordsVacant c*
proof −
{ fix *p x*
assume *GlobVacantUpto c p x*
then have *GlobNonposUpto c p x*
unfolding *nonpos-upto-def vacant-upto-def* **by** *simp*
note *invs-imp-InvGlobNonposImpRecordsNonpos*[*OF assms*, *unfolded InvGlobNonposImpRecordsNonpos-def*, *rule-format*, *OF this*]
then have *RecordsVacantUpto c x*
using *assms*(*3*)
unfolding *nonpos-upto-def vacant-upto-def InvRecordsNonneg-def*
by (*simp add*: *order-class.order.antisym*)
}
then show *?thesis*
unfolding *InvGlobVacantImpRecordsVacant-def* **by** *simp*
qed

lemma *alw-InvGlobNonposImpRecordsNonpos*: *spec s* ⟹ *alw* (*holds InvGlobNonposImpRecordsNonpos*) *s*
apply (*frule alw-InvJustifiedGII*)
apply (*frule alw-InvRecordCount*)
apply (*drule alw-InvRecordsNonneg*)
apply (*simp add*: *alw-iff-sdrop invs-imp-InvGlobNonposImpRecordsNonpos*)
done

lemma *alw-InvGlobVacantImpRecordsVacant*: *spec s* ⟹ *alw* (*holds InvGlobVacantImpRecordsVacant*) *s*
apply (*frule alw-InvGlobNonposImpRecordsNonpos*)
apply (*frule alw-InvJustifiedGII*)
apply (*frule alw-InvRecordCount*)
apply (*drule alw-InvRecordsNonneg*)
apply (*simp add*: *alw-iff-sdrop invs-imp-InvGlobVacantImpRecordsVacant*)
done

4.3.8 SafeGlobVacantUptoImpliesStickyNrec

This is the main safety property proved in the Abadi paper.

lemma *invs-imp-SafeGlobVacantUptoImpliesStickyNrec*:
SafeRecordsMono s ⟹ *holds InvGlobVacantImpRecordsVacant s* ⟹ *SafeGlobVacantUptoImpliesStickyNrec s*
by (*simp add*: *InvGlobVacantImpRecordsVacant-def SafeRecordsMono-def SafeGlobVacantUptoImpliesStickyNrec-def*)

lemma *alw-SafeGlobVacantUptoImpliesStickyNrec*:
spec s ⟹ *alw SafeGlobVacantUptoImpliesStickyNrec s*
by (*meson alw-iff-sdrop invs-imp-SafeGlobVacantUptoImpliesStickyNrec alw-SafeRecordsMono alw-InvGlobVacantImpRecordsVacant*)

4.3.9 InvGlobNonposEqVacant

The least pointstamps in glob are always positive, i.e. *nonpos-upto* and *vacant-upto* on glob are equivalent.

definition *InvGlobNonposEqVacant* **where**
InvGlobNonposEqVacant c = (∀ *q t*. *GlobVacantUpto c q t* = *GlobNonposUpto c q t*)

lemma *invs-imp-InvGlobNonposEqVacant*:
assumes *InvRecordCount c*
and *InvJustifiedGII c*
and *InvRecordsNonneg c*
shows *InvGlobNonposEqVacant c*
proof −
note *safe* = *invs-imp-InvGlobNonposImpRecordsNonpos*[*OF assms*(*2*,*1*,*3*), *unfolded InvGlobNonposImpRecordsNonpos-def nonpos-upto-def*, *THEN spec2*, *THEN mp*]
note *nonneg* = *assms*(*3*)[*unfolded InvRecordsNonneg-def*, *rule-format*]
note *count* = *assms*(*1*)[*unfolded InvRecordCount-def*, *rule-format*]
{ fix *q t*
assume *np*: *GlobNonposUpto c q t*
assume *nv*: ¬ *GlobVacantUpto c q t*
— Obtain the least, negative pointstamp in glob
obtain *s* **where** *s*: *s* ≤ *t zcount* (*c-glob c q*) *s* < *0* ∀ *s′*<*s*. *zcount* (*c-glob c q*) *s′* = *0*
apply *atomize-elim*
using *nv*[*unfolded vacant-upto-def*]
apply *clarsimp*
apply (*drule elem-order-zmset-exists-foundation*)
apply *clarsimp*
subgoal for - *s*
apply (*rule exI*[*of* - *s*])
apply (*meson order.ordering-axioms nonpos-upto-def np order-class.le-less order-trans zcount-ne-zero-iff*)
done
done
— No records *s′* ≤ *s* can exist, since that would violate InvGlobNonposImpRecordsNonpos
have *norec*: *s′* ≤ *s* ⟹ *zcount* (*records c*) *s′* = *0* **for** *s′*
by (*metis* (*full-types*) *nonneg nonpos-upto-def np order-class.antisym-conv order-trans s*(*1*) *safe*)
— Hence GII must be positive at *s*
have *gii*: *zcount* (*GlobalIncomingInfoAt c q*) *s* > *0*
using *count*[*of q*] *s*(*2*) *norec*[*of s*, *simplified*]
by (*metis add.commute less-add-same-cancel1 zcount-union*)
— which means it must be justified by one of these three disjuncts
then consider
supported-strong (*GlobalIncomingInfoAt c q*) *s* |
∃ *t′*<*s*. *0* < *zcount* (*records c*) *t′* |

zcount (*GlobalIncomingInfoAt c q*) *s* < *zcount* (*records c*) *s*
using *assms*(*2*)[*unfolded InvJustifiedGII-def justified-alt*, *rule-format*, *OF gii*]
by *atomize-elim auto*
then have *False*
proof *cases*
case *1* — *s* can't be *supported-strong*, since either glob or records would have to be positive at the support
then show *False*
using *norec count s*(*3*)
unfolding *supported-strong-def*
by (*metis* (*full-types*) *add.commute add.left-neutral less-le preorder-class.less-irrefl zcount-union*)
next
case *2* — no lesser capabilities exist
then show *?thesis*
using *norec s*(*2*,*3*) *order.order-iff-strict* **by** *auto*
next
case *3* — no capabilities at *s* exist
then show *?thesis*
unfolding *norec*[*of s*, *simplified*]
using *gii* **by** *auto*
qed
}
then show *?thesis*
unfolding *InvGlobNonposEqVacant-def*
apply (*intro allI*)
apply (*rule iffI*)
apply (*simp add*: *nonpos-upto-def vacant-upto-def*)
apply *auto*
done
qed

lemma *alw-InvGlobNonposEqVacant*: *spec s* ⟹ *alw* (*holds InvGlobNonposEqVacant*) *s*
using
alw-InvJustifiedGII
alw-InvRecordCount
alw-InvRecordsNonneg
invs-imp-InvGlobNonposEqVacant
by (*metis alw-iff-sdrop holds.elims*(*2*) *holds.elims*(*3*))

4.3.10 InvInfoJustifiedWithII and InvInfoJustifiedWithGII

definition *InvInfoJustifiedWithII* **where**
InvInfoJustifiedWithII c = (∀ *k p q*. *justified-with* (*c-caps c p*) (*InfoAt c k p q*) (*IncomingInfo c* (*k*+*1*) *p q*))

definition *InvInfoJustifiedWithGII* **where**
InvInfoJustifiedWithGII c = (∀ *k p q*. *justified-with* (*records c*) (*InfoAt c k p q*)

(*GlobalIncomingInfo c (k+1) p q*))

lemma *init-config-implies-InvInfoJustifiedWithII*: *init-config c* $\Longrightarrow$ *InvInfoJustifiedWithII c*
unfolding *init-config-def InvInfoJustifiedWithII-def justified-with-def InfoAt-def*
by *auto*

This proof relies heavily on the addition properties summarized in the lemma $[\![$*justified-with* (*c-caps ?c0.0 ?p*) *?M ?N*; *next-performop′ ?c0.0 ?c1.0 ?p ?*Δ*neg ?*Δ*mint-msg ?*Δ*mint-self*; $\forall$ *t. 0* $\leq$ *zcount* (*c-caps ?c0.0 ?p*) *t*; *justified* (*c-caps ?c0.0 ?p*) *?N*$]\!]$ $\Longrightarrow$ *justified-with* (*c-caps ?c0.0 ?p* + *zmset-of ?*Δ*mint-self* − *zmset-of ?*Δ*neg*) *?M* (*?N* + *zmset-of ?*Δ*mint-self* + *timestamps* (*zmset-of ?*Δ*mint-msg*) − *zmset-of ?*Δ*neg*)

lemma *performop-preserves-InvInfoJustifiedWithII*:
assumes *InvInfoJustifiedWithII c0*
and *next-performop′ c0 c1 p′* Δ*neg* Δ*mint-msg* Δ*mint-self*
and *InvJustifiedII c0*
and *InvCapsNonneg c0*
shows *InvInfoJustifiedWithII c1*
unfolding *InvInfoJustifiedWithII-def*
apply (*intro allI*)
subgoal for *k p q*
apply (*cases p′=p*)
subgoal
unfolding *next-performop-complexD*[*OF assms*(*2*)] *next-performopD*[*OF assms*(*2*)]
apply (*simp only*: *add-diff-eq if-P fun-upd-same*)
apply (*subst* (*4*) *add.commute*)
apply (*subst add.assoc*[*symmetric*])
apply (*rule next-performop′-preserves-justified-with*)
using *assms*(*1*)[*unfolded InvInfoJustifiedWithII-def*] **apply** *simp*
using *assms*(*2*) **apply** *simp*
using *assms*(*4*)[*unfolded InvCapsNonneg-def*] **apply** *simp*
using *assms*(*3*)[*unfolded InvJustifiedII-def*] **apply** *simp*
done
subgoal
using *assms*(*1*)[*unfolded InvInfoJustifiedWithII-def justified-with-def*]
by (*auto simp*: *not-less justified-with-def next-performop-complexD*[*OF assms*(*2*)] *next-performopD*[*OF assms*(*2*)])
done
done

lemma *sendupd-preserves-InvInfoJustifiedWithII*:
assumes *InvInfoJustifiedWithII c0*
and *next-sendupd′ c0 c1 p′ tt*
and *InvTempJustified c0*
shows *InvInfoJustifiedWithII c1*
unfolding *InvInfoJustifiedWithII-def*
proof (*intro allI impI*)

```
  fix k :: nat and p q
  note complex = next-sendupd-complexD[OF assms(2)]
  note change = next-sendupdD[OF assms(2)]
  consider
    p' = p k < length (c-msg c0 p q) |
    p' = p k = length (c-msg c0 p q) |
    p' = p k > length (c-msg c0 p q) |
    p' ≠ p
    by atomize-elim auto
  then show justified-with (c-caps c1 p) (InfoAt c1 k p q) (IncomingInfo c1 (k+1)
p q)
  proof cases
    case 1
    then show ?thesis
     by (metis InvInfoJustifiedWithII-def Suc-eq-plus1 Suc-le-eq assms(1) change(6)
complex(4,7) order-class.less-le)
  next
    case 2
    have temp0: c-temp c0 p = InfoAt c1 k p q + c-temp c1 p
      unfolding complex change
      using 2 by (auto simp: algebra-simps)
    have pi: PositiveImplies (InfoAt c1 k p q) (c-temp c0 p)
      unfolding PositiveImplies-def complex
      using 2(2) by (auto simp: InfoAt-def)
    have iitemp: IncomingInfo c1 (k+1) p q = c-temp c1 p
      unfolding IncomingInfo-def
      using 2 by (simp add: change)
    show ?thesis
      apply (rule PositiveImplies-justified-with)
      unfolding iitemp temp0[symmetric]
      unfolding change
      using assms(3) apply (simp add: InvTempJustified-def)
      using pi apply simp
      done
  next
    case 3
    then show ?thesis
    by (metis add-cancel-right-left complex(7) justified-with-leastI preorder-class.less-asym
InfoAt-def zcount-union)
  next
    case 4
    then show ?thesis
      unfolding complex change
      using assms(1)[unfolded InvInfoJustifiedWithII-def, rule-format]
      apply simp
      done
  qed
qed
```

lemma *recvupd-preserves-InvInfoJustifiedWithII*:
assumes *InvInfoJustifiedWithII c0*
and *next-recvupd′ c0 c1 p q*
shows *InvInfoJustifiedWithII c1*
using *assms*(*1*)
unfolding
InvInfoJustifiedWithII-def
next-recvupd-complexD[*OF assms*(*2*)]
next-recvupdD[*OF assms*(*2*)]
by *auto*

lemma *recvcap-preserves-InvInfoJustifiedWithII*:
assumes *InvInfoJustifiedWithII c0*
and *next-recvcap′ c0 c1 p t*
shows *InvInfoJustifiedWithII c1*
using *assms*(*1*)
unfolding
InvInfoJustifiedWithII-def
next-recvcap-complexD[*OF assms*(*2*)]
next-recvcapD[*OF assms*(*2*)]
by *simp*

lemma *invs-imp-InvInfoJustifiedWithGII*:
assumes *InvInfoJustifiedWithII c*
and *InvJustifiedII c*
and *InvCapsNonneg c*
shows *InvInfoJustifiedWithGII c*
unfolding *InvInfoJustifiedWithGII-def*
apply *clarify*
subgoal for *k p q*
unfolding *GlobalIncomingInfo-def records-def*
apply (*rule justified-with-add-records*)
subgoal
apply (*rule justified-with-sum*[**where** $x = p$])
apply *simp*
apply *simp*
apply *simp*
using *assms*(*1*)[*unfolded InvInfoJustifiedWithII-def*, *rule-format*, *of p k q*]
apply *simp*
using *assms*(*2*)[*unfolded InvJustifiedII-def*] **apply** *simp*
using *assms*(*3*)[*unfolded InvCapsNonneg-def*] **apply** *simp*
done
apply *simp*
done
done

lemma *next′-preserves-InvInfoJustifiedWithII*:
assumes *InvInfoJustifiedWithII c0*
and *next′ c0 c1*

and *InvCapsNonneg c0*
and *InvJustifiedII c0*
and *InvTempJustified c0*
shows *InvInfoJustifiedWithII c1*
using *assms* **unfolding** *next′-def*
apply (*elim disjE exE*)
apply (*drule* (*4*) *performop-preserves-InvInfoJustifiedWithII*)
apply (*drule* (*3*) *sendupd-preserves-InvInfoJustifiedWithII*)
apply (*drule* (*2*) *recvupd-preserves-InvInfoJustifiedWithII*)
apply (*drule* (*2*) *recvcap-preserves-InvInfoJustifiedWithII*)
apply *simp*
done

lemma *alw-InvInfoJustifiedWithII*: *spec s* ⟹ *alw* (*holds InvInfoJustifiedWithII*) *s*
apply (*frule alw-InvCapsNonneg*)
apply (*frule alw-InvJustifiedII*)
apply (*frule alw-InvTempJustified*)
unfolding *spec-def*
apply (*elim conjE*)
apply (*subst* (*asm*) *holds.simps*)
apply (*drule init-config-implies-InvInfoJustifiedWithII*)
apply (*coinduction arbitrary*: *s rule*: *alw.coinduct*)
apply (*subst* (*asm*) (*1 2 3*) *alw-nxt*)
apply *clarsimp*
using *next′-preserves-InvInfoJustifiedWithII*
apply *blast*
done

lemma *alw-InvInfoJustifiedWithGII*: *spec s* ⟹ *alw* (*holds InvInfoJustifiedWithGII*) *s*
by (*metis alw-InvCapsNonneg alw-InvInfoJustifiedWithII alw-InvJustifiedII alw-iff-sdrop holds.elims*(*2,3*) *invs-imp-InvInfoJustifiedWithGII*)

4.3.11 SafeGlobMono and InvMsgInGlob

The records in glob are monotonic. This implies the corollary InvMsgInGlob; No incoming message carries a timestamp change that would cause glob to regress.

definition *SafeGlobMono* **where**
SafeGlobMono c0 c1 = (∀ *p t*. *GlobVacantUpto c0 p t* ⟶ *GlobVacantUpto c1 p t*)

definition *InvMsgInGlob* **where**
InvMsgInGlob c = (∀ *p q t*. *c-msg c p q* ≠ [] ⟶ *t* ∈#$_z$ *hd* (*c-msg c p q*) ⟶ (∃ *t′*≤*t*. *0* < *zcount* (*c-glob c q*) *t′*))

lemma *not-InvMsgInGlob-imp-not-SafeGlobMono*:
assumes ¬ *InvMsgInGlob c0*

and *InvGlobNonposEqVacant c0*
shows $\exists$ *c1. next-recvupd c0 c1* $\wedge \neg$ *SafeGlobMono c0 c1*
proof −
note *npeq0 = assms(2)*[*unfolded InvGlobNonposEqVacant-def*, *rule-format*]
from *assms(1)* **obtain** *p q t c1* **where** *t*:
next-recvupd′ c0 c1 p q
t $\in\#_z$ *hd* (*c-msg c0 p q*)
GlobVacantUpto c0 q t
by (*auto simp*: *InvMsgInGlob-def npeq0 nonpos-upto-def not-less dest*!: *ex-next-recvupd*)
have *nvu*: $\neg$ *GlobVacantUpto c1 q t*
unfolding *vacant-upto-def next-recvupdD(4)*[*OF t(1)*]
using *t(2) t(3) vacant-upto-def* **by** *auto*
from *t(3) nvu* **have** $\neg$ *SafeGlobMono c0 c1*
by (*auto simp*: *SafeGlobMono-def*)
with *t(1)* **show** *?thesis*
by *blast*
qed

lemma *GII-eq-GIA*: *GlobalIncomingInfo c 1 p q* = (*if c-msg c p q* = [] *then GlobalIncomingInfoAt c q else GlobalIncomingInfoAt c q* − *hd* (*c-msg c p q*))
unfolding *GlobalIncomingInfo-def*
apply (*cases c-msg c p q* = [])
apply (*simp add*: *IncomingInfo-def*)
apply (*rule sum.cong*[*OF refl*], *simp*)
apply *simp*
apply (*subst diff-conv-add-uminus*)
apply (*rule Sum-eq-pick-changed-elem*)
apply (*auto simp*: *IncomingInfo-def drop-Suc sum-list-hd-tl algebra-simps*)
done

lemma *recvupd-preserves-GlobVacantUpto*:
assumes *GlobVacantUpto c0 q t*
and *next-recvupd′ c0 c1 p q*
and *InvInfoJustifiedWithGII c0*
and *InvGlobNonposEqVacant c1*
and *InvGlobVacantImpRecordsVacant c0*
and *InvRecordCount c0*
shows *GlobVacantUpto c1 q t*
proof (*rule ccontr*)
note *npeq1 = assms(4)*[*unfolded InvGlobNonposEqVacant-def*, *rule-format*]
note *gvu0 = assms(1)*[*unfolded vacant-upto-def*, *rule-format*]
note *nvu0 = assms(5)*[*unfolded InvGlobVacantImpRecordsVacant-def*, *rule-format*, *OF assms(1)*]
note *recordcount = assms(6)*[*unfolded InvRecordCount-def zmultiset-eq-iff zcount-union*, *rule-format*]
note *change = next-recvupdD*[*OF assms(2)*]
let *?κ = hd* (*c-msg c0 p q*)
from *change(1)* **have** *iaκ*: *?κ = InfoAt c0 0 p q*
unfolding *InfoAt-def* **by** (*simp add*: *hd-conv-nth*)

assume *gvu1*: ¬ *GlobVacantUpto c1 q t*
obtain *t′* **where** *t′*:
t′ ≤ *t*
0 < *zcount* (*c-glob c1 q*) *t′*
zcount (*c-glob c0 q*) *t′* = *0*
∀ *s*<*t′*. *zcount* (*c-glob c1 q*) *s* ≤ *0*
apply *atomize-elim*
using *gvu1*
unfolding *npeq1 nonpos-upto-def*
apply (*simp add*: *not-le*)
apply (*elim exE conjE*)
apply (*drule order-zmset-exists-foundation′*)
apply (*elim exE conjE*)
subgoal for *s s′*
apply (*rule exI*[*of* - *s′*])
using *gvu0* **apply** *auto*
done
done
from *t′*(*2*,*3*) **have** *countκ*: *0* < *zcount ?κ t′*
by (*auto simp*: *change*(*4*))
note *assms*(*3*)[*unfolded InvInfoJustifiedWithGII-def justified-with-alt*, *rule-format*, *OF countκ*[*unfolded iaκ*]]
then consider
∃ *s*<*t′*. *zcount* (*InfoAt c0 0 p q*) *s* < *0* ∧ (∀ *s′*<*s*. *zcount* (*InfoAt c0 0 p q*) *s′* ≤ *0*) |
∃ *s*<*t′*. *zcount* (*GlobalIncomingInfo c0 1 p q*) *s* < *0* ∧ (∀ *s′*<*s*. *zcount* (*InfoAt c0 0 p q*) *s′* ≤ *0*)
∄ *s*. *s* < *t′* ∧ *zcount* (*InfoAt c0 0 p q*) *s* < *0* ∧ (∀ *s′*<*s*. *zcount* (*InfoAt c0 0 p q*) *s′* ≤ *0*) |
∃ *s*<*t′*. *0* < *zcount* (*records c0*) *s* |
zcount (*InfoAt c0 0 p q* + *GlobalIncomingInfo c0 1 p q*) *t′* < *zcount* (*records c0*) *t′*
by *atomize-elim auto*
then show *False*
proof *cases*
case *1*
then obtain *s* **where** *s*: *s* < *t′ zcount* (*InfoAt c0 0 p q*) *s* < *0* ∀ *s′*<*s*. *zcount* (*InfoAt c0 0 p q*) *s′* ≤ *0*
by *blast*
then have *globs*: *zcount* (*c-glob c1 q*) *s* < *0*
using *gvu0 iaκ t′*(*1*) **by** (*auto simp*: *change*(*4*))
have *zcount* (*c-glob c1 q*) *s* ≤ *0*
using *globs* **by** *linarith*
then have ∀ *s′*≤*s*. *zcount* (*c-glob c1 q*) *s′* ≤ *0*
using *s*(*1*) *t′*(*4*) **by** *auto*
with *globs* **show** *False*
by (*auto simp*: *npeq1*[*unfolded nonpos-upto-def*, *symmetric*] *vacant-upto-def*)
next
case *2*

```
    then obtain s where s: s < t' zcount (GlobalIncomingInfo c0 1 p q) s < 0
∀ s'<s. zcount (InfoAt c0 0 p q) s' ≤ 0
      by blast
    from 2(2) s(1,3) have zcount (InfoAt c0 0 p q) s ≥ 0
      by force
    have rc: zcount (records c0) s = 0
    using order.strict-implies-order order.strict-trans2 nvu0 s(1) t'(1) vacant-upto-def
by blast
    show False
      using assms(6) change(1,4) s(1,2) rc t'(4)
      unfolding GII-eq-GIA InvRecordCount-def fun-upd-same
        by clarsimp (metis add.commute add-mono-thms-linordered-field(1) not-le
recordcount)
  next
    case 3
    with nvu0 t'(1) show False
      unfolding vacant-upto-def by auto
  next
    case 4
    then have 0 < zcount (c-glob c0 q) t'
      using change(1) iaκ t'(3)
      unfolding GII-eq-GIA
      apply clarsimp
      apply (metis add.right-neutral preorder-class.less-irrefl recordcount)
      done
    then show False
      by (simp add: t'(3))
  qed
qed

lemma recvupd-imp-SafeGlobMono:
  assumes next-recvupd' c0 c1 p q
    and   InvInfoJustifiedWithGII c0
    and   InvGlobNonposEqVacant c1
    and   InvGlobVacantImpRecordsVacant c0
    and   InvRecordCount c0
  shows   SafeGlobMono c0 c1
  unfolding SafeGlobMono-def
  apply clarify
  subgoal for q' t
    apply (cases q'=q)
    subgoal
      apply (rule recvupd-preserves-GlobVacantUpto)
      using assms apply simp-all
      done
    subgoal
      unfolding next-recvupdD[OF assms(1)]
      by simp
    done
```

done

lemma *next′-imp-SafeGlobMono*:
assumes *next′ c0 c1*
and *InvInfoJustifiedWithGII c0*
and *InvGlobNonposEqVacant c1*
and *InvGlobVacantImpRecordsVacant c0*
and *InvRecordCount c0*
shows *SafeGlobMono c0 c1*
using *assms* **unfolding** *next′-def*
apply (*elim disjE exE*)
subgoal
by (*auto simp*: *SafeGlobMono-def dest*: *next-performopD*(*7*))
subgoal
by (*auto simp*: *SafeGlobMono-def dest*: *next-sendupdD*(*5*))
subgoal
by (*auto intro*: *recvupd-imp-SafeGlobMono*)
subgoal
by (*auto simp*: *SafeGlobMono-def dest*: *next-recvcapD*(*4*))
subgoal
by (*simp add*: *SafeGlobMono-def*)
done

lemma *invs-imp-InvMsgInGlob*:
fixes *c0* :: (*′p*::*finite*, *′a*) *configuration*
assumes *InvInfoJustifiedWithGII c0*
and *InvGlobNonposEqVacant c0*
and *InvGlobVacantImpRecordsVacant c0*
and *InvRecordCount c0*
and *InvJustifiedII c0*
and *InvCapsNonneg c0*
and *InvRecordsNonneg c0*
shows *InvMsgInGlob c0*
using *assms*
apply −
apply (*rule ccontr*)
apply (*drule not-InvMsgInGlob-imp-not-SafeGlobMono*[*rotated*, *OF assms*(*2*)])
apply (*elim exE conjE*)
apply (*frule recvupd-imp-SafeGlobMono*)
apply *simp*
subgoal
apply (*rule invs-imp-InvGlobNonposEqVacant*)
apply (*drule* (*2*) *recvupd-preserves-InvRecordCount*)
apply (*drule* (*1*) *recvupd-preserves-InvJustifiedII*)
apply (*rule InvJustifiedII-implies-InvJustifiedGII*)
apply *simp*
apply (*drule* (*2*) *recvupd-preserves-InvCapsNonneg*)
apply (*simp add*: *InvRecordsNonneg-def next-recvupd-complexD*)
done

```
    apply simp
   apply simp
  apply simp
  done

lemma alw-SafeGlobMono: spec s ⟹ alw (relates SafeGlobMono) s
proof −
  assume spec: spec s
  { assume alw next s
    moreover assume alw (holds InvGlobNonposEqVacant) s
    moreover assume alw (holds InvGlobVacantImpRecordsVacant) s
    moreover assume alw (holds InvInfoJustifiedWithGII) s
    moreover assume alw (holds InvRecordCount) s
    ultimately have alw (relates SafeGlobMono) s
      apply (coinduction arbitrary: s)
      apply (clarsimp simp: relates-def intro!: next′-imp-SafeGlobMono)
      apply (rule conjI)
       apply (rule next′-imp-SafeGlobMono)
           apply auto [2]
         apply (subst (asm) alw-holds2)
         apply clarify
         apply (drule alwD)+
         apply simp
        apply auto
      done
  }
  with spec show ?thesis
    apply −
    apply (frule alw-InvGlobNonposEqVacant)
    apply (frule alw-InvGlobVacantImpRecordsVacant)
    apply (frule alw-InvInfoJustifiedWithGII)
    apply (frule alw-InvRecordCount)
    unfolding spec-def
    apply auto
    done
qed

lemma alw-InvMsgInGlob: spec s ⟹ alw (holds InvMsgInGlob) s
  apply (frule alw-InvInfoJustifiedWithGII)
  apply (frule alw-InvGlobNonposEqVacant)
  apply (frule alw-InvGlobVacantImpRecordsVacant)
  apply (frule alw-InvRecordCount)
  apply (frule alw-InvJustifiedII)
  apply (frule alw-InvCapsNonneg)
  apply (drule alw-InvRecordsNonneg)
  apply (simp add: alw-iff-sdrop invs-imp-InvMsgInGlob)
  done

lemma SafeGlobMono-preserves-vacant:
```

assumes $\forall\, t'{\leq}t.\ zcount\ (c\text{-}glob\ c0\ q)\ t' = 0$
and $(\lambda c0\ c1.\ SafeGlobMono\ c0\ c1)^{**}\ c0\ c1$
shows $\forall\, t'{\leq}t.\ zcount\ (c\text{-}glob\ c1\ q)\ t' = 0$
using *assms*(*2*,*1*)
by (*induct rule*: *rtranclp-induct*)(*auto simp*: *SafeGlobMono-def vacant-upto-def*)

lemma *rtranclp-all-imp-rel*: $r^{**}\ x\ y \Longrightarrow \forall\, a\ b.\ r\ a\ b \longrightarrow r'\ a\ b \Longrightarrow r'^{**}\ x\ y$
by (*metis mono-rtranclp*)

lemma *rtranclp-rel-and-invar*: $r^{**}\ x\ y \Longrightarrow Q\ x \Longrightarrow \forall\, a\ b.\ Q\ a \wedge r\ a\ b \longrightarrow P\ a\ b \wedge Q\ b \Longrightarrow (\lambda x\ y.\ P\ x\ y \wedge Q\ y)^{**}\ x\ y$
apply (*induct rule*: *rtranclp-induct*)
apply *auto* []
apply (*metis* (*no-types*, *lifting*) *rtranclp.simps*)
done

lemma *rtranclp-invar-conclude-last*: $(\lambda x\ y.\ P\ x\ y \wedge Q\ y)^{**}\ x\ y \Longrightarrow Q\ x \Longrightarrow Q\ y$
using *rtranclp.cases* **by** *fastforce*

lemma *InvCapsNonneg-imp-InvRecordsNonneg*: *InvCapsNonneg c* $\Longrightarrow$ *InvRecordsNonneg c*
unfolding *InvCapsNonneg-def InvRecordsNonneg-def records-def*
by (*auto simp*: *zcount-sum intro*!: *sum-nonneg add-nonneg-nonneg*)

lemma *invs-imp-msg-in-glob*:
fixes $c :: ('p{::}finite, 'a)\ configuration$
assumes $M \in set\ (c\text{-}msg\ c\ p\ q)$
and $t \in\#_z\ M$
and *InvGlobNonposEqVacant c*
and *InvJustifiedII c*
and *InvInfoJustifiedWithII c*
and *InvGlobVacantImpRecordsVacant c*
and *InvRecordCount c*
and *InvCapsNonneg c*
and *InvMsgInGlob c*
shows $\exists\, t'{\leq}t.\ 0 < zcount\ (c\text{-}glob\ c\ q)\ t'$
proof (*rule ccontr*)
assume $\neg\ (\exists\, t'{\leq}t.\ 0 < zcount\ (c\text{-}glob\ c\ q)\ t')$
then have *vacant*: $\forall\, t'{\leq}t.\ zcount\ (c\text{-}glob\ c\ q)\ t' = 0$
using *assms*(*3*)
by (*auto simp*: *InvGlobNonposEqVacant-def vacant-upto-def nonpos-upto-def*)
let *?c1* = *the* (*while-option* $(\lambda c.\ hd\ (c\text{-}msg\ c\ p\ q) \neq M)\ (\lambda c.\ SOME\ c'.\ next\text{-}recvupd'\ c\ c'\ p\ q)\ c$)
have *r*[*simp*]: $M \in set\ (c\text{-}msg\ c\ p\ q) \Longrightarrow c\text{-}msg\ (SOME\ c'.\ next\text{-}recvupd'\ c\ c'\ p\ q)\ p\ q = tl\ (c\text{-}msg\ c\ p\ q)$ **for** *c*
by (*rule someI2-ex*[*OF ex-next-recvupd*]) (*auto simp*: *next-recvupd'-def*)
obtain *c1* **where** *while-some*:
while-option $(\lambda c.\ hd\ (c\text{-}msg\ c\ p\ q) \neq M)\ (\lambda c.\ SOME\ c'.\ next\text{-}recvupd'\ c\ c'\ p\ q)\ c = Some\ c1$

apply *atomize-elim*
apply (*rule measure-while-option-Some*[**where** $P{=}\lambda c.\ M \in set\ (c\text{-}msg\ c\ p\ q)$
and $f{=}\lambda c.\ Min\ \{i.\ i < length\ (c\text{-}msg\ c\ p\ q) \wedge M = c\text{-}msg\ c\ p\ q\ !\ i\}$])
apply *clarsimp*
apply *safe*
apply (*metis list.exhaust-sel list.sel*(*2*) *set-ConsD*)
apply (*subst Min-gr-iff*)
apply (*auto simp*: *in-set-conv-nth nth-tl*) [*2*]
apply *clarsimp*
apply (*subst Min-less-iff*)
apply (*auto simp*: *in-set-conv-nth nth-tl*) []
apply (*clarsimp simp*: *in-set-conv-nth nth-tl*)
apply (*metis* (*no-types*, *opaque-lifting*) *Suc-less-eq Suc-pred hd-conv-nth list.size*(*3*) *not-gr-zero not-less-zero*)
apply (*clarsimp simp*: *in-set-conv-nth nth-tl*)
subgoal for *s x*
by (*rule exI*[*of* - *x*−*1*])
(*metis One-nat-def Suc-le-eq Suc-pred*′ *diff-less diff-less-mono hd-conv-nth length-tl list.size*(*3*) *not-gr-zero nth-tl zero-less-Suc*)
apply (*meson assms*(*1*) *in-set-conv-nth*)
done
have *c1*: $(\lambda c0\ c1.\ next\text{-}recvupd'\ c0\ c1\ p\ q)^{**}\ c\ c1 \Longrightarrow c\text{-}msg\ c1\ p\ q \neq [] \Longrightarrow hd\ (c\text{-}msg\ c1\ p\ q) = M$
subgoal
apply (*rule conjunct2*)
apply (*rule while-option-rule*[*OF* - *while-some*, **where** $P{=}\lambda d.\ M \in set\ (c\text{-}msg\ d\ p\ q) \wedge (\lambda c0\ c1.\ next\text{-}recvupd'\ c0\ c1\ p\ q)^{**}\ c\ d$])
apply (*rule conjI*)
apply (*metis list.sel*(*1*) *list.sel*(*3*) *list.set-cases r*)
apply (*auto simp*: *assms*(*1*) *elim*!: *rtrancl-into-rtrancl*[*to-pred*] *intro*: *someI2-ex*[*OF ex-next-recvupd*])
done
subgoal
using *while-option-rule*[*OF* - *while-some*, **where** $P{=}\lambda c.\ M \in set\ (c\text{-}msg\ c\ p\ q)$]
by (*metis assms*(*1*) *empty-iff hd-Cons-tl list.set*(*1*) *r set-ConsD*)
subgoal
using *while-option-stop*[*OF while-some*] **by** *simp*
done
have *invs-trancl*:
$(\lambda c0\ c1.\ SafeGlobMono\ c0\ c1 \wedge InvJustifiedII\ c1 \wedge InvGlobNonposEqVacant\ c1 \wedge InvInfoJustifiedWithII\ c1 \wedge InvGlobVacantImpRecordsVacant\ c1 \wedge InvRecordCount\ c1 \wedge InvCapsNonneg\ c1)^{**}\ c\ c1$
apply (*rule rtranclp-rel-and-invar*)
using *c1*(*1*) **apply** *simp*
using *assms*(*3*−*8*) **apply** *simp*
apply *clarsimp*
subgoal for *a b*
apply (*frule InvJustifiedII-implies-InvJustifiedGII*)

```
      apply simp
     apply (frule recvupd-preserves-InvJustifiedII)
      apply simp
     apply (frule InvCapsNonneg-imp-InvRecordsNonneg)
     apply (frule next-preserves-InvRecordCount[of - b])
      apply (auto simp: next'-def) []
     apply (frule recvupd-preserves-InvCapsNonneg[of - b])
      apply simp
     apply (frule InvJustifiedII-implies-InvJustifiedGII[of b])
      apply simp
     apply (frule InvCapsNonneg-imp-InvRecordsNonneg[of b])
     apply (frule invs-imp-InvGlobNonposEqVacant[of b])
       apply simp-all [2]
     apply (frule recvupd-preserves-InvInfoJustifiedWithII[of - b])
      apply simp
     apply (frule invs-imp-InvInfoJustifiedWithGII)
       apply simp-all [2]
     apply (frule invs-imp-InvInfoJustifiedWithGII[of b])
       apply simp-all [2]
     apply (intro conjI)
          apply (rule next'-imp-SafeGlobMono)
              apply (clarsimp simp: next'-def)
             apply simp-all [7]
       apply (rule invs-imp-InvGlobVacantImpRecordsVacant)
         apply simp-all
      done
   done
  then have trancl-mono: (λc0 c1. SafeGlobMono c0 c1)** c c1
    by (metis (no-types, lifting) rtranclp-all-imp-rel)
  then have vacant-c1: ∀ t'≤t. zcount (c-glob c1 q) t' = 0
    by (rule SafeGlobMono-preserves-vacant[OF vacant])
  have InvMsgInGlob: InvMsgInGlob c1
    using rtranclp-invar-conclude-last[OF invs-trancl] assms
    apply (intro invs-imp-InvMsgInGlob)
   using invs-imp-InvInfoJustifiedWithGII InvCapsNonneg-imp-InvRecordsNonneg
apply blast+
    done
  have ∃ t'≤t. 0 < zcount (c-glob c1 q) t'
    using InvMsgInGlob[unfolded InvMsgInGlob-def] c1(2,3) assms(1,2)
    by auto
  then show False
    using vacant-c1 by auto
qed

lemma alw-msg-glob: spec s ⟹
  alw (holds (λc. ∀ p q t. (∃ M ∈ set (c-msg c p q). t ∈#z M) ⟶ (∃ t'≤t. 0 <
zcount (c-glob c q) t'))) s
  apply (frule alw-InvGlobNonposEqVacant)
  apply (frule alw-InvJustifiedII)
```

apply (*frule alw-InvInfoJustifiedWithII*)
apply (*frule alw-InvGlobVacantImpRecordsVacant*)
apply (*frule alw-InvRecordCount*)
apply (*frule alw-InvCapsNonneg*)
apply (*drule alw-InvMsgInGlob*)
apply (*coinduction arbitrary*: *s*)
apply *clarsimp*
apply (*rule conjI*)
apply *clarify*
apply (*rule invs-imp-msg-in-glob*)
apply *auto* [*2*]
using *holds.elims*(*2*) **apply** *blast*+
done

end

5 Antichains

definition *incomparable* **where**
incomparable A = (∀ *x* ∈ *A*. ∀ *y* ∈ *A*. *x* ≠ *y* ⟶ ¬ *x* < *y* ∧ ¬ *y* < *x*)

lemma *incomparable-empty*[*simp*, *intro*]: *incomparable* {}
unfolding *incomparable-def* **by** *auto*

typedef (**overloaded**) 'a :: *order antichain* =
{*A* :: 'a *set*. *finite A* ∧ *incomparable A*}
morphisms *set-antichain antichain*
by *auto*

setup-lifting *type-definition-antichain*

lift-definition *member-antichain* :: 'a :: *order* ⇒ 'a *antichain* ⇒ *bool* ((-/ ∈$_A$ -) [*51*, *51*] *50*) **is** *Set.member* .

abbreviation *not-member-antichain* :: 'a :: *order* ⇒ 'a *antichain* ⇒ *bool* ((-/ ∉$_A$ -) [*51*, *51*] *50*) **where**
x ∉$_A$ *A* ≡ ¬ *x* ∈$_A$ *A*

lift-definition *empty-antichain* :: 'a :: *order antichain* ({}$_A$) **is** {} **by** *simp*

lemma *mem-antichain-nonempty*[*simp*]: *s* ∈$_A$ *A* ⟹ *A* ≠ {}$_A$
by *transfer auto*

definition *minimal-antichain A* = {*x* ∈ *A*. ¬(∃ *y* ∈ *A*. *y* < *x*)}

lemma *in-minimal-antichain*: *x* ∈ *minimal-antichain A* ⟷ *x* ∈ *A* ∧ ¬(∃ *y* ∈ *A*. *y* < *x*)

unfolding *minimal-antichain-def* **by** *auto*

lemma *in-antichain-minimal-antichain*[*simp*]: *finite M* $\Longrightarrow$ *x* $\in_A$ *antichain* (*minimal-antichain M*) $\longleftrightarrow$ *x* $\in$ *minimal-antichain M*
apply (*clarsimp simp*: *minimal-antichain-def member-antichain.rep-eq*)
apply (*intro conjI iffI*)
apply (*subst* (*asm*) *antichain-inverse*)
apply (*simp add*: *incomparable-def*)
apply *simp*
apply (*subst* (*asm*) *antichain-inverse*)
apply (*simp add*: *incomparable-def*)
apply *simp*
apply (*subst antichain-inverse*)
apply (*simp add*: *incomparable-def*)
apply *simp*
done

lemma *incomparable-minimal-antichain*[*simp*]: *incomparable* (*minimal-antichain A*)
unfolding *incomparable-def minimal-antichain-def*
by *auto*

lemma *finite-minimal-antichain*[*simp*]: *finite A* $\Longrightarrow$ *finite* (*minimal-antichain A*)
unfolding *minimal-antichain-def* **by** *auto*

lemma *finite-set-antichain*[*simp*, *intro*]: *finite* (*set-antichain A*)
by *transfer auto*

lemma *minimal-antichain-subset*: *minimal-antichain A* $\subseteq$ *A*
unfolding *minimal-antichain-def* **by** *auto*

lift-definition *frontier* :: *'t* :: *order zmultiset* $\Rightarrow$ *'t antichain* **is**
λM. *minimal-antichain* {*t*. *zcount M t* > *0*}
by (*auto simp*: *finite-subset*[*OF minimal-antichain-subset finite-zcount-pos*])

lemma *member-frontier-pos-zmset*: *t* $\in_A$ *frontier M* $\Longrightarrow$ *0* < *zcount M t*
by (*simp add*: *frontier-def in-minimal-antichain*)

lemma *frontier-comparable-False*[*simp*]: *x* $\in_A$ *frontier M* $\Longrightarrow$ *y* $\in_A$ *frontier M* $\Longrightarrow$ *x* < *y* $\Longrightarrow$ *False*
by *transfer* (*auto simp*: *minimal-antichain-def*)

lemma *minimal-antichain-idempotent*[*simp*]: *minimal-antichain* (*minimal-antichain A*) = *minimal-antichain A*
by (*auto simp*: *minimal-antichain-def*)

instantiation *antichain* :: (*order*) *minus* **begin**
lift-definition *minus-antichain* :: *'a antichain* $\Rightarrow$ *'a antichain* $\Rightarrow$ *'a antichain* **is**
(−)

by (*auto simp*: *incomparable-def*)
instance ..
end

instantiation *antichain* :: (*order*) *plus* **begin**
lift-definition *plus-antichain* :: *'a antichain* ⇒ *'a antichain* ⇒ *'a antichain* **is** λ*M N*. *minimal-antichain* (*M* ∪ *N*)
by (*auto simp*: *incomparable-def minimal-antichain-def*)
instance ..
end

lemma *antichain-add-commute*: (*M* :: *'a* :: *order antichain*) + *N* = *N* + *M*
by *transfer* (*auto simp*: *incomparable-def sup-commute*)

lift-definition *filter-antichain* :: (*'a* :: *order* ⇒ *bool*) ⇒ *'a antichain* ⇒ *'a antichain* **is** *Set.filter*
by (*auto simp*: *incomparable-def*)

syntax (*ASCII*)
-ACCollect :: *pttrn* ⇒ *'a* :: *order antichain* ⇒ *bool* ⇒ *'a antichain* ((*1*{- :*A* -./ -}))
syntax
-ACCollect :: *pttrn* ⇒ *'a* :: *order antichain* ⇒ *bool* ⇒ *'a antichain* ((*1*{- ∈*A* -./ -}))
translations
{*x* ∈*A* *M*. *P*} == *CONST filter-antichain* (λ*x*. *P*) *M*

declare *empty-antichain.rep-eq*[*simp*]

lemma *minimal-antichain-empty*[*simp*]: *minimal-antichain* {} = {}
by (*simp add*: *minimal-antichain-def*)

lemma *minimal-antichain-singleton*[*simp*]: *minimal-antichain* {*x*::- ::*order*} = {*x*}
by (*auto simp*: *minimal-antichain-def*)

lemma *minimal-antichain-nonempty*:
finite A ⟹ (*t*::-::*order*) ∈ *A* ⟹ *minimal-antichain A* ≠ {}
by (*auto simp*: *minimal-antichain-def dest*: *order-finite-set-exists-foundation*[*of* - *t*])

lemma *minimal-antichain-member*:
finite A ⟹ (*t*::-::*order*) ∈ *A* ⟹ ∃ *t'*. *t'* ∈ *minimal-antichain A* ∧ *t'* ≤ *t*
by (*auto simp*: *minimal-antichain-def dest*: *order-finite-set-exists-foundation*[*of* - *t*])

lemma *minimal-antichain-union*: *minimal-antichain* ((*A*::(- :: *order*) *set*) ∪ *B*) ⊆ *minimal-antichain* (*minimal-antichain A* ∪ *minimal-antichain B*)

by (*auto simp*: *minimal-antichain-def*)

lemma *ac-Diff-iff*: $c \in_A A - B \longleftrightarrow c \in_A A \wedge c \notin_A B$
by *transfer simp*

lemma *ac-DiffD2*: $c \in_A A - B \Longrightarrow c \in_A B \Longrightarrow P$
by *transfer simp*

lemma *ac-notin-Diff*: $\neg\ x \in_A A - B \Longrightarrow \neg\ x \in_A A \vee x \in_A B$
by *transfer simp*

lemma *ac-eq-iff*: $A = B \longleftrightarrow (\forall x.\ x \in_A A \longleftrightarrow x \in_A B)$
by *transfer auto*

lemma *antichain-obtain-foundation*:
assumes $t \in_A M$
obtains *s* **where** $s \in_A M \wedge s \leq t \wedge (\forall u.\ u \in_A M \longrightarrow \neg\ u < s)$
using *assms* **unfolding** *member-antichain.rep-eq*
by − (*rule order-finite-set-obtain-foundation*[*of set-antichain M t*]; *auto*)

lemma *set-antichain1*[*simp*]: $x \in$ *set-antichain X* $\Longrightarrow x \in_A X$
by *transfer simp*

lemma *set-antichain2*[*simp*]: $x \in_A X \Longrightarrow x \in$ *set-antichain X*
by *transfer simp*

6 Multigraphs with Partially Ordered Weights

abbreviation (*input*) *FROM* **where**
FROM $\equiv \lambda(s,\ l,\ t).\ s$

abbreviation (*input*) *LBL* **where**
LBL $\equiv \lambda(s,\ l,\ t).\ l$

abbreviation (*input*) *TO* **where**
TO $\equiv \lambda(s,\ l,\ t).\ t$

notation *subseq* (**infix** $\preceq$ *50*)

locale *graph* =
fixes *weights* :: *'vtx* :: *finite* ⇒ *'vtx* ⇒ *'lbl* :: {*order*, *monoid-add*} *antichain*
assumes *zero-le*[*simp*]: *0* ≤ (*s*::*'lbl*)
and *plus-mono*: (*s1*::*'lbl*) ≤ *s2* ⟹ *s3* ≤ *s4* ⟹ *s1* + *s3* ≤ *s2* + *s4*
and *summary-self*: *weights loc loc* = {}$_A$
begin

lemma *le-plus*: (*s*::*'lbl*) ≤ *s* + *s'* (*s'*::*'lbl*) ≤ *s* + *s'*

by (*intro plus-mono*[*of s s 0 s'*, *simplified*] *plus-mono*[*of 0 s s' s'*, *simplified*])+

6.1 Paths

inductive *path* :: *'vtx* ⇒ *'vtx* ⇒ (*'vtx* × *'lbl* × *'vtx*) *list* ⇒ *bool* **where**
  path0: *l1* = *l2* ⟹ *path l1 l2* []
| *path*: *path l1 l2 xs* ⟹ *lbl* ∈$_A$ *weights l2 l3* ⟹ *path l1 l3* (*xs* @ [(*l2*, *lbl*, *l3*)])

inductive-cases *path0E*: *path l1 l2* []
inductive-cases *path-AppendE*: *path l1 l3* (*xs* @ [(*l2*,*s*,*l2'*)])

lemma *path-trans*: *path l1 l2 xs* ⟹ *path l2 l3 ys* ⟹ *path l1 l3* (*xs* @ *ys*)
  by (*rotate-tac*, *induct l2 l3 ys rule*: *path.induct*)
    (*auto intro*: *path.path simp flip*: *append-assoc*)

lemma *path-take-from*: *path l1 l2 xs* ⟹ *m* < *length xs* ⟹ *FROM* (*xs* ! *m*) = *l2'*
⟹ *path l1 l2'* (*take m xs*)
proof (*induct l1 l2 xs rule*: *path.induct*)
  case (*path l1 l2 xs lbl l3*)
  then show *?case*
    apply (*unfold take-append*)
    apply *simp*
    apply (*cases l2=l2'*)
     apply (*metis linorder-not-less nth-append take-all*)
    apply (*metis case-prod-conv less-Suc-eq nth-append nth-append-length*)
    done
qed *simp*

lemma *path-take-to*: *path l1 l2 xs* ⟹ *m* < *length xs* ⟹ *TO* (*xs* ! *m*) = *l2'* ⟹
path l1 l2' (*take* (*m*+*1*) *xs*)
proof (*induct l1 l2 xs rule*: *path.induct*)
  case (*path l1 l2 xs lbl l3*)
  then show *?case*
    apply (*cases m* < *length xs*)
     apply (*simp add*: *nth-append*)
    apply *clarsimp*
    apply (*metis case-prod-conv less-antisym nth-append-length path.path*)
    done
qed *simp*

lemma *path-determines-loc*: *path l1 l2 xs* ⟹ *path l1 l3 xs* ⟹ *l2* = *l3*
  by (*induct l1 l2 xs rule*: *path.induct*) (*auto elim*: *path.cases*)

lemma *path-first-loc*: *path loc loc' xs* ⟹ *xs* ≠ [] ⟹ *FROM* (*xs* ! *0*) = *loc*
proof (*induct rule*: *path.induct*)
  case (*path l1 l2 xs lbl l3*)
  then show *?case*
    by (*auto elim*: *path0E simp*: *nth-append*)
qed *simp*

lemma *path-to-eq-from*: *path loc1 loc2 xs* $\Longrightarrow$ *i + 1 < length xs* $\Longrightarrow$ *FROM* (*xs* ! (*i+1*)) = *TO* (*xs* ! *i*)
proof (*induct rule*: *path.induct*)
  case (*path l1 l2 xs lbl l3*)
  then show *?case*
    apply (*cases i + 1 < length xs*)
     apply (*simp add*: *nth-append*)
    apply (*simp add*: *nth-append*)
   apply (*metis add.commute drop-eq-Nil hd-drop-conv-nth id-take-nth-drop linorder-not-less path-determines-loc path-take-to plus-1-eq-Suc take-hd-drop*)
    done
qed *simp*

lemma *path-singleton*[*intro*, *simp*]: *s* $\in_A$ *weights l1 l2* $\Longrightarrow$ *path l1 l2* [(*l1*,*s*,*l2*)]
  by (*subst path.simps*) (*auto simp*: *path.intros*)

lemma *path-appendE*: *path l1 l3* (*xs* @ *ys*) $\Longrightarrow$ $\exists$ *l2*. *path l2 l3 ys* $\wedge$ *path l1 l2 xs*
proof (*induct l1 l3 xs@ys arbitrary*: *xs ys rule*: *path.induct*)
  case (*path0 l1 l2*)
  then show *?case* **by** (*auto intro*: *path.intros*)
next
  case (*path l1 l2 xs lbl l3 xs' ys'*)
  from *path*(*1*,*3*−) **show** *?case*
    apply −
    apply (*subst* (*asm*) *append-eq-append-conv2*[*of xs* [(*l2*,*lbl*,*l3*)] *xs' ys'*])
    apply (*elim exE conjE disjE*)
    subgoal for *us*
      using *path*(*2*)[*of xs' us*]
      by (*auto intro*: *path.intros*)
    subgoal for *us*
      by (*cases us*=[]) (*auto intro*: *path.intros simp*: *Cons-eq-append-conv*)
    done
qed

lemma *path-replace-prefix*:
  path l1 l3 (*xs* @ *zs*) $\Longrightarrow$ *path l1 l2 ys* $\Longrightarrow$ *path l1 l2 xs* $\Longrightarrow$ *path l1 l3* (*ys* @ *zs*)
  by (*drule path-appendE*) (*auto elim*!: *path-trans dest*: *path-determines-loc*)

lemma *drop-subseq*: *n* $\leq$ *length xs* $\Longrightarrow$ *drop n xs* $\preceq$ *xs*
  by (*auto simp*: *suffix-def intro*!: *exI*[*of - take n xs*])

lemma *take-subseq*[*simp*, *intro*]: *take n xs* $\preceq$ *xs*
  by (*induct xs*) *auto*

lemma *map-take-subseq*[*simp*, *intro*]: *map f* (*take n xs*) $\preceq$ *map f xs*
  by (*rule subseq-map*, *induct xs*) *auto*

lemma *path-distinct*:

path l1 l2 xs $\Longrightarrow \exists\, xs'.$ *distinct xs'* $\wedge$ *path l1 l2 xs'* $\wedge$ *map LBL xs'* $\preceq$ *map LBL xs*
proof (*induct rule*: *path.induct*)
case (*path0 l1 l2*)
then show *?case*
by (*intro exI*[*of* - []]) (*auto intro*: *path.intros*)
next
case (*path l1 l2 xs lbl l3*)
then obtain *xs'* **where** *ih*: *path l1 l2 xs' distinct xs' map LBL xs'* $\preceq$ *map LBL xs*
by *blast*
then show *?case*
proof (*cases* (*l2*, *lbl*, *l3*) $\in$ *set xs'*)
case *True*
then obtain *m* **where** *m*: *m* < *length xs' xs'* ! *m* = (*l2*, *lbl*, *l3*)
unfolding *in-set-conv-nth* **by** *blast*
from *m ih* **have** *path l1 l2* (*take m xs'*)
by (*auto intro*: *path-take-from*)
with *m ih path* **show** *?thesis*
apply (*intro exI*[*of* - *take m xs'* @ [(*l2*, *lbl*, *l3*)]])
apply (*rule conjI*)
apply (*metis distinct-take take-Suc-conv-app-nth*)
apply (*rule conjI*)
apply (*rule path.intros*)
apply *simp*
apply *simp*
apply *simp*
apply (*metis ih*(*3*) *subseq-order.trans take-map take-subseq*)
done
next
case *False*
with *ih path*(*3*) **show** *?thesis*
by (*auto intro*!: *exI*[*of* - *xs'* @ [(*l2*, *lbl*, *l3*)]] *intro*: *path.intros*)
qed
qed

lemma *path-edge*: (*l1'*, *lbl*, *l2'*) $\in$ *set xs* $\Longrightarrow$ *path l1 l2 xs* $\Longrightarrow$ *lbl* $\in_A$ *weights l1' l2'*
by (*rotate-tac*, *induct rule*: *path.induct*) *auto*

6.2 Path Weights

abbreviation *sum-weights* :: *'lbl list* $\Rightarrow$ *'lbl* **where**
sum-weights xs $\equiv$ *foldr* (+) *xs 0*
abbreviation *sum-path-weights xs* $\equiv$ *sum-weights* (*map LBL xs*)

definition *path-weightp l1 l2 s* $\equiv$ ($\exists\, xs.$ *path l1 l2 xs* $\wedge$ *s* = *sum-path-weights xs*)

lemma *sum-not-less-zero*[*simp*, *dest*]: (*s*::*'lbl*) < *0* $\Longrightarrow$ *False*
by (*simp add*: *less-le-not-le*)

lemma *sum-le-zero*[*simp*]: (*s*::*'lbl*) ≤ *0* ⟷ *s* = *0*
by (*simp add*: *eq-iff*)

lemma *sum-le-zeroD*[*dest*]: (*x*::*'lbl*) ≤ *0* ⟹ *x* = *0*
by *simp*

lemma *foldr-plus-mono*: (*n*::*'lbl*) ≤ *m* ⟹ *foldr* (+) *xs n* ≤ *foldr* (+) *xs m*
by (*induct xs*) (*auto simp*: *plus-mono*)

lemma *sum-weights-append*:
sum-weights (*ys* @ *xs*) = *sum-weights ys* + *sum-weights xs*
by (*induct ys*) (*auto simp*: *add.assoc*)

lemma *sum-summary-prepend-le*: *sum-path-weights ys* ≤ *sum-path-weights xs* ⟹
sum-path-weights (*zs* @ *ys*) ≤ *sum-path-weights* (*zs* @ *xs*)
by (*induct zs arbitrary*: *xs ys*) (*auto intro*: *plus-mono*)

lemma *sum-summary-append-le*: *sum-path-weights ys* ≤ *sum-path-weights xs* ⟹
sum-path-weights (*ys* @ *zs*) ≤ *sum-path-weights* (*xs* @ *zs*)
proof (*induct zs arbitrary*: *xs ys*)
case (*Cons a zs*)
then show *?case*
by (*metis plus-mono map-append order-refl sum-weights-append*)
qed *simp*

lemma *foldr-plus-zero-le*: *foldr* (+) *xs* (*0*::*'lbl*) ≤ *foldr* (+) *xs a*
by (*induct xs*) (*simp-all add*: *plus-mono*)

lemma *subseq-sum-weights-le*:
assumes *xs* ⪯ *ys*
shows *sum-weights xs* ≤ *sum-weights ys*
using *assms*
proof (*induct rule*: *list-emb.induct*)
case (*list-emb-Nil ys*)
then show *?case* **by** *auto*
next
case (*list-emb-Cons xs ys y*)
then show *?case* **by** (*auto elim*!: *order-trans simp*: *le-plus*)
next
case (*list-emb-Cons2 x y xs ys*)
then show *?case* **by** (*auto elim*!: *order-trans simp*: *plus-mono*)
qed

lemma *subseq-sum-path-weights-le*:
map LBL xs ⪯ *map LBL ys* ⟹ *sum-path-weights xs* ≤ *sum-path-weights ys*
by (*rule subseq-sum-weights-le*)

lemma *sum-path-weights-take-le*[*simp*, *intro*]: *sum-path-weights* (*take i xs*) ≤ *sum-path-weights xs*

by (*auto intro*!: *subseq-sum-path-weights-le*)

lemma *sum-weights-append-singleton*:
sum-weights (*xs* @ [*x*]) = *sum-weights xs* + *x*
by (*induct xs*) (*simp-all add*: *add.assoc*)

lemma *sum-path-weights-append-singleton*:
sum-path-weights (*xs* @ [(*l*,*x*,*l'*)]) = *sum-path-weights xs* + *x*
by (*induct xs*) (*simp-all add*: *add.assoc*)

lemma *path-weightp-ex-path*:
path-weightp l1 l2 s ⟹ ∃ *xs*.
(*let s'* = *sum-path-weights xs in s'* ≤ *s* ∧ *path-weightp l1 l2 s'* ∧ *distinct xs* ∧
(∀ (*l1*,*s*,*l2*) ∈ *set xs*. *s* ∈$_A$ *weights l1 l2*))
unfolding *path-weightp-def*
apply (*erule exE conjE*)+
apply (*drule path-distinct*)
apply (*erule exE conjE*)+
subgoal for *xs xs'*
apply (*rule exI*[*of* - *xs'*])
apply (*auto simp*: *Let-def dest*!: *path-edge intro*: *subseq-sum-path-weights-le*)
done
done

lemma *finite-set-summaries*:
finite ((λ((*l1*,*l2*),*s*). (*l1*,*s*,*l2*)) ‘ (*Sigma UNIV* (λ(*l1*,*l2*). *set-antichain* (*weights l1 l2*))))
by *force*

lemma *finite-summaries*: *finite* {*xs*. *distinct xs* ∧ (∀ (*l1*, *s*, *l2*) ∈ *set xs*. *s* ∈$_A$ *weights l1 l2*)}
apply (*rule finite-subset*[*OF* - *finite-subset-distinct*[*of* ((λ((*l1*,*l2*),*s*). (*l1*,*s*,*l2*)) ‘ (*Sigma UNIV* (λ(*l1*,*l2*). *set-antichain* (*weights l1 l2*))))]])
apply (*force simp*: *finite-set-summaries*)+
done

lemma *finite-minimal-antichain-path-weightp*:
finite (*minimal-antichain* {*x*. *path-weightp l1 l2 x*})
apply (*rule finite-surj*[*OF finite-summaries*, **where** *f* = *sum-path-weights*])
apply (*clarsimp simp*: *minimal-antichain-def image-iff dest*!: *path-weightp-ex-path*)
apply (*fastforce simp*: *Let-def*)
done

lift-definition *path-weight* :: *'vtx* ⇒ *'vtx* ⇒ *'lbl antichain*
is λ*l1 l2*. *minimal-antichain* {*x*. *path-weightp l1 l2 x*}
using *finite-minimal-antichain-path-weightp*
by *auto*

definition *reachable l1 l2* ≡ *path-weight l1 l2* ≠ {}$_A$

lemma *in-path-weight*: *s* ∈$_A$ *path-weight loc1 loc2* ⟷ *s* ∈ *minimal-antichain* {*s*. *path-weightp loc1 loc2 s*}
by *transfer simp*

lemma *path-weight-refl*[*simp*]: *0* ∈$_A$ *path-weight loc loc*
proof −
have ∗: *path loc loc* []
by (*simp add*: *path0*)
then have *0* = *sum-path-weights* [] **by** *auto*
with ∗ **have** *path-weightp loc loc 0*
using *path-weightp-def* **by** *blast*
then show *?thesis*
by (*auto simp*: *in-path-weight in-minimal-antichain*)
qed

lemma *zero-in-minimal-antichain*[*simp*]: (*0*::'*lbl*) ∈ *S* ⟹ *0* ∈ *minimal-antichain S*
by (*auto simp*: *in-minimal-antichain intro*: *sum-not-less-zero*)

definition *path-weightp-distinct l1 l2 s* ≡ (∃ *xs*. *distinct xs* ∧ *path l1 l2 xs* ∧ *s* = *sum-path-weights xs*)

lemma *minimal-antichain-path-weightp-distinct*:
minimal-antichain {*xs*. *path-weightp l1 l2 xs*} = *minimal-antichain* {*xs*. *path-weightp-distinct l1 l2 xs*}
unfolding *path-weightp-def path-weightp-distinct-def minimal-antichain-def*
apply *safe*
apply *clarsimp*
apply (*metis path-distinct order.strict-iff-order subseq-sum-path-weights-le*)
apply (*blast*+) [*2*]
apply *clarsimp*
apply (*metis* (*no-types*, *lifting*) *le-less-trans path-distinct subseq-sum-weights-le*)
done

lemma *finite-path-weightp-distinct*[*simp*, *intro*]: *finite* {*xs*. *path-weightp-distinct l1 l2 xs*}
unfolding *path-weightp-distinct-def*
apply (*rule finite-subset*[**where** *B* = *sum-path-weights* ‘ {*xs*. *distinct xs* ∧ *path l1 l2 xs*}])
apply *clarsimp*
apply (*rule finite-imageI*)
apply (*rule finite-subset*[*OF* - *finite-summaries*])
apply (*clarsimp simp*: *path-edge*)
done

lemma *path-weightp-distinct-nonempty*:
{*xs*. *path-weightp l1 l2 xs*} ≠ {} ⟷ {*xs*. *path-weightp-distinct l1 l2 xs*} ≠ {}

by (*auto dest*: *path-distinct simp*: *path-weightp-def path-weightp-distinct-def*)

lemma *path-weightp-distinct-member*:
$s \in \{s.\ path\text{-}weightp\ l1\ l2\ s\} \Longrightarrow \exists\, u.\ u \in \{s.\ path\text{-}weightp\text{-}distinct\ l1\ l2\ s\} \wedge u \leq s$
apply (*clarsimp simp*: *path-weightp-def path-weightp-distinct-def*)
apply (*drule path-distinct*)
apply (*auto dest*: *subseq-sum-path-weights-le*)
done

lemma *minimal-antichain-path-weightp-member*:
$s \in \{xs.\ path\text{-}weightp\ l1\ l2\ xs\} \Longrightarrow \exists\, u.\ u \in minimal\text{-}antichain\ \{xs.\ path\text{-}weightp\ l1\ l2\ xs\} \wedge u \leq s$
proof −
assume $s \in \{xs.\ path\text{-}weightp\ l1\ l2\ xs\}$
then obtain *u* **where** *u*: $u \in \{s.\ path\text{-}weightp\text{-}distinct\ l1\ l2\ s\} \wedge u \leq s$
using *path-weightp-distinct-member* **by** *blast*
have *finite*: *finite* $\{xs.\ path\text{-}weightp\text{-}distinct\ l1\ l2\ xs\}$ **..**
from *u finite* **obtain** *v* **where** $v \in minimal\text{-}antichain\ \{xs.\ path\text{-}weightp\text{-}distinct\ l1\ l2\ xs\} \wedge v \leq u$
by *atomize-elim* (*auto intro*: *minimal-antichain-member*)
with *u* **show** *?thesis*
by (*auto simp*: *minimal-antichain-path-weightp-distinct*)
qed

lemma *path-path-weight*: $path\ l1\ l2\ xs \Longrightarrow \exists\, s.\ s \in_A path\text{-}weight\ l1\ l2 \wedge s \leq sum\text{-}path\text{-}weights\ xs$
proof −
assume *path l1 l2 xs*
then have *sum-path-weights xs* $\in \{x.\ path\text{-}weightp\ l1\ l2\ x\}$
by (*auto simp*: *path-weightp-def*)
then obtain *u* **where** $u \in minimal\text{-}antichain\ \{x.\ path\text{-}weightp\ l1\ l2\ x\} \wedge u \leq sum\text{-}path\text{-}weights\ xs$
apply *atomize-elim*
apply (*drule minimal-antichain-path-weightp-member*)
apply *auto*
done
then show *?thesis*
by *transfer auto*
qed

lemma *path-weight-conv-path*:
$s \in_A path\text{-}weight\ l1\ l2 \Longrightarrow \exists\, xs.\ path\ l1\ l2\ xs \wedge s = sum\text{-}path\text{-}weights\ xs \wedge (\forall\, ys.\ path\ l1\ l2\ ys \longrightarrow \neg\ sum\text{-}path\text{-}weights\ ys < sum\text{-}path\text{-}weights\ xs)$
by *transfer* (*auto simp*: *in-minimal-antichain path-weightp-def*)

abbreviation $optimal\text{-}path\ loc1\ loc2\ xs \equiv path\ loc1\ loc2\ xs \wedge$
$(\forall\, ys.\ path\ loc1\ loc2\ ys \longrightarrow \neg\ sum\text{-}path\text{-}weights\ ys < sum\text{-}path\text{-}weights\ xs)$

lemma *path-weight-path*: $s \in_A$ *path-weight loc1 loc2* $\Longrightarrow$
(⋀*xs*. *optimal-path loc1 loc2 xs* $\Longrightarrow$ *distinct xs* $\Longrightarrow$ *sum-path-weights xs* = *s* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
apply *atomize-elim*
apply *transfer*
apply (*clarsimp simp*: *in-minimal-antichain path-weightp-def*)
apply (*drule path-distinct*)
apply (*erule exE*)
subgoal for *loc1 loc2 xs xs'*
apply (*rule exI*[*of* - *xs'*])
apply *safe*
using *order.strict-iff-order subseq-sum-path-weights-le* **apply** *metis*
using *less-le subseq-sum-path-weights-le* **apply** *fastforce*
done
done

lemma *path-weight-elem-trans*:
s $\in_A$ *path-weight l1 l2* $\Longrightarrow$ *s'* $\in_A$ *path-weight l2 l3* $\Longrightarrow$ ∃ *u*. *u* $\in_A$ *path-weight l1 l3* ∧ *u* ≤ *s* + *s'*
proof −
assume *ps1*: *s* $\in_A$ *path-weight l1 l2*
assume *ps2*: *s'* $\in_A$ *path-weight l2 l3*
from *ps1* **obtain** *xs* **where** *path1*: *path l1 l2 xs s* = *sum-path-weights xs*
by (*auto intro*: *path-weight-path*)
from *ps2* **obtain** *ys* **where** *path2*: *path l2 l3 ys s'* = *sum-path-weights ys*
by (*auto intro*: *path-weight-path*)
from *path1*(*1*) *path2*(*1*) **have** *path l1 l3* (*xs* @ *ys*)
by (*rule path-trans*)
with *path1*(*2*) *path2*(*2*) **have** *s* + *s'* ∈ {*s*. *path-weightp l1 l3 s*}
by (*auto simp*: *path-weightp-def sum-weights-append*[*symmetric*])
then show ∃ *u*. *u* $\in_A$ *path-weight l1 l3* ∧ *u* ≤ *s* + *s'*
by *transfer* (*simp add*: *minimal-antichain-path-weightp-member*)
qed

end

7 Local Progress Propagation

7.1 Specification

record (**overloaded**) ('*loc*, '*t*) *configuration* =
c-work :: '*loc* ⇒ '*t zmultiset*
c-pts :: '*loc* ⇒ '*t zmultiset*
c-imp :: '*loc* ⇒ '*t zmultiset*

type-synonym ('*loc*, '*t*) *computation* = ('*loc*, '*t*) *configuration stream*

locale *dataflow-topology* = *flow?*: *graph summary*

for *summary* :: *'loc* ⇒ *'loc* :: *finite* ⇒ *'sum* :: {*order*, *monoid-add*} *antichain* +
fixes *results-in* :: *'t* :: *order* ⇒ *'sum* ⇒ *'t*
assumes *results-in-zero*: *results-in t 0* = *t*
and *results-in-mono-raw*: *t1* ≤ *t2* ⟹ *s1* ≤ *s2* ⟹ *results-in t1 s1* ≤ *results-in t2 s2*
and *followed-by-summary*: *results-in* (*results-in t s1*) *s2* = *results-in t* (*s1* + *s2*)
and *no-zero-cycle*: *path loc loc xs* ⟹ *xs* ≠ [] ⟹ *s* = *sum-path-weights xs* ⟹ *t* < *results-in t s*
begin

lemma *results-in-mono*:
t1 ≤ *t2* ⟹ *results-in t1 s* ≤ *results-in t2 s*
s1 ≤ *s2* ⟹ *results-in t s1* ≤ *results-in t s2*
using *results-in-mono-raw* **by** *auto*

abbreviation *path-summary* ≡ *path-weight*
abbreviation *followed-by* :: *'sum* ⇒ *'sum* ⇒ *'sum* **where**
followed-by ≡ *plus*

definition *safe* :: (*'loc*, *'t*) *configuration* ⇒ *bool* **where**
safe c ≡ ∀ *loc1 loc2 t s. zcount* (*c-pts c loc1*) *t* > *0* ∧ *s* ∈$_A$ *path-summary loc1 loc2*
⟶ (∃ *t'*≤*results-in t s. t'* ∈$_A$ *frontier* (*c-imp c loc2*))

Implications are always non-negative.

definition *inv-implications-nonneg* **where**
inv-implications-nonneg c = (∀ *loc t. zcount* (*c-imp c loc*) *t* ≥ *0*)

abbreviation *unchanged f c0 c1* ≡ *f c1* = *f c0*

abbreviation *zmset-frontier* **where**
zmset-frontier M ≡ *zmset-of* (*mset-set* (*set-antichain* (*frontier M*)))

definition *init-config* **where**
init-config c ≡ ∀ *loc.*
c-imp c loc = {#}$_z$ ∧
c-work c loc = *zmset-frontier* (*c-pts c loc*)

definition *after-summary* :: *'t zmultiset* ⇒ *'sum antichain* ⇒ *'t zmultiset* **where**
after-summary M S ≡ (∑ *s* ∈ *set-antichain S. image-zmset* (λ*t. results-in t s*) *M*)

abbreviation *frontier-changes* :: *'t zmultiset* ⇒ *'t zmultiset* ⇒ *'t zmultiset* **where**
frontier-changes M N ≡ *zmset-frontier M* − *zmset-frontier N*

definition *next-change-multiplicity'* :: (*'loc*, *'t*) *configuration* ⇒ (*'loc*, *'t*) *configuration* ⇒ *'loc* ⇒ *'t* ⇒ *int* ⇒ *bool* **where**
next-change-multiplicity' c0 c1 loc t n ≡

— n is the non-zero change in pointstamps at loc for timestamp t
$n \neq 0 \wedge$
— change can only happen at timestamps not in advance of implication-frontier
$(\exists\, t'.\ t' \in_A \mathit{frontier}\ (\mathit{c\text{-}imp}\ \mathit{c0}\ \mathit{loc}) \wedge t' \leq t) \wedge$
— at loc, t is added to pointstamps n times
$\mathit{c1} = \mathit{c0}(\!|\mathit{c\text{-}pts} := (\mathit{c\text{-}pts}\ \mathit{c0})(\mathit{loc} := \mathit{update\text{-}zmultiset}\ (\mathit{c\text{-}pts}\ \mathit{c0}\ \mathit{loc})\ t\ n),$
— worklist at loc is adjusted by frontier changes
$\mathit{c\text{-}work} := (\mathit{c\text{-}work}\ \mathit{c0})(\mathit{loc} := \mathit{c\text{-}work}\ \mathit{c0}\ \mathit{loc} +$
$\mathit{frontier\text{-}changes}\ (\mathit{update\text{-}zmultiset}\ (\mathit{c\text{-}pts}\ \mathit{c0}\ \mathit{loc})\ t\ n)\ (\mathit{c\text{-}pts}\ \mathit{c0}\ \mathit{loc}))|\!)$

abbreviation *next-change-multiplicity* :: ('*loc*, '*t*) *configuration* ⇒ ('*loc*, '*t*) *configuration* ⇒ *bool* **where**
next-change-multiplicity c0 c1 ≡ ∃ *loc t n. next-change-multiplicity′ c0 c1 loc t n*

lemma *cm-unchanged-worklist*:
assumes *next-change-multiplicity′ c0 c1 loc t n*
and $\mathit{loc}' \neq \mathit{loc}$
shows $\mathit{c\text{-}work}\ \mathit{c1}\ \mathit{loc}' = \mathit{c\text{-}work}\ \mathit{c0}\ \mathit{loc}'$
using *assms* **unfolding** *next-change-multiplicity′-def*
by *auto*

definition *next-propagate′* :: ('*loc*, '*t*) *configuration* ⇒ ('*loc*, '*t*) *configuration* ⇒ '*loc* ⇒ '*t* ⇒ *bool* **where**
next-propagate′ c0 c1 loc t ≡
— t is a least timestamp of all worklist entries
$(t \in\#_z\ \mathit{c\text{-}work}\ \mathit{c0}\ \mathit{loc} \wedge$
$(\forall\, t'\ \mathit{loc}'.\ t' \in\#_z\ \mathit{c\text{-}work}\ \mathit{c0}\ \mathit{loc}' \longrightarrow \neg\ t' < t) \wedge$
$\mathit{c1} = \mathit{c0}(\!|\mathit{c\text{-}imp} := (\mathit{c\text{-}imp}\ \mathit{c0})(\mathit{loc} := \mathit{c\text{-}imp}\ \mathit{c0}\ \mathit{loc} + \{\#t' \in\#_z\ \mathit{c\text{-}work}\ \mathit{c0}\ \mathit{loc}.\ t' = t\#\}),$
$\mathit{c\text{-}work} := (\lambda \mathit{loc}'.$
— worklist entries for t are removed from loc's worklist
$\mathit{if}\ \mathit{loc}' = \mathit{loc}\ \mathit{then}\ \{\#t' \in\#_z\ \mathit{c\text{-}work}\ \mathit{c0}\ \mathit{loc}'.\ t' \neq t\#\}$
— worklists at other locations change by the loc's frontier change after adding summaries
$\mathit{else}\ \mathit{c\text{-}work}\ \mathit{c0}\ \mathit{loc}'$
$+\ \mathit{after\text{-}summary}$
$(\mathit{frontier\text{-}changes}\ (\mathit{c\text{-}imp}\ \mathit{c0}\ \mathit{loc} + \{\#t' \in\#_z\ \mathit{c\text{-}work}\ \mathit{c0}\ \mathit{loc}.\ t' = t\#\})\ (\mathit{c\text{-}imp}\ \mathit{c0}\ \mathit{loc}))$
$(\mathit{summary}\ \mathit{loc}\ \mathit{loc}'))|\!)$

abbreviation *next-propagate* :: ('*loc*, '*t* :: *order*) *configuration* ⇒ ('*loc*, '*t*) *configuration* ⇒ *bool* **where**
next-propagate c0 c1 ≡ ∃ *loc t. next-propagate′ c0 c1 loc t*

definition *next′* **where**
next′ c0 c1 = (*next-propagate c0 c1* ∨ *next-change-multiplicity c0 c1* ∨ *c1* = *c0*)

abbreviation *next* **where**
next s ≡ *next′* (*shd s*) (*shd* (*stl s*))

abbreviation *cm-valid* **where**
cm-valid ≡ *nxt* (λ*s*. *next-change-multiplicity* (*shd s*) (*shd* (*stl s*))) *impl*
(λ*s*. *next-change-multiplicity* (*shd s*) (*shd* (*stl s*))) *or nxt* (*holds* (λ*c*. (∀ *l*. *c-work c l* = {#}$_z$)))

definition *spec* :: ('*loc*, '*t* :: *order*) *computation* ⇒ *bool* **where**
spec ≡ *holds init-config aand alw next*

lemma *next'-inv*[*consumes 1*, *case-names next-change-multiplicity next-propagate next-finish-init*]:
assumes *next' c0 c1 P c0*
and ⋀*loc t n*. *P c0* ⟹ *next-change-multiplicity' c0 c1 loc t n* ⟹ *P c1*
and ⋀*loc t*. *P c0* ⟹ *next-propagate' c0 c1 loc t* ⟹ *P c1*
shows *P c1*
using *assms* **unfolding** *next'-def* **by** *auto*

7.2 Auxiliary

lemma *next-change-multiplicity'-unique*:
assumes *n* ≠ *0*
and ∃ *t'*. *t'* ∈$_A$ *frontier* (*c-imp c loc*) ∧ *t'* ≤ *t*
shows ∃!*c'*. *next-change-multiplicity' c c' loc t n*
proof −
let *?pointstamps'* = (*c-pts c*)(*loc* := *update-zmultiset* (*c-pts c loc*) *t n*)
let *?worklist'* = λ*loc'*. *c-work c loc'* + *frontier-changes* (*?pointstamps' loc'*) (*c-pts c loc'*)
let *?c'* = *c*(|*c-pts* := *?pointstamps'*, *c-work* := *?worklist'*|)
from *assms* **have** *next-change-multiplicity' c ?c' loc t n*
by (*auto simp*: *next-change-multiplicity'-def intro*!: *configuration.equality*)
then show *?thesis*
by (*rule ex1I*[*of* - *?c'*])
(*auto simp*: *next-change-multiplicity'-def intro*!: *configuration.equality*)
qed

lemma *frontier-change-zmset-frontier*:
assumes *t* ∈$_A$ *frontier M1* − *frontier M0*
shows *zcount* (*zmset-frontier M1*) *t* = *1* ∧ *zcount* (*zmset-frontier M0*) *t* = *0*
using *assms* **by** (*clarsimp simp*: *ac-Diff-iff*) (*simp add*: *member-antichain.rep-eq*)

lemma *frontier-empty*[*simp*]: *frontier* {#}$_z$ = {}$_A$
by *transfer' simp*

lemma *zmset-frontier-empty*[*simp*]: *zmset-frontier* {#}$_z$ = {#}$_z$
by *simp*

lemma *after-summary-empty*[*simp*]: *after-summary* {#}$_z$ *S* = {#}$_z$
by (*simp add*: *after-summary-def*)

lemma *after-summary-empty-summary*[*simp*]: *after-summary M* $\{\}_A$ = $\{\#\}_z$
by (*simp add*: *after-summary-def*)

lemma *mem-frontier-diff*:
assumes *t* $\in_A$ *frontier M* − *frontier N*
shows *zcount* (*frontier-changes M N*) *t* = *1*
proof −
note *assms*
then have *t*: *t* $\in_A$ *frontier M t* $\notin_A$ *frontier N*
using *ac-Diff-iff* **by** *blast*+
from *t*(*1*) **have** *zcount* (*zmset-frontier M*) *t* = *1*
by (*simp add*: *member-antichain.rep-eq*)
moreover from *t*(*2*) **have** *zcount* (*zmset-frontier N*) *t* = *0*
by (*simp add*: *member-antichain.rep-eq*)
ultimately show *zcount* (*frontier-changes M N*) *t* = *1*
by *simp*
qed

lemma *mem-frontier-diff′*:
assumes *t* $\in_A$ *frontier N* − *frontier M*
shows *zcount* (*frontier-changes M N*) *t* = −*1*
proof −
note *assms*
then have *t*: *t* $\in_A$ *frontier N t* $\notin_A$ *frontier M*
using *ac-Diff-iff* **by** *blast*+
from *t*(*2*) **have** *zcount* (*zmset-frontier M*) *t* = *0*
by (*simp add*: *member-antichain.rep-eq*)
moreover from *t*(*1*) **have** *zcount* (*zmset-frontier N*) *t* = *1*
by (*simp add*: *member-antichain.rep-eq*)
ultimately show *zcount* (*frontier-changes M N*) *t* = −*1*
by *simp*
qed

lemma *not-mem-frontier-diff*:
assumes *t* $\notin_A$ *frontier M* − *frontier N*
and *t* $\notin_A$ *frontier N* − *frontier M*
shows *zcount* (*frontier-changes M N*) *t* = *0*
proof −
{ assume *M*: *t* $\in_A$ *frontier M*
with *assms* **have** *N*: *t* $\in_A$ *frontier N*
by (*auto dest*: *ac-notin-Diff*)
from *M N* **have** *zcount* (*zmset-frontier M*) *t* = *1 zcount* (*zmset-frontier N*) *t* = *1*
by (*simp-all add*: *member-antichain.rep-eq*)
then have *zcount* (*frontier-changes M N*) *t* = *0*
by *simp*
}
moreover
{ assume *M*: *t* $\notin_A$ *frontier M*

with *assms* **have** N: $t \notin_A$ *frontier* N
by (*auto dest*: *ac-notin-Diff*)
from M N **have** *zcount* (*zmset-frontier* M) $t = 0$ *zcount* (*zmset-frontier* N) $t = 0$
by (*simp-all add*: *member-antichain.rep-eq*)
then have *zcount* (*frontier-changes* M N) $t = 0$
by *simp*
}
ultimately show *zcount* (*frontier-changes* M N) $t = 0$
by *blast*
qed

lemma *mset-neg-after-summary*: *mset-neg* $M = \{\#\} \Longrightarrow$ *mset-neg* (*after-summary* M S) $= \{\#\}$
by (*auto intro*: *mset-neg-image-zmset mset-neg-sum-set simp*: *after-summary-def*)

— Changes in loc's frontier are reflected in the worklist of loc'.
lemma *next-p-frontier-change*:
assumes *next-propagate'* *c0* *c1* *loc* *t*
and *summary loc loc'* $\neq \{\}_A$
shows *c-work c1 loc'* =
c-work c0 loc'
+ *after-summary*
(*frontier-changes* (*c-imp c1 loc*) (*c-imp c0 loc*))
(*summary loc loc'*)
using *assms* **by** (*auto simp*: *summary-self next-propagate'-def intro*!: *configuration.equality*)

lemma *after-summary-union*: *after-summary* $(M + N)$ S = *after-summary* M S + *after-summary* N S
by (*simp add*: *sum.distrib after-summary-def*)

7.3 Invariants

7.3.1 Invariant: *inv-imps-work-sum*

abbreviation *union-frontiers* :: (*'loc*, *'t*) *configuration* ⇒ *'loc* ⇒ *'t zmultiset* **where**
union-frontiers c loc ≡
$(\sum loc' \in UNIV.$ *after-summary* (*zmset-frontier* (*c-imp c loc'*)) (*summary loc' loc*))

— Implications + worklist is equal to the frontiers of pointstamps and all preceding nodes (after accounting for summaries).
definition *inv-imps-work-sum* :: (*'loc*, *'t*) *configuration* ⇒ *bool* **where**
inv-imps-work-sum c ≡
$\forall$ *loc*. *c-imp c loc* + *c-work c loc*
= *zmset-frontier* (*c-pts c loc*) + *union-frontiers c loc*

— Version with zcount is easier to reason with
definition *inv-imps-work-sum-zcount* :: (*'loc*, *'t*) *configuration* ⇒ *bool* **where**

```
  inv-imps-work-sum-zcount c ≡
    ∀ loc t. zcount (c-imp c loc + c-work c loc) t
          = zcount (zmset-frontier (c-pts c loc) + union-frontiers c loc) t

lemma inv-imps-work-sum-zcount: inv-imps-work-sum c ⟷ inv-imps-work-sum-zcount
c
  unfolding inv-imps-work-sum-zcount-def inv-imps-work-sum-def
  by (simp add: zmultiset-eq-iff)


lemma union-frontiers-nonneg: 0 ≤ zcount (union-frontiers c loc) t
  apply (subst zcount-sum)
  apply (rule sum-nonneg)
  apply simp
  apply (rule mset-neg-zcount-nonneg)
  apply (rule mset-neg-after-summary)
  apply simp
  done

lemma next-p-union-frontier-change:
  assumes next-propagate' c0 c1 loc t
    and summary loc loc' ≠ {}_A
  shows union-frontiers c1 loc' =
        union-frontiers c0 loc'
         + after-summary
             (frontier-changes (c-imp c1 loc) (c-imp c0 loc))
             (summary loc loc')
  using assms
  apply (subst zmultiset-eq-iff)
  apply (rule allI)
  subgoal for x
    apply (simp del: zcount-of-mset image-zmset-Diff)
    apply (subst (1 2) zcount-sum)
    apply (rule Sum-eq-pick-changed-elem[of UNIV loc])
      apply simp
     apply simp
    subgoal
      apply (subst zcount-union[symmetric])
      apply (subst after-summary-union[symmetric])
      apply simp
      done
    apply (auto simp: next-propagate'-def)
    done
  done

— init-config satisfies inv-imps-work-sum
lemma init-imp-inv-imps-work-sum: init-config c ⟹ inv-imps-work-sum c
  by (simp add: inv-imps-work-sum-def init-config-def)
```

```
— CM preserves inv-imps-work-sum
lemma cm-preserves-inv-imps-work-sum:
  assumes next-change-multiplicity' c0 c1 loc t n
    and   inv-imps-work-sum c0
  shows   inv-imps-work-sum c1
proof −
  — Given CM at loc, t, we show result for loc', t'
  { fix loc t loc' t' n
    assume cm': next-change-multiplicity' c0 c1 loc t n
    note cm = this[unfolded next-change-multiplicity'-def]
    from cm have unchanged-imps: unchanged c-imp c0 c1
      by simp
    assume inv-imps-work-sum c0
    then have iiws': inv-imps-work-sum-zcount c0
      by (simp add: inv-imps-work-sum-zcount)
    note iiws = iiws'[unfolded inv-imps-work-sum-zcount-def, THEN spec2]
    have unchanged-union: union-frontiers c1 locX = union-frontiers c0 locX for
locX
      using unchanged-imps by (auto intro: sum.cong)
        — For locations other than loc nothing changes.
    { assume loc: loc' ≠ loc
       note iiws = iiws'[unfolded inv-imps-work-sum-zcount-def, THEN spec2, of
loc' t']
      from loc cm have unchanged-worklist:
        zcount (c-work c1 loc') t' = zcount (c-work c0 loc') t'
        by simp
      from loc cm have unchanged-frontier:
        zcount (zmset-frontier (c-pts c1 loc')) t'
       = zcount (zmset-frontier (c-pts c0       loc')) t'
        by simp
      with loc have
        zcount (c-imp c1 loc' + c-work c1 loc') t'
            = zcount (zmset-frontier (c-pts c1 loc')
                + union-frontiers c1 loc') t'
        apply (subst (1 2) zcount-union)
        unfolding
          unchanged-imps
          unchanged-union
          unchanged-frontier
          unchanged-worklist
        apply (subst (1 2) zcount-union[symmetric])
        apply (rule iiws)
        done
    }
    moreover
     — For pointstamps at location loc we make a case distinction on whether their
"status" in the frontier has changed.
    { assume loc: loc' = loc
      note iiws = iiws'[unfolded inv-imps-work-sum-zcount-def, simplified, THEN
```

spec, *of loc*, *simplified*]
— If t appeared in the frontier
{ assume *t'*: $t' \in_A$ *frontier* (*c-pts c1 loc*) − *frontier* (*c-pts c0 loc*)
note *t'*[*THEN mem-frontier-diff*]
— then the worklist at t increased by 1
then have *zcount* (*c-work c1 loc*) *t'* = *zcount* (*c-work c0 loc*) *t'* + *1*
using *cm* **by** *auto*
— and the frontier at t increased by 1
moreover
have *zcount* (*zmset-frontier* (*c-pts c1 loc*)) *t'*
= *zcount* (*zmset-frontier* (*c-pts c0 loc*)) *t'* + *1*
using *t'*[*THEN frontier-change-zmset-frontier*] **by** *simp*
— and the sum didn't change
moreover note *unchanged-union*
— hence, the invariant is preserved.
ultimately have
zcount (*c-imp c1 loc* + *c-work c1 loc*) *t'*
= *zcount* (*zmset-frontier* (*c-pts c1 loc*)
+ *union-frontiers c1 loc*) *t'*
using *iiws unchanged-imps* **by** *simp*
}
moreover
— If t disappeared from the frontier
{ assume *t'*: $t' \in_A$ *frontier* (*c-pts c0 loc*) − *frontier* (*c-pts c1 loc*)
note *t'*[*THEN mem-frontier-diff'*]
— then the worklist at t decreased by 1
then have *zcount* (*c-work c1 loc*) *t'* = *zcount* (*c-work c0 loc*) *t'* − *1*
using *cm* **by** (*auto simp*: *ac-Diff-iff*)
— and the frontier at t decreased by 1
moreover
have *zcount* (*zmset-frontier* (*c-pts c1 loc*)) *t'*
= *zcount* (*zmset-frontier* (*c-pts c0 loc*)) *t'* − *1*
using *t'*[*THEN frontier-change-zmset-frontier*] **by** *simp*
— and the sum didn't change
moreover note *unchanged-union*
— hence, the invariant is preserved.
ultimately have
zcount (*c-imp c1 loc* + *c-work c1 loc*) *t'*
= *zcount* (*zmset-frontier* (*c-pts c1 loc*)
+ *union-frontiers c1 loc*) *t'*
using *iiws unchanged-imps* **by** *simp*
}
moreover
— If t's multiplicity in the frontier didn't change
{ assume *a1*: ¬ $t' \in_A$ *frontier* (*c-pts c1 loc*) − *frontier* (*c-pts c0 loc*)
assume *a2*: ¬ $t' \in_A$ *frontier* (*c-pts c0 loc*) − *frontier* (*c-pts c1 loc*)
from *a1 a2* **have** *zcount* (*frontier-changes* (*c-pts c1 loc*) (*c-pts c0 loc*)) *t'* =
0
by (*intro not-mem-frontier-diff*)

— then the worklist at t didn't change
with *cm* **have** *zcount* (*c-work c1 loc*) *t′* = *zcount* (*c-work c0 loc*) *t′*
by (*auto simp*: *ac-Diff-iff*)
— and the frontier at t didn't change
moreover
have *zcount* (*zmset-frontier* (*c-pts c1 loc*)) *t′*
= *zcount* (*zmset-frontier* (*c-pts c0 loc*)) *t′*
using *a1 a2*
apply (*clarsimp simp*: *member-antichain.rep-eq dest*!: *ac-notin-Diff*)
apply (*metis ac-Diff-iff count-mset-set*(*1*) *count-mset-set*(*3*) *finite-set-antichain member-antichain.rep-eq*)
done
— and the sum didn't change
moreover note *unchanged-union*
— hence, the invariant is preserved.
ultimately have
zcount (*c-imp c1 loc* + *c-work c1 loc*) *t′*
= *zcount* (*zmset-frontier* (*c-pts c1 loc*)
+ *union-frontiers c1 loc*) *t′*
using *iiws unchanged-imps* **by** *simp*
}
ultimately have
zcount (*c-imp c1 loc′* + *c-work c1 loc′*) *t′*
= *zcount* (*zmset-frontier* (*c-pts c1 loc′*)
+ *union-frontiers c1 loc′*) *t′*
using *loc* **by** *auto*
}
ultimately have
zcount (*c-imp c1 loc′* + *c-work c1 loc′*) *t′*
= *zcount* (*zmset-frontier* (*c-pts c1 loc′*)
+ *union-frontiers c1 loc′*) *t′*
by *auto*
}
with *assms* **show** *?thesis*
by (*auto simp*: *Let-def inv-imps-work-sum-zcount inv-imps-work-sum-zcount-def*)
qed

— PR preserves *inv-imps-work-sum*
lemma *p-preserves-inv-imps-work-sum*:
assumes *next-propagate′ c0 c1 loc t*
and *inv-imps-work-sum c0*
shows *inv-imps-work-sum c1*
proof −
— Given *next-propagate′* for loc, t, we show the result for loc', t'.
{ fix *loc t loc′ t′*
assume *p′*: *next-propagate′ c0 c1 loc t*
note *p* = *this*[*unfolded next-propagate′-def*]
from *p* **have** *unchanged-ps*: *unchanged c-pts c0 c1*
by *simp*

assume *inv-imps-work-sum c0*
then have *iiws'*: *inv-imps-work-sum-zcount c0*
by (*simp add*: *inv-imps-work-sum-zcount*)
note *iiws* = *iiws'*[*unfolded inv-imps-work-sum-zcount-def*, *THEN spec2*]
{ assume *loc*: *loc*=*loc'*
note *iiws*
moreover note *unchanged-ps*
— The t entries in loc's worklist are shifted to the implications.
moreover from *p* **have** *zcount* (*c-work c1 loc*) *t* = *0*
by *simp*
moreover from *p* **have**
zcount (*c-imp c1 loc*) *t*
= *zcount* (*c-imp c0 loc*) *t* + *zcount* (*c-work c0 loc*) *t*
by *simp*
— Since the implications of other locations don't change and loc can't have an edge to itself, $\lambda c\ loc.\ \sum loc' \in UNIV.$ *after-summary* (*zmset-of* (*mset-set* (*set-antichain* (*frontier* (*c-imp c loc'*))))) (*summary loc' loc*) at loc doesn't change.
moreover from *p* **have** *union-frontiers c1 loc* = *union-frontiers c0 loc*
using *summary-self* **by** (*auto intro*!: *sum.cong arg-cong*[**where** *f* = *Sum*])
— For all the other timestamps the worklist and implications don't change.
moreover from *p* **have**
tX ≠ *t* ⟹ *zcount* (*c-work c1 loc*) *tX* = *zcount* (*c-work c0 loc*) *tX* **for** *tX*
by *simp*
moreover from *p* **have**
tX ≠ *t* ⟹ *zcount* (*c-imp c1 loc*) *tX* = *zcount* (*c-imp c0 loc*) *tX* **for** *tX*
by *simp*
ultimately have
zcount (*c-imp c1 loc'* + *c-work c1 loc'*) *t'*
= *zcount* (*zmset-frontier* (*c-pts c1 loc'*) + *union-frontiers c1 loc'*) *t'*
unfolding *loc*
by (*cases t*=*t'*) *simp-all*
}
moreover
{ assume *loc*: *loc*≠*loc'*
— The implications are unchanged at all locations other than loc.
from *p loc* **have** *unchanged-imps*: *c-imp c1 loc'* = *c-imp c0 loc'*
by *simp*
{ assume *sum*: *summary loc loc'* = $\{\}_A$
note *iiws*
moreover note *unchanged-ps*
moreover note *unchanged-imps*
— The worklist only changes if loc, loc' are connected.
moreover from *p loc sum* **have** *c-work c1 loc'* = *c-work c0 loc'*
by *simp*
— Since the implications only change at loc and loc is not connected to loc', $\lambda c\ loc.\ \sum loc' \in UNIV.$ *after-summary* (*zmset-of* (*mset-set* (*set-antichain* (*frontier* (*c-imp c loc'*))))) (*summary loc' loc*) doesn't change.
moreover from *p loc sum* **have** *union-frontiers c1 loc'* = *union-frontiers c0 loc'*

by (*auto intro*!: *sum.cong arg-cong*[**where** *f* = *Sum*])
ultimately have
zcount (*c-imp c1 loc′* + *c-work c1 loc′*) *t′*
= *zcount* (*zmset-frontier* (*c-pts c1 loc′*) + *union-frontiers c1 loc′*) *t′*
by *simp*
}
moreover
{ assume *sum*: *summary loc loc′* ≠ {}$_A$
— $\lambda c\ loc.\ \sum loc' \in UNIV.\ after\text{-}summary\ (zmset\text{-}of\ (mset\text{-}set\ (set\text{-}antichain\ (frontier\ (c\text{-}imp\ c\ loc')))))\ (summary\ loc'\ loc)$ at loc' changed by whatever amount the frontier changed.
note *iiws*
unchanged-imps
unchanged-ps
moreover from *p′ sum* **have**
union-frontiers c1 loc′ =
union-frontiers c0 loc′
+ *after-summary*
(*zmset-frontier* (*c-imp c1 loc*) − *zmset-frontier* (*c-imp c0 loc*))
(*summary loc loc′*)
by (*auto intro*!: *next-p-union-frontier-change*)
— The worklist at loc' changed by the same amount
moreover from *p′ sum* **have**
c-work c1 loc′ =
c-work c0 loc′
+ *after-summary*
(*zmset-frontier* (*c-imp c1 loc*) − *zmset-frontier* (*c-imp c0 loc*))
(*summary loc loc′*)
by (*auto intro*!: *next-p-frontier-change*)
— The two changes cancel out.
ultimately have
zcount (*c-imp c1 loc′* + *c-work c1 loc′*) *t′*
= *zcount* (*zmset-frontier* (*c-pts c1 loc′*) + *union-frontiers c1 loc′*) *t′*
by *simp*
}
ultimately have
zcount (*c-imp c1 loc′* + *c-work c1 loc′*) *t′*
= *zcount* (*zmset-frontier* (*c-pts c1 loc′*) + *union-frontiers c1 loc′*) *t′*
by *auto*
}
ultimately have
zcount (*c-imp c1 loc′* + *c-work c1 loc′*) *t′*
= *zcount* (*zmset-frontier* (*c-pts c1 loc′*) + *union-frontiers c1 loc′*) *t′*
by (*cases loc*=*loc′*) *auto*
}
with *assms* **show** *?thesis*
by (*auto simp*: *next-propagate′-def Let-def inv-imps-work-sum-zcount inv-imps-work-sum-zcount-def*)
qed

lemma *next-preserves-inv-imps-work-sum*:
assumes *next s*
and *holds inv-imps-work-sum s*
shows *nxt* (*holds inv-imps-work-sum*) *s*
using *assms*
cm-preserves-inv-imps-work-sum
p-preserves-inv-imps-work-sum
by (*simp*, *cases rule*: *next′-inv*)

lemma *spec-imp-iiws*: *spec s* $\Longrightarrow$ *alw* (*holds inv-imps-work-sum*) *s*
using *init-imp-inv-imps-work-sum next-preserves-inv-imps-work-sum*
by (*auto intro*: *alw-invar simp*: *alw-mono spec-def*)

7.3.2 Invariant: *inv-imp-plus-work-nonneg*

There is never an update in the worklist that could cause implications to become negative.

definition *inv-imp-plus-work-nonneg* **where**
inv-imp-plus-work-nonneg c $\equiv \forall$ *loc t*. *0* $\leq$ *zcount* (*c-imp c loc*) *t* + *zcount* (*c-work c loc*) *t*

lemma *iiws-imp-iipwn*:
assumes *inv-imps-work-sum c*
shows *inv-imp-plus-work-nonneg c*
proof −
{
fix *loc*
fix *t*
assume *inv-imps-work-sum c*
then have *iiws′*: *inv-imps-work-sum-zcount c*
by (*simp add*: *inv-imps-work-sum-zcount*)
note *iiws* = *iiws′*[*unfolded inv-imps-work-sum-zcount-def*, *THEN spec2*]
have *0* $\leq$ *zcount* (*union-frontiers c loc*) *t*
by (*simp add*: *union-frontiers-nonneg*)
with *iiws* **have** *0* $\leq$ *zcount* (*c-imp c loc* + *c-work c loc*) *t*
by *simp*
}
with *assms* **show** *?thesis*
by (*simp add*: *inv-imp-plus-work-nonneg-def*)
qed

lemma *spec-imp-iipwn*: *spec s* $\Longrightarrow$ *alw* (*holds inv-imp-plus-work-nonneg*) *s*
using *spec-imp-iiws iiws-imp-iipwn*
alw-mono holds-mono
by *blast*

7.3.3 Invariant: *inv-implications-nonneg*

lemma *init-imp-inv-implications-nonneg*:

assumes *init-config c*
shows *inv-implications-nonneg c*
using *assms* **by** (*simp add*: *init-config-def inv-implications-nonneg-def*)

lemma *cm-preserves-inv-implications-nonneg*:
assumes *next-change-multiplicity′ c0 c1 loc t n*
and *inv-implications-nonneg c0*
shows *inv-implications-nonneg c1*
using *assms* **by** (*simp add*: *next-change-multiplicity′-def inv-implications-nonneg-def*)

lemma *p-preserves-inv-implications-nonneg*:
assumes *next-propagate′ c0 c1 loc t*
and *inv-implications-nonneg c0*
and *inv-imp-plus-work-nonneg c0*
shows *inv-implications-nonneg c1*
using *assms*
by (*auto simp*: *next-propagate′-def Let-def inv-implications-nonneg-def inv-imp-plus-work-nonneg-def*)

lemma *next-preserves-inv-implications-nonneg*:
assumes *next s*
and *holds inv-implications-nonneg s*
and *holds inv-imp-plus-work-nonneg s*
shows *nxt* (*holds inv-implications-nonneg*) *s*
using *assms*
cm-preserves-inv-implications-nonneg
p-preserves-inv-implications-nonneg
by (*simp*, *cases rule*: *next′-inv*)

lemma *alw-inv-implications-nonneg*: *spec s* $\Longrightarrow$ *alw* (*holds inv-implications-nonneg*) *s*
apply (*frule spec-imp-iipwn*)
unfolding *spec-def*
apply (*rule alw-invar*)
using *init-imp-inv-implications-nonneg* **apply** *auto* []
using *next-preserves-inv-implications-nonneg*
apply (*metis* (*no-types*, *lifting*) *alw-iff-sdrop*)
done

lemma *after-summary-Diff*: *after-summary* (*M* − *N*) *S* = *after-summary M S* − *after-summary N S*
by (*simp add*: *sum-subtractf after-summary-def*)

lemma *mem-zmset-frontier*: *x* $\in\#_z$ *zmset-frontier M* $\longleftrightarrow$ *x* $\in_A$ *frontier M*
by *transfer simp*

lemma *obtain-frontier-elem*:
assumes *0* < *zcount M t*
obtains *u* **where** *u* $\in_A$ *frontier M u* ≤ *t*
using *assms* **by** (*atomize-elim*, *transfer*)

(*auto simp*: *minimal-antichain-def dest*: *order-zmset-exists-foundation*)

lemma *frontier-unionD*: $t \in_A$ *frontier* $(M+N) \Longrightarrow 0 <$ *zcount* $M\ t \vee 0 <$ *zcount* $N\ t$
by *transfer'* (*auto simp*: *minimal-antichain-def*)

lemma *ps-frontier-in-imps-wl*:
assumes *inv-imps-work-sum c*
and $0 <$ *zcount* (*zmset-frontier* (*c-pts c loc*)) *t*
shows $0 <$ *zcount* (*c-imp c loc* + *c-work c loc*) *t*
proof −
have $0 \leq$ *zcount* (*union-frontiers c loc*) *t*
by (*rule union-frontiers-nonneg*)
with *assms*(*2*) **show** *?thesis*
using *assms*(*1*)[*unfolded inv-imps-work-sum-def*, *THEN spec*, *of loc*]
by *fastforce*
qed

lemma *obtain-elem-frontier*:
assumes $0 <$ *zcount* $M\ t$
obtains *s* **where** $s \leq t \wedge s \in_A$ *frontier* M
by (*rule order-finite-set-obtain-foundation*[*of* {*s*. *zcount* $M\ s > 0$} *t thesis*])
(*auto simp*: *assms antichain-inverse frontier-def member-antichain.rep-eq in-minimal-antichain*)

lemma *obtain-elem-zmset-frontier*:
assumes $0 <$ *zcount* $M\ t$
obtains *s* **where** $s \leq t \wedge 0 <$ *zcount* (*zmset-frontier* M) *s*
using *assms* **by** (*auto simp*: *member-antichain.rep-eq intro*: *obtain-elem-frontier*)

lemma *ps-in-imps-wl*:
assumes *inv-imps-work-sum c*
and $0 <$ *zcount* (*c-pts c loc*) *t*
obtains *s* **where** $s \leq t \wedge 0 <$ *zcount* (*c-imp c loc* + *c-work c loc*) *s*
proof *atomize-elim*
note *iiws* = *assms*(*1*)[*unfolded inv-imps-work-sum-def*, *THEN spec*, *of loc*]
obtain *u* **where** *u*: $u \leq t \wedge u \in_A$ *frontier* (*c-pts c loc*)
using *assms*(*2*) *obtain-elem-frontier* **by** *blast*
with *assms*(*1*) **have** $0 <$ *zcount* (*c-imp c loc* + *c-work c loc*) *u*
apply (*intro ps-frontier-in-imps-wl*[*of c loc u*])
apply (*auto intro*: *iffD1*[*OF member-antichain.rep-eq*])
done
with *u* **show** $\exists s{\leq}t.\ 0 <$ *zcount* (*c-imp c loc* + *c-work c loc*) *s*
by (*auto intro*: *exI*[*of* - *u*])
qed

lemma *zero-le-after-summary-single*[*simp*]: $0 \leq$ *zcount* (*after-summary* $\{\#t\#\}_z$ *S*) *x*
by (*auto intro*: *zero-le-sum-single simp*: *after-summary-def*)

lemma *one-le-zcount-after-summary*: $s \in_A S \Longrightarrow 1 \leq$ *zcount* (*after-summary* $\{\#t\#\}_z$ *S*) (*results-in t s*)
unfolding *image-zmset-single after-summary-def*
apply (*subst zcount-sum*)
apply (*subst forw-subst*[*of 1 zcount* $\{\#$*results-in t s*$\#\}_z$ (*results-in t s*)])
apply *simp*
apply (*rule sum-nonneg-leq-bound*[*of set-antichain S* $\lambda u.$ *zcount* $\{\#$*results-in t u*$\#\}_z$ (*results-in t s*) - *s*])
apply (*auto simp*: *member-antichain.rep-eq*)
done

lemma *zero-lt-zcount-after-summary*: $s \in_A S \Longrightarrow 0 <$ *zcount* (*after-summary* $\{\#t\#\}_z$ *S*) (*results-in t s*)
apply (*subst int-one-le-iff-zero-less*[*symmetric*])
apply (*intro one-le-zcount-after-summary*)
apply *simp*
done

lemma *pos-zcount-after-summary*:
$(\bigwedge t.\ 0 \leq$ *zcount M t*$) \Longrightarrow 0 <$ *zcount M t* $\Longrightarrow s \in_A S \Longrightarrow 0 <$ *zcount* (*after-summary M S*) (*results-in t s*)
by (*auto intro*!: *sum-pos2 pos-zcount-image-zmset simp*: *member-antichain.rep-eq zcount-sum after-summary-def*)

lemma *after-summary-nonneg*: $(\bigwedge t.\ 0 \leq$ *zcount M t*$) \Longrightarrow 0 \leq$ *zcount* (*after-summary M S*) *t*
by (*auto simp*: *zcount-sum after-summary-def intro*: *sum-nonneg*)

lemma *after-summary-zmset-of-nonneg*[*simp*, *intro*]: $0 \leq$ *zcount* (*after-summary* (*zmset-of M*) *S*) *t*
by (*simp add*: *mset-neg-image-zmset mset-neg-sum-set mset-neg-zcount-nonneg after-summary-def*)

lemma *pos-zcount-union-frontiers*:
zcount (*after-summary* (*zmset-frontier* (*c-imp c l1*)) (*summary l1 l2*)) (*results-in t s*)
$\leq$ *zcount* (*union-frontiers c l2*) (*results-in t s*)
apply (*subst zcount-sum*)
apply (*rule member-le-sum*)
apply (*auto intro*!: *pos-zcount-image-zmset*)
done

lemma *after-summary-Sum-fun*: *finite MM* $\Longrightarrow$ *after-summary* $(\sum M{\in}MM.\ f\ M)$ $A = (\sum M{\in}MM.$ *after-summary* $(f\ M)\ A)$
by (*induct MM rule*: *finite-induct*) (*auto simp*: *after-summary-union*)

lemma *after-summary-obtain-pre*:
assumes $\bigwedge t.\ 0 \leq$ *zcount M t*

```
    and   0 < zcount (after-summary M S) t
  obtains t' s where 0 < zcount M t' results-in t' s = t s ∈A S
  using assms unfolding after-summary-def
  apply atomize-elim
  apply (subst (asm) zcount-sum)
  apply (drule sum-pos-ex-elem-pos)
  apply clarify
  subgoal for s
    apply (subst ex-comm)
    apply (rule exI[of - s])
    apply (drule pos-image-zmset-obtain-pre[rotated])
     apply simp
    apply (simp add: member-antichain.rep-eq)
    done
  done

lemma empty-antichain[dest]: x ∈A antichain {} ⟹ False
  by (metis empty-antichain.abs-eq mem-antichain-nonempty)

definition impWitnessPath where
  impWitnessPath c loc1 loc2 xs t = (
    path loc1 loc2 xs ∧
    distinct xs ∧
    (∃ t'. t' ∈A frontier (c-imp c loc1) ∧ t = results-in t' (sum-path-weights xs) ∧
    (∀ k<length xs. (∃ t. t ∈A frontier (c-imp c (TO (xs ! k))) ∧ t = results-in t'
(sum-path-weights (take (k+1) xs))))))

lemma impWitnessPathEx:
  assumes t ∈A frontier (c-imp c loc2)
  shows (∃ loc1 xs. impWitnessPath c loc1 loc2 xs t)
proof −
  have 1: path loc2 loc2 [] by (simp add: path0)
  have 2: distinct [] by auto
  have 0 = sum-path-weights [] using foldr-Nil list.map(1) by auto
 then have 3: t = results-in t (sum-path-weights []) by (simp add: results-in-zero)
  with 1 2 assms show ?thesis
    unfolding impWitnessPath-def
    by (force simp: results-in-zero)
qed

definition longestImpWitnessPath where
  longestImpWitnessPath  c loc1 loc2 xs t = (
  impWitnessPath c loc1 loc2 xs t ∧
  (∀ loc' xs'. impWitnessPath c loc' loc2 xs' t ⟶ length (xs') ≤ length (xs)))

lemma finite-edges: finite {(loc1,s,loc2). s ∈A summary loc1 loc2}
proof −
  let ?sums = (⋃ ((λ(loc1,loc2). set-antichain (summary loc1 loc2)) ‘ UNIV))
  have finite ?sums
```

```
    by auto
  then have finite ((UNIV::'loc set) × ?sums × (UNIV::'loc set))
    by auto
  moreover have {(loc1,s,loc2). s ∈A summary loc1 loc2} ⊆ ((UNIV::'loc set) ×
?sums × (UNIV::'loc set))
    by (force simp: member-antichain.rep-eq)
  ultimately show ?thesis
    by (rule rev-finite-subset)
qed

lemma longestImpWitnessPathEx:
  assumes t ∈A frontier (c-imp c loc2)
  shows (∃ loc1 xs. longestImpWitnessPath c loc1 loc2 xs t)
proof −
  define paths where paths = {(loc1, xs). impWitnessPath c loc1 loc2 xs t}
  with impWitnessPathEx[OF assms] obtain p where p ∈ paths by auto
  have ∀ p. p ∈ paths ⟶ (length ∘ snd) p  < card {(loc1,s,loc2). s ∈A summary
loc1 loc2} + 1
  proof (intro allI impI)
    fix p
    assume p:  p ∈ paths
    then show (length ∘ snd) p < card {(loc1,s,loc2). s ∈A summary loc1 loc2} +
1
        by (auto simp: paths-def impWitnessPath-def less-Suc-eq-le finite-edges
path-edge
         dest!: distinct-card[symmetric] intro!: card-mono)
  qed
  from ex-has-greatest-nat[OF ‹p ∈ paths› this] show ?thesis
    by (auto simp: paths-def longestImpWitnessPath-def)
qed

lemma path-first-loc: path l1 l2 xs ⟹ xs ≠ [] ⟹ xs ! 0 = (l1',s,l2') ⟹ l1 = l1'
proof (induct arbitrary: l1' s l2 rule: path.induct)
  case (path0 l1 l2)
  then show ?case by auto
next
  case (path l1 l2 xs lbl l3)
  then show ?case
    apply (cases xs=[])
     apply (auto elim: path0E) []
    apply (rule path(2)[of l1' s])
    by (auto simp: nth-append)
qed

lemma find-witness-from-frontier:
  assumes t ∈A frontier (c-imp c loc2)
    and inv-imps-work-sum c
  shows ∃ t' loc1 xs. (path loc1 loc2 xs ∧ t =  results-in t' (sum-path-weights xs) ∧
         (t' ∈A frontier (c-pts c loc1) ∨ 0 > zcount (c-work c loc1) t'))
```

proof −
obtain *loc1 xs* **where** *longestP*: *longestImpWitnessPath c loc1 loc2 xs t*
using *assms*(*1*) *longestImpWitnessPathEx* **by** *blast*
then obtain *t'* **where** *t'*: *t' ∈A frontier* (*c-imp c loc1*) *t* = *results-in t'* (*sum-path-weights xs*)
(∀ *k*<*length xs*. (∃ *t*. *t ∈A frontier* (*c-imp c* (*TO* (*xs* ! *k*))) ∧ *t* = *results-in t'* (*sum-path-weights* (*take* (*k*+*1*) *xs*))))
by (*auto simp add*: *longestImpWitnessPath-def impWitnessPath-def*)
from *t'*(*1*) **have** *cases*: *0* > *zcount* (*c-work c loc1*) *t'* ∨
(*t' ∈#z* (*zmset-frontier* (*c-pts c loc1*) + *union-frontiers c loc1*))
using *assms*(*2*)
apply (*clarsimp intro*!: *verit-forall-inst*(*6*) *simp*: *inv-imps-work-sum-def not-less*)
apply (*metis add-pos-nonneg mem-zmset-frontier member-frontier-pos-zmset obtain-frontier-elem zcount-empty zcount-ne-zero-iff zcount-union zmset-frontier-empty*)
done
then show *?thesis*
proof *cases*
assume *case1*: *0* > *zcount* (*c-work c loc1*) *t'*
then show *?thesis* **using** *t' longestP*
using *impWitnessPath-def longestImpWitnessPath-def dataflow-topology-axioms* **by** *blast*
next
assume *case2*: ¬*0* > *zcount* (*c-work c loc1*) *t'*
have (*t' ∈#z* (*zmset-frontier* (*c-pts c loc1*) + *union-frontiers c loc1*))
using *case2 cases* **by** *auto*
then have *case-split2*: (*t' ∈#z zmset-frontier* (*c-pts c loc1*)) ∨ (*t' ∈#z union-frontiers c loc1*)
by (*metis* (*no-types*, *lifting*) *add-diff-cancel-left' in-diff-zcount zcount-ne-zero-iff*)
then show *?thesis*
proof *cases*
assume *case2-1*: *t' ∈#z zmset-frontier* (*c-pts c loc1*)
have *t' ∈A frontier* (*c-pts c loc1*)
using *case2-1 mem-zmset-frontier* **by** *blast*
then show *?thesis*
using *t' impWitnessPath-def longestImpWitnessPath-def dataflow-topology-axioms longestP* **by** *blast*
next
assume ¬*t' ∈#z zmset-frontier* (*c-pts c loc1*)
then have *case2-2*: *t' ∈#z union-frontiers c loc1* **using** *case-split2* **by** *blast*
then obtain *loc0 t0 s0* **where** *loc0* : *t0 ∈A frontier* (*c-imp c loc0*)
s0 ∈A (*summary loc0 loc1*)
t' = *results-in t0 s0*
by (*fastforce simp*: *after-summary-def set-zmset-def zcount-sum*
member-antichain.rep-eq[*symmetric*] *zcount-image-zmset card-gt-0-iff*
simp del: *zcount-ne-zero-iff*
elim!: *sum.not-neutral-contains-not-neutral*)
let *?xs'* = (*loc0*, *s0*, *loc1*) # *xs*
have *path-xs*: *path loc1 loc2 xs*
using *impWitnessPath-def longestImpWitnessPath-def longestP* **by** *blast*

```
    have is-path-xs': path loc0 loc2 ?xs' using longestP
      apply (simp add: longestImpWitnessPath-def impWitnessPath-def)
      by (metis append-Cons append-Nil path-singleton path-trans loc0(2))
    have ∀ k<length ?xs'.
          results-in t0 (sum-path-weights (take (k+1) ?xs'))
          ∈A frontier (c-imp c (TO (?xs' ! k)))
      apply clarify
      subgoal for k
        apply (cases k=0)
        subgoal
          using loc0(3) t'(1) by (auto simp: results-in-zero)
        subgoal
          using t'(3)[rule-format, unfolded loc0(3) followed-by-summary, of k−1,
simplified]
          by auto
        done
      done
    note r = this[rule-format]
    have distinctxs: distinct xs
      using longestP
      by (simp add: longestImpWitnessPath-def impWitnessPath-def)
    then show ?thesis
    proof cases
      assume case-distinct: distinct ?xs'

      have t = results-in t0 (sum-path-weights ?xs') using longestP loc0(3)
        apply (simp add: longestImpWitnessPath-def impWitnessPath-def)
        by (simp add: followed-by-summary t'(2))
      then have impPath: impWitnessPath c loc0 loc2 ?xs' t
        using is-path-xs' case-distinct loc0(1)
        apply (simp add: impWitnessPath-def)
        using r by auto
      have length ?xs' > length xs by auto
      then have ¬ longestImpWitnessPath c loc1 loc2 xs t
        using impPath leD unfolding longestImpWitnessPath-def by blast
      then show ?thesis using longestP by blast
    next
      assume ¬ distinct ?xs'

      with distinctxs obtain k where k: TO (?xs' ! k) = loc0 k < length ?xs'
        apply atomize-elim
        apply clarsimp
        apply (subst (asm) in-set-conv-nth)
        apply clarify
        subgoal for i
          apply (cases i=0)
          subgoal
            using path-first-loc[OF path-xs]
            by force
```

```
          subgoal
            apply (rule exI[of - i])
            using path-xs
            apply (auto dest: path-to-eq-from[of - - xs i−1])
            done
          done
        done
      have results-in t0 (sum-path-weights (take (k+1) ?xs')) ∈A frontier (c-imp
c loc0)
        by (rule r[OF k(2), unfolded k(1)])
      moreover have t0 < results-in t0 (sum-path-weights (take (k+1) ?xs'))
        apply simp
        apply (rule no-zero-cycle[of loc0 take (k+1) ?xs' sum-path-weights (take
(k+1) ?xs') t0, simplified])
        using is-path-xs' k path-take-to by fastforce
      ultimately show ?thesis
        using loc0(1) frontier-comparable-False by blast
    qed
  qed
 qed
qed

lemma implication-implies-pointstamp:
 assumes t ∈A frontier (c-imp c loc)
   and   inv-imps-work-sum c
 shows  ∃ t' loc' s. s ∈A path-summary loc' loc ∧ t ≥ results-in t' s ∧
          (t' ∈A frontier (c-pts c loc') ∨ 0 > zcount (c-work c loc') t')
proof −
 obtain loc' t' xs where witness:
   path loc' loc xs
   t = results-in t' (sum-path-weights xs)
   t' ∈A frontier (c-pts c loc') ∨ 0 > zcount (c-work c loc') t'
   using assms find-witness-from-frontier by blast
 obtain s where s: s ∈A path-summary loc' loc  s ≤ (sum-path-weights xs)
   using witness(1) path-path-weight by blast
 then have t ≥ results-in t' s
   using witness(2) results-in-mono(2) by blast
 then show ?thesis
   using witness(3) s by auto
qed
```

7.4 Proof of Safety

```
lemma results-in-sum-path-weights-append:
  results-in t (sum-path-weights (xs @ [(loc2, s, loc3)])) = results-in (results-in t
(sum-path-weights xs)) s
 by (metis followed-by-summary sum-path-weights-append-singleton)

context
```

fixes $c :: ('loc, 't)$ *configuration*
begin

inductive *loc-imps-fw* **where**
loc-imps-fw loc loc (*c-imp c loc*) [] |
loc-imps-fw loc1 loc2 M xs $\Longrightarrow$ *s* $\in_A$ *summary loc2 loc3* $\Longrightarrow$ *distinct* (*xs* @ [(*loc2*,*s*,*loc3*)]) $\Longrightarrow$
loc-imps-fw loc1 loc3 ({# *results-in t s. t* $\in\#_z$ *M* #} + *c-imp c loc3*) (*xs* @ [(*loc2*,*s*,*loc3*)])

end

lemma *loc-imps-fw-conv-path*: *loc-imps-fw c loc1 loc2 M xs* $\Longrightarrow$ *path loc1 loc2 xs*
by (*induct rule*: *loc-imps-fw.induct*) (*auto intro*: *path.intros*)

lemma *path-conv-loc-imps-fw*: *path loc1 loc2 xs* $\Longrightarrow$ *distinct xs* $\Longrightarrow$ $\exists$ *M*. *loc-imps-fw c loc1 loc2 M xs*
proof (*induct rule*: *path.induct*)
case (*path0 l1 l2*)
then show *?case* **by** (*auto intro*: *loc-imps-fw.intros*)
next
case (*path l1 l2 xs lbl l3*)
then obtain *M* **where** *loc-imps-fw c l1 l2 M xs*
by *auto*
with *path* **show** *?case*
by (*force intro*: *loc-imps-fw.intros*(*2*))
qed

lemma *path-summary-conv-loc-imps-fw*:
s $\in_A$ *path-summary loc1 loc2* $\Longrightarrow$ $\exists$ *M xs. loc-imps-fw c loc1 loc2 M xs* $\wedge$ *sum-path-weights xs* = *s*
proof −
assume *path-sum*: *s* $\in_A$ *path-summary loc1 loc2*
then obtain *M xs* **where** *le*: *path loc1 loc2 xs loc-imps-fw c loc1 loc2 M xs sum-path-weights xs* $\leq$ *s distinct xs*
apply *atomize-elim*
apply (*drule path-weight-conv-path*)
apply *clarsimp*
apply (*drule path-distinct*)
apply *clarsimp*
subgoal for *ys xs*
apply (*rule exI*[*of* - *xs*])
apply (*rule conjI*, *assumption*)
apply (*drule path-conv-loc-imps-fw*[*of loc1 loc2 xs c*])
using *subseq-sum-weights-le* **apply** *auto*
done
done
show *?thesis*
proof (*cases sum-path-weights xs* = *s*)

```
    case True
    with le show ?thesis by auto
  next
    case False
    with le have sum-path-weights xs < s
      by auto
    with le(1) path-sum have False
      by (auto dest: path-weight-conv-path)
    then show ?thesis
      by blast
  qed
qed

lemma image-zmset-id[simp]: {#x. x ∈#z M#} = M
  by transfer (auto simp: equiv-zmset-def)

lemma sum-pos: finite M ⟹ ∀ x∈M. 0 ≤ f x ⟹ y ∈ M ⟹ 0 < (f y::-::ordered-comm-monoid-add)
⟹ 0 < (∑ x∈M. f x)
proof (induct M rule: finite-induct)
  case empty
  then show ?case by simp
next
  case (insert x F)
  then show ?case
    by (cases 0 < f x) (auto intro: sum-nonneg add-pos-nonneg add-nonneg-pos)
qed

lemma loc-imps-fw-M-in-implications:
  assumes loc-imps-fw c loc1 loc2 M xs
    and   inv-imps-work-sum c
    and   inv-implications-nonneg c
    and   ⋀loc. c-work c loc = {#}z
    and   0 < zcount M t
  shows   ∃ s. s ≤ t ∧ s ∈A frontier (c-imp c loc2)
  using assms(1,5)
proof (induct arbitrary: t rule: loc-imps-fw.induct)
  note iiws = assms(2)[unfolded inv-imps-work-sum-def assms(4), simplified, rule-format]
  case (1 loc t)
  then show ?case
    by (auto elim: obtain-elem-frontier)
next
  note iiws = assms(2)[unfolded inv-imps-work-sum-def assms(4), simplified, rule-format]
  case (2 loc1 loc2 M xs s loc3 t)
  from 2(5) have disj: 0 < zcount {# results-in t s. t ∈#z M #} t ∨ 0 < zcount
(c-imp c loc3) t
    by auto
  show ?case
  proof −
    { assume 0 < zcount {# results-in t s. t ∈#z M #} t
```

```
      then obtain t' where t': results-in t' s = t 0 < zcount M t'
        apply atomize-elim
        apply (rule ccontr)
        apply (subst (asm) zcount-image-zmset)
        apply (clarsimp simp: vimage-def)
      apply (metis (mono-tags, lifting) Int-iff mem-Collect-eq sum-pos-ex-elem-pos)
        done
      obtain u where u: u ≤ t' u ∈A frontier (c-imp c loc2)
        by atomize-elim (rule 2(2)[OF t'(2)])
      then have riu-le-rit': results-in u s ≤ t
        by (simp add: t'(1)[symmetric] results-in-mono)
      from u have 0 < zcount (union-frontiers c loc3) (results-in u s)
        apply (subst zcount-sum)
        apply (rule sum-pos[where y=loc2])
          apply simp-all [3]
        apply (clarsimp simp: after-summary-def)
        apply (subst zcount-sum)
        apply (rule sum-pos[where y=s])
        using 2(3) apply simp-all [3]
        apply (subst zcount-image-zmset)
        apply simp
        apply (subst card-eq-sum)
        apply (rule sum-pos[where y=u])
          apply simp-all
        done
     then have 0 < zcount (zmset-frontier (c-pts c loc3)) (results-in u s) + zcount
(union-frontiers c loc3) (results-in u s)
        by (auto intro: add-nonneg-pos)
      with riu-le-rit' have ?thesis
        apply (subst (asm) zcount-union[symmetric])
        apply (subst iiws)
        apply (erule obtain-elem-frontier)
        subgoal for u'
          by (auto intro!: exI[of - u'])
        done
    }
    moreover
    { — Same as induction base case
      assume 0 < zcount (c-imp c loc3) t
      then have ?thesis
        by (auto elim: obtain-elem-frontier)
    }
    moreover note disj
    ultimately show ?thesis
      by blast
  qed
qed

lemma loc-imps-fw-M-nonneg[simp]:
```

assumes *loc-imps-fw c loc1 loc2 M xs*
and *inv-implications-nonneg c*
shows $0 \leq$ *zcount M t*
using *assms*
by (*induct arbitrary*: *t rule*: *loc-imps-fw.induct*)
(*auto intro*!: *add-nonneg-nonneg sum-nonneg simp*: *zcount-image-zmset assms*(*2*)[*unfolded inv-implications-nonneg-def*])

lemma *loc-imps-fw-implication-in-M*:
assumes *inv-imps-work-sum c*
and *inv-implications-nonneg c*
and *loc-imps-fw c loc1 loc2 M xs*
and $0 <$ *zcount* (*c-imp c loc1*) *t*
shows $0 <$ *zcount M* (*results-in t* (*sum-path-weights xs*))
using *assms*(*3*,*4*)
proof (*induct rule*: *loc-imps-fw.induct*)
note *iiws* = *assms*(*1*)[*unfolded inv-imps-work-sum-def assms*(*3*), *simplified*, *rule-format*]
case (*1 loc*)
then show *?case*
by (*simp add*: *results-in-zero*)
next
case (*2 loc1 loc2 M xs s loc3*)
have $0 <$ *zcount M* (*results-in t* (*sum-path-weights xs*))
by (*rule 2*(*2*)[*OF 2*(*5*)])
then show *?case*
apply (*subst results-in-sum-path-weights-append*)
apply (*subst zcount-union*)
apply (*rule add-pos-nonneg*)
apply (*subst zcount-image-zmset*)
apply (*rule sum-pos*[**where** $y =$ *results-in t* (*sum-weights* (*map* (λ(*s*, *l*, *t*). *l*) *xs*))])
apply *simp*
apply (*auto simp*: *loc-imps-fw-M-nonneg*[*OF 2*(*1*) *assms*(*2*)] *zcount-inI*) [*3*]
apply (*auto simp*: *assms*(*2*)[*unfolded inv-implications-nonneg-def*])
done
qed

definition *impl-safe* :: (*'loc*, *'t*) *configuration* $\Rightarrow$ *bool* **where**
impl-safe c $\equiv \forall$ *loc1 loc2 t s*. *zcount* (*c-imp c loc1*) *t* $>$ *0* $\wedge$ *s* $\in_A$ *path-summary loc1 loc2*
$\longrightarrow$ ($\exists$ *t'*. *t'* $\in_A$ *frontier* (*c-imp c loc2*) $\wedge$ *t'* $\leq$ *results-in t s*)

lemma *impl-safe*:
assumes *inv-imps-work-sum c*
and *inv-implications-nonneg c*
and $\bigwedge$*loc*. *c-work c loc* = $\{\#\}_z$
shows *impl-safe c*
unfolding *impl-safe-def*
apply *clarify*

proof −
note *iiws* = *assms*(*1*)[*unfolded inv-imps-work-sum-def assms*(*3*), *simplified*, *rule-format*]
fix *loc1 loc2 t s*
assume *0* < *zcount* (*c-imp c loc1*) *t*
then obtain *t′* **where** *t′*: *t′* ∈$_A$ *frontier* (*c-imp c loc1*) *t′* ≤ *t*
by (*auto elim*: *obtain-elem-frontier*)
then have *t′-zcount*: *0* < *zcount* (*c-imp c loc1*) *t′*
by (*simp add*: *member-frontier-pos-zmset*)
assume *path-sum*: *s* ∈$_A$ *path-summary loc1 loc2*
obtain *M xs* **where** *Mxs*: *loc-imps-fw c loc1 loc2 M xs sum-path-weights xs* = *s*
by *atomize-elim* (*rule path-summary-conv-loc-imps-fw*[*OF path-sum*])
have *inM*: *0* < *zcount M* (*results-in t′* (*sum-path-weights xs*))
by (*rule loc-imps-fw-implication-in-M*[*OF assms*(*1,2*) *Mxs*(*1*) *t′-zcount*])
obtain *u* **where** *u*: *u* ≤ *results-in t′* (*sum-path-weights xs*) *u* ∈$_A$ *frontier* (*c-imp c loc2*)
by *atomize-elim* (*rule loc-imps-fw-M-in-implications*[*OF Mxs*(*1*) *assms*(*1,2,3*) *inM*])
then show ∃ *t′*. *t′* ∈$_A$ *frontier* (*c-imp c loc2*) ∧ *t′* ≤ *results-in t s*
apply (*intro exI*[*of* - *u*])
apply (*simp add*: *Mxs*(*2*))
using *t′*(*2*) **apply** (*meson order.trans results-in-mono*(*1*))
done
qed

— Safety for states where worklist is non-empty

lemma *cm-preserves-impl-safe*:
assumes *impl-safe c0*
and *next-change-multiplicity′ c0 c1 loc t n*
shows *impl-safe c1*
using *assms*
by (*auto simp*: *impl-safe-def next-change-multiplicity′-def*)

lemma *cm-preserves-safe*:
assumes *safe c0*
and *impl-safe c0*
and *next-change-multiplicity′ c0 c1 loc t n*
shows *safe c1*
proof −
from *assms*(*1*) **have** *safe*: *0* < *zcount* (*c-pts c0 loc1*) *t* ⟹ *s* ∈$_A$ *path-summary loc1 loc2*
⟹ ∃ *t′*≤*results-in t s*. *t′* ∈$_A$ *frontier* (*c-imp c0 loc2*) **for** *loc1 loc2 t s*
by (*auto simp*: *safe-def*)
from *assms*(*2*) **have** *impl-safe*: *0* < *zcount* (*c-imp c0 loc1*) *t* ⟹ *s* ∈$_A$ *path-summary loc1 loc2*
⟹ ∃ *t′*. *t′* ∈$_A$ *frontier* (*c-imp c0 loc2*) ∧ *t′* ≤ *results-in t s* **for** *loc1 loc2 t s*
by (*auto simp*: *impl-safe-def*)
from *assms*(*3*) **have** *imps*: *c-imp c1* = *c-imp c0*
unfolding *next-change-multiplicity′-def* **by** *auto*

```
  { fix loc1 loc2 u s — 'safe c1' variables
    assume u: 0 < zcount (c-pts c1 loc1) u
    then obtain u' where u': u' ∈_A frontier (c-pts c1 loc1) u' ≤ u
      using obtain-frontier-elem by blast
    assume path-sum: s ∈_A path-summary loc1 loc2
      — CM state changes:
    assume n-neq-zero: n ≠ 0
    assume impl: ∃ t'. t' ∈_A frontier (c-imp c0 loc) ∧ t' ≤ t
    assume pointstamps:
      ∀ loc'. c-pts c1 loc' =
                  (if loc' = loc then update-zmultiset (c-pts c0 loc') t n
                                 else c-pts c0 loc')
    have ∃ t'≤results-in u s. t' ∈_A frontier (c-imp c1 loc2)
    proof (cases n < 0)
      case True — Trivial, because no new pointstamp could have appeared
      with u have u-c0: 0 < zcount (c-pts c0 loc1) u
        unfolding pointstamps[rule-format]
        by (cases loc1=loc; cases t=u) (auto simp: zcount-update-zmultiset)
      show ?thesis
        unfolding imps
        by (rule safe[OF u-c0 path-sum])
    next
      case False
      with n-neq-zero have n > 0
        by linarith
      then show ?thesis
        unfolding imps
        apply (cases loc=loc1; cases t=u)
        using impl
          apply (elim exE conjE)
        subgoal for t'
          apply simp
          apply (drule member-frontier-pos-zmset)
          apply (drule impl-safe[rotated, OF path-sum])
          apply (elim exE)
          subgoal for t''
            apply (rule exI[of - t''])
            using results-in-mono(1) order-trans apply blast
            done
          done
     using u path-sum apply (auto simp: zcount-update-zmultiset pointstamps[rule-format]
intro: safe)
        done
    qed
  }
  note r = this
  show ?thesis
    unfolding safe-def
    apply clarify
```

subgoal for *loc1 loc2 t s*
using *assms(3)*[*unfolded next-change-multiplicity'-def*]
by (*intro r*[*of loc1 t s loc2*]) *auto*
done
qed

7.5 A Better (More Invariant) Safety

definition *worklists-vacant-to* :: ('*loc*, '*t*) *configuration* ⇒ '*t* ⇒ *bool* **where**
worklists-vacant-to c t =
(∀ *loc1 loc2 s t'*. *s* ∈$_A$ *path-summary loc1 loc2* ∧ *t'* ∈#$_z$ *c-work c loc1* ⟶ ¬ *results-in t' s* ≤ *t*)

definition *inv-safe* :: ('*loc*, '*t*) *configuration* ⇒ *bool* **where**
inv-safe c = (∀ *loc1 loc2 t s*. *0* < *zcount* (*c-pts c loc1*) *t*
∧ *s* ∈$_A$ *path-summary loc1 loc2*
∧ *worklists-vacant-to c* (*results-in t s*)
⟶ (∃ *t'* ≤ *results-in t s*. *t'* ∈$_A$ *frontier* (*c-imp c loc2*)))

Intuition: Unlike safe, *inv-safe* is an invariant because it only claims the safety property *t'* ∈$_A$ *frontier* (*c-imp c loc2*) for pointstamps that can't be modified by future propagated updates anymore (i.e. there are no upstream worklist entries which can result in a less or equal pointstamp).

lemma *in-frontier-diff*: ∀ *y*∈#$_z$*N*. ¬ *y* ≤ *x* ⟹ *x* ∈$_A$ *frontier* (*M* − *N*) ⟷ *x* ∈$_A$ *frontier M*
apply *transfer'*
unfolding *in-minimal-antichain*
apply (*metis* (*mono-tags*, *lifting*) *diff-zero le-less mem-Collect-eq set-zmset-def zcount-diff*)
done

lemma *worklists-vacant-to-trans*:
worklists-vacant-to c t ⟹ *t'* ≤ *t* ⟹ *worklists-vacant-to c t'*
unfolding *worklists-vacant-to-def*
using *order.trans* **by** *blast*

lemma *loc-imps-fw-M-in-implications'*:
assumes *loc-imps-fw c loc1 loc2 M xs*
and *inv-imps-work-sum c*
and *inv-implications-nonneg c*
and *worklists-vacant-to c t*
and *0* < *zcount M t*
shows ∃ *s*≤*t*. *s* ∈$_A$ *frontier* (*c-imp c loc2*)
using *assms(1,4,5)*
proof (*induct arbitrary*: *t rule*: *loc-imps-fw.induct*)
note *iiws* = *assms(2)*[*unfolded inv-imps-work-sum-def*, *rule-format*]
case (*1 loc t*)
then show *?case*
by (*auto elim*: *obtain-elem-frontier*)

next
note *iiws* = *assms*(*2*)[*unfolded inv-imps-work-sum-def eq-diff-eq*[*symmetric*], *rule-format*]
case (*2 loc1 loc2 M xs s loc3 t*)
from *2*(*6*) **consider** *0* < *zcount* {# *results-in t s*. *t* ∈#$_z$ *M* #} *t* | *0* < *zcount* (*c-imp c loc3*) *t*
by *atomize-elim auto*
then show *?case*
proof *cases*
case *1*
then obtain *t′* **where** *t′*: *results-in t′ s* = *t 0* < *zcount M t′*
apply *atomize-elim*
apply (*rule ccontr*)
apply (*subst* (*asm*) *zcount-image-zmset*)
apply (*clarsimp simp*: *vimage-def*)
apply (*metis* (*mono-tags*, *lifting*) *Int-iff mem-Collect-eq sum-pos-ex-elem-pos*)
done
have *vacant-to*: *worklists-vacant-to c t′*
apply (*rule worklists-vacant-to-trans*)
apply (*rule 2*(*5*))
using *zero-le results-in-mono*(*2*) *results-in-zero t′*(*1*) **apply** *fastforce*
done
obtain *u* **where** *u*: *u* ≤ *t′ u* ∈$_A$ *frontier* (*c-imp c loc2*)
using *2*(*2*)[*OF vacant-to t′*(*2*)] **by** *fast*
then have *riu-le-rit′*: *results-in u s* ≤ *t*
by (*simp add*: *t′*(*1*)[*symmetric*] *results-in-mono*)
from *u* **have** *0* < *zcount* (*union-frontiers c loc3*) (*results-in u s*)
apply (*subst zcount-sum*)
apply (*rule sum-pos*[**where** *y*=*loc2*])
apply *simp-all* [*3*]
apply (*clarsimp simp*: *after-summary-def*)
apply (*subst zcount-sum*)
apply (*rule sum-pos*[**where** *y*=*s*])
using *2*(*3*) **apply** *simp-all* [*3*]
apply (*subst zcount-image-zmset*)
apply *simp*
apply (*subst card-eq-sum*)
apply (*rule sum-pos*[**where** *y*=*u*])
apply *simp-all*
done
then have *0* < *zcount* (*zmset-frontier* (*c-pts c loc3*)) (*results-in u s*) + *zcount* (*union-frontiers c loc3*) (*results-in u s*)
by (*auto intro*: *add-nonneg-pos*)
with *riu-le-rit′* **show** *?thesis*
apply (*subst* (*asm*) *zcount-union*[*symmetric*])
apply (*subst iiws*)
apply (*erule obtain-elem-frontier*)
subgoal for *u′*
apply (*rule exI*[*of* - *u′*])
apply (*subst in-frontier-diff*)

```
        apply clarify
       subgoal for t'
         using 2(5)[unfolded worklists-vacant-to-def, rule-format, of 0 loc3 loc3 t']
         apply −
         apply (drule meta-mp)
          apply (intro conjI)
           apply simp
          apply simp
         apply (metis order-trans results-in-zero)
         done
       apply auto
       done
     done
  next
    case 2
    then show ?thesis
      by (auto elim: obtain-elem-frontier)
  qed
qed

lemma inv-safe:
  assumes inv-imps-work-sum c
    and   inv-implications-nonneg c
  shows   inv-safe c
  unfolding inv-safe-def
  apply clarify
proof −
  note iiws = assms(1)[unfolded inv-imps-work-sum-def, rule-format]
  fix loc1 loc2 t s
  assume vacant: worklists-vacant-to c (results-in t s)
  assume 0 < zcount (c-pts c loc1) t
  then obtain t' where t': t' ∈A frontier (c-pts c loc1) t' ≤ t
    using obtain-frontier-elem by blast
  have zcount-wl: zcount (c-work c loc1) t' = 0
     using vacant[unfolded worklists-vacant-to-def, rule-format, of 0 loc1 loc1 t',
simplified]
   by (metis add.left-neutral order.trans le-plus(1) results-in-mono(2) results-in-zero
t'(2) zcount-ne-zero-iff)
  obtain t'' where t'': t'' ∈A frontier (c-imp c loc1) t'' ≤ t'
  proof atomize-elim
      from t'(1) have 0 < zcount (zmset-frontier (c-pts c loc1)) t' + zcount
(union-frontiers c loc1) t'
      by (auto intro: add-pos-nonneg simp: union-frontiers-nonneg)
    then show ∃ t''. t'' ∈A frontier (c-imp c loc1) ∧ t'' ≤ t'
      apply (subst (asm) zcount-union[symmetric])
      apply (subst (asm) iiws[symmetric])
      using zcount-wl
      apply (auto elim: obtain-frontier-elem)
      done
```

qed
then have *t''-zcount*: *0 < zcount (c-imp c loc1) t''*
by (*simp add*: *member-frontier-pos-zmset*)
assume *path-sum*: *s* $\in_A$ *path-summary loc1 loc2*
obtain *M xs* **where** *Mxs*: *loc-imps-fw c loc1 loc2 M xs sum-path-weights xs = s*
by *atomize-elim* (*rule path-summary-conv-loc-imps-fw*[*OF path-sum*])
have *inM*: *0 < zcount M* (*results-in t''* (*sum-path-weights xs*))
by (*rule loc-imps-fw-implication-in-M*[*OF assms*(*1,2*) *Mxs*(*1*) *t''-zcount*])
have *vacant2*: *worklists-vacant-to c* (*results-in t''* (*sum-weights* (*map* (λ(*s*, *l*, *t*). *l*) *xs*)))
apply (*subst Mxs*(*2*))
apply (*meson results-in-mono*(*1*) *worklists-vacant-to-trans t''*(*2*) *t'*(*2*) *vacant*)
done
obtain *u* **where** *u*: *u* ≤ *results-in t''* (*sum-path-weights xs*) *u* $\in_A$ *frontier* (*c-imp c loc2*)
by *atomize-elim* (*rule loc-imps-fw-M-in-implications'*[*OF Mxs*(*1*) *assms*(*1,2*) *vacant2 inM*])
then show ∃ *t'*≤*results-in t s. t'* $\in_A$ *frontier* (*c-imp c loc2*)
apply (*intro exI*[*of - u*])
apply (*simp add*: *Mxs*(*2*))
using *t''*(*2*) *t'*(*2*) **apply** (*meson order.trans results-in-mono*(*1*))
done
qed

lemma *alw-conjI*: *alw P s* ⟹ *alw Q s* ⟹ *alw* (λ*s*. *P s* ∧ *Q s*) *s*
by (*coinduction arbitrary*: *s*) *auto*

lemma *alw-inv-safe*: *spec s* ⟹ *alw* (*holds inv-safe*) *s*
apply (*frule spec-imp-iiws*)
apply (*drule alw-inv-implications-nonneg*)
apply (*rule alw-mp*[**where** φ = λ*s*. *holds inv-imps-work-sum s* ∧ *holds inv-implications-nonneg s*])
apply (*rule alw-conjI*)
apply *assumption+*
apply (*simp add*: *alw-mono inv-safe*)
done

lemma *empty-worklists-vacant-to*: ∀ *loc*. *c-work c loc* = {#}$_z$ ⟹ *worklists-vacant-to c t*
unfolding *worklists-vacant-to-def*
by *auto*

lemma *inv-safe-safe*: (⋀*loc*. *c-work c loc* = {#}$_z$) ⟹ *inv-safe c* ⟹ *safe c*
unfolding *inv-safe-def safe-def worklists-vacant-to-def* **by** *auto*

lemma *safe*:
assumes *inv-imps-work-sum c*
and *inv-implications-nonneg c*
and ⋀*loc*. *c-work c loc* = {#}$_z$

shows *safe c*
by (*rule inv-safe-safe*[*OF assms*(*3*) *inv-safe*[*OF assms*(*1,2*)]])

7.6 Implied Frontier

abbreviation *zmset-pos* **where** *zmset-pos M* ≡ *zmset-of* (*mset-pos M*)

definition *implied-frontier* **where**
implied-frontier P loc = *frontier* (∑ *loc'*∈*UNIV*. *after-summary* (*zmset-pos* (*P loc'*)) (*path-summary loc' loc*))

definition *implied-frontier-alt* **where**
implied-frontier-alt c loc = *frontier* (∑ *loc'*∈*UNIV*. *after-summary* (*zmset-frontier* (*c-pts c loc'*)) (*path-summary loc' loc*))

lemma *in-frontier-least*: *x* ∈$_A$ *frontier M* ⟹ ∀ *y*. *0* < *zcount M y* ⟶ ¬ *y* < *x*
by *transfer'* (*auto simp*: *minimal-antichain-def*)

lemma *in-frontier-trans*: *0* < *zcount M y* ⟹ *x* ∈$_A$ *frontier M* ⟹ *y* ≤ *x* ⟹ *y* ∈$_A$ *frontier M*
by *transfer* (*simp add*: *le-less minimal-antichain-def*)

lemma *implied-frontier-alt-least*:
assumes *b* ∈$_A$ *implied-frontier-alt c loc2*
shows ∀ *loc a' s'*. *a'* ∈$_A$ *frontier* (*c-pts c loc*) ⟶ *s'* ∈$_A$ *path-summary loc loc2* ⟶ ¬ *results-in a' s'* < *b*
proof (*intro allI impI notI*)
fix *loc a' s'*
assume *a'*: *a'* ∈$_A$ *frontier* (*c-pts c loc*)
assume *s'*: *s'* ∈$_A$ *path-summary loc loc2*
assume *lt*: *results-in a' s'* < *b*
have *0* < *zcount* (*after-summary* (*zmset-frontier* (*c-pts c loc*)) (*path-summary loc loc2*)) (*results-in a' s'*)
using *a' s'* **by** (*auto intro*!: *pos-zcount-after-summary*)
then have *0* < *zcount* (∑ *loc'*∈*UNIV*. *after-summary* (*zmset-frontier* (*c-pts c loc'*)) (*path-summary loc' loc2*)) (*results-in a' s'*)
by (*auto intro*!: *sum-pos*[**where** *y* = *loc*] *simp*: *zcount-sum*)
then have *results-in a' s'* ∈$_A$ *implied-frontier-alt c loc2*
using *assms lt* **unfolding** *implied-frontier-alt-def*
by (*auto dest*!: *in-frontier-trans*[**where** *y* = *results-in a' s'* **and** *x* = *b*])
with *lt assms* **show** *False*
unfolding *implied-frontier-alt-def*
using *frontier-comparable-False*
by *blast*
qed

lemma *implied-frontier-alt-in-pointstamps*:
assumes *b* ∈$_A$ *implied-frontier-alt c loc2*
obtains *a s loc1* **where**

$a \in_A$ *frontier* (*c-pts c loc1*) $s \in_A$ *path-summary loc1 loc2 results-in a s* = *b*
using *assms* **apply** *atomize-elim*
unfolding *implied-frontier-alt-def*
apply (*drule member-frontier-pos-zmset*)
apply (*subst* (*asm*) *zcount-sum*)
apply (*drule sum-pos-ex-elem-pos*)
apply *clarify*
apply (*rule after-summary-obtain-pre*[*rotated*])
apply *simp*
subgoal for *loc1 a s*
by (*auto intro*!: *exI*[*of - loc1*] *exI*[*of - a*] *exI*[*of - s*])
apply *simp*
done

lemma *in-implied-frontier-alt-in-implication-frontier*:
assumes *inv-imps-work-sum c*
and *inv-implications-nonneg c*
and *worklists-vacant-to c b*
and *b* $\in_A$ *implied-frontier-alt c loc2*
shows *b* $\in_A$ *frontier* (*c-imp c loc2*)
proof −
have *safe*: *0* < *zcount* (*c-pts c loc1*) *t* $\Longrightarrow$ *s* $\in_A$ *path-summary loc1 loc2* $\Longrightarrow$ *results-in t s* ≤ *b*
$\Longrightarrow \exists t' \leq$*results-in t s*. *t'* $\in_A$ *frontier* (*c-imp c loc2*) **for** *loc1 loc2 t s*
by (*rule inv-safe*[*OF assms*(*1,2*), *unfolded inv-safe-def*, *rule-format*])
(*auto elim*: *worklists-vacant-to-trans*[*OF assms*(*3*)])
— Pointstamp *b* in the *implied-frontier-alt* is caused by a pointstamp *a* and summary *s* and *results-in a s* is least among such pointstamps
from *assms*(*4*) **obtain** *loc1 a s* **where** *loc1-a-s*:
a $\in_A$ *frontier* (*c-pts c loc1*) *s* $\in_A$ *path-summary loc1 loc2 results-in a s* = *b*
$\forall$ *loc a' s'*. *a'* $\in_A$ *frontier* (*c-pts c loc*) $\longrightarrow$ *s'* $\in_A$ *path-summary loc loc2* $\longrightarrow \neg$ *results-in a' s'* < *b*
apply *atomize-elim*
apply (*rule implied-frontier-alt-in-pointstamps*)
apply *simp*
apply (*drule implied-frontier-alt-least*)
apply *fast*
done
then have *zcount-ps*: *0* < *zcount* (*c-pts c loc1*) *a*
using *member-frontier-pos-zmset* **by** *blast*
— From ‘safe‘ we know that pointstamp *a* is reflected in the implications by some poinstamp *b'* ≤ *b*
obtain *b'* **where** *b'*: *b'* $\in_A$ *frontier* (*c-imp c loc2*) *b'* ≤ *results-in a s*
using *safe*[*OF zcount-ps loc1-a-s*(*2*)] *loc1-a-s*(*3*) **by** *blast*
have *b'* = *results-in a s*
proof −
have *zcount* (*c-work c loc*) *t* = *0* **if** *results-in t s* ≤ *b'* **for** *t s loc*
proof −
have *results-in t s* ≤ *b*

using *b'(2) loc1-a-s(3) that* **by** *force*
then show *?thesis*
by (*meson assms(3) results-in-mono(2) worklists-vacant-to-def flow.zero-le order-trans*
path-weight-refl zcount-inI)
qed
— but the pointstamp can't be strictly less, because we know that *results-in a s* is least
then obtain *a' loc1' s'* **where** *a'*:
s' ∈A path-summary loc1' loc2 results-in a' s' ≤ b' a' ∈A frontier (*c-pts c loc1'*)
using *implication-implies-pointstamp*[*OF b'(1) assms(1), simplified*] **by** *force*
{ assume *b' ≠ results-in a s*
with *b'(2)* **have** *b'-lt*: *b' < results-in a s*
by *simp*
note *loc1-a-s(4)*[*rule-format, unfolded loc1-a-s(3)*[*symmetric*], *OF a'(3,1)*]
with *b'-lt a'(2)* **have** *False*
by (*simp add*: *leD less-le order-trans*)
}
then show *?thesis*
by (*rule ccontr*)
qed
— Hence, the *implied-frontier-alt* pointstamp *b* is reflected in the implications
with *b'* **show** *b ∈A frontier* (*c-imp c loc2*)
by (*auto simp*: *loc1-a-s(3)*)
qed

lemma *in-implication-frontier-in-implied-frontier-alt*:
assumes *inv-imps-work-sum c*
and *inv-implications-nonneg c*
and *worklists-vacant-to c b*
and *b ∈A frontier* (*c-imp c loc2*)
shows *b ∈A implied-frontier-alt c loc2*
proof −
have *safe*: *0 < zcount* (*c-pts c loc1*) *t ⟹ s ∈A path-summary loc1 loc2 ⟹ results-in t s ≤ b*
⟹ ∃ t'≤results-in t s. t' ∈A frontier (*c-imp c loc2*) **for** *loc1 loc2 t s*
by (*rule inv-safe*[*OF assms(1,2), unfolded inv-safe-def, rule-format*])
(*auto elim*: *worklists-vacant-to-trans*[*OF assms(3)*])
have *zcount* (*c-work c loc*) *t = 0* **if** *results-in t s ≤ b* **for** *t s loc*
using *that* **by** (*meson assms(3) results-in-mono(2) worklists-vacant-to-def flow.zero-le*
order-trans path-weight-refl zcount-inI)
— Pointstamp *b* in the implications is caused by a pointstamp *a* and a summary *s*
then obtain *loc1 a s* **where** *loc1-a-s*:
s ∈A path-summary loc1 loc2 results-in a s ≤ b a ∈A frontier (*c-pts c loc1*)
using *implication-implies-pointstamp*[*OF assms(4) assms(1), simplified*] **by** *force*
then have *zcount-a*: *0 < zcount* (*c-pts c loc1*) *a*

```
    using member-frontier-pos-zmset by blast
  have b-ria: results-in a s = b
  proof (rule ccontr)
    assume results-in a s ≠ b
    with loc1-a-s(2) have results-in a s < b
      by simp
    then show False
      using safe[OF zcount-a loc1-a-s(1)] assms(4) loc1-a-s(2)
      using order.strict-trans1 frontier-comparable-False by blast
  qed
    — results-in a s is a candidate for inclusion in the implied-frontier-alt..
  have 0 < zcount (∑ loc'∈UNIV. after-summary (zmset-frontier (c-pts c loc'))
(path-summary loc' loc2)) (results-in a s)
    unfolding after-summary-def
    apply (subst zcount-sum)
    apply (rule sum-pos[of - - loc1])
      apply simp
     apply (clarsimp simp: zcount-sum)
     apply (rule sum-nonneg)
     apply (subst zcount-image-zmset)
     apply auto [2]
    apply (subst zcount-sum)
    apply (rule sum-pos[of - - s])
    using loc1-a-s(1) apply simp-all [3]
    apply (subst zcount-image-zmset)
    apply (rule sum-pos[of - - a])
    using loc1-a-s(3) apply auto
    done
      — ..which means a pointstamp b' ≤ results-in a s must exist in the im-
plied-frontier-alt..
  then obtain b' where b': b' ∈A implied-frontier-alt c loc2 b' ≤ results-in a s
    by (auto simp: implied-frontier-alt-def elim: obtain-frontier-elem)
  then have worklists-vacant-to c b'
    using loc1-a-s(2) by (auto intro: worklists-vacant-to-trans[OF assms(3)])
  with b' have b'-frontier: b' ∈A frontier (c-imp c loc2)
    using in-implied-frontier-alt-in-implication-frontier assms by blast
    — ..and this pointstamp must be equal to b'
  have b'-ria: b' = results-in a s
  proof (rule ccontr)
    assume b' ≠ results-in a s
    with b'(2) have b'-lt: b' < results-in a s
      by simp
    from b'-frontier b'-lt b-ria assms(4) show False
      using frontier-comparable-False by blast
  qed
  — Hence, the implication frontier pointstamp b is reflected in the implied-frontier-alt
  from b' b'-ria b-ria show b ∈A implied-frontier-alt c loc2
    by (auto simp: implied-frontier-alt-def)
qed
```

lemma *implication-frontier-iff-implied-frontier-alt-vacant*:
assumes *inv-imps-work-sum c*
and *inv-implications-nonneg c*
and *worklists-vacant-to c b*
shows $b \in_A$ *frontier* (*c-imp c loc*) $\longleftrightarrow b \in_A$ *implied-frontier-alt c loc*
using
ac-eq-iff
in-implication-frontier-in-implied-frontier-alt[*OF assms*]
in-implied-frontier-alt-in-implication-frontier[*OF assms*]
by *blast*

lemma *next-propagate-implied-frontier-alt-def*:
next-propagate c c' $\Longrightarrow$ *implied-frontier-alt c loc* = *implied-frontier-alt c' loc*
by (*auto simp*: *next-propagate'-def implied-frontier-alt-def*)

lemma *implication-frontier-eq-implied-frontier-alt*:
assumes *inv-imps-work-sum c*
and *inv-implications-nonneg c*
and $\bigwedge$*loc. c-work c loc* = $\{\#\}_z$
shows *frontier* (*c-imp c loc*) = *implied-frontier-alt c loc*
using *ac-eq-iff implication-frontier-iff-implied-frontier-alt-vacant empty-worklists-vacant-to assms*
by *blast*

lemma *alw-implication-frontier-eq-implied-frontier-alt-empty*: *spec s* $\Longrightarrow$
alw (*holds* (λ*c*. ($\forall$ *loc*. *c-work c loc* = $\{\#\}_z$) $\longrightarrow$ *frontier* (*c-imp c loc*) = *implied-frontier-alt c loc*)) *s*
apply (*frule spec-imp-iiws*)
apply (*drule alw-inv-implications-nonneg*)
apply (*drule* (*1*) *alw-conjI*)
apply (*erule thin-rl*)
apply (*auto elim*!: *alw-mono simp*: *implication-frontier-eq-implied-frontier-alt*)
done

lemma *alw-implication-frontier-eq-implied-frontier-alt-vacant*: *spec s* $\Longrightarrow$
alw (*holds* (λ*c*. *worklists-vacant-to c b* $\longrightarrow b \in_A$ *frontier* (*c-imp c loc*) $\longleftrightarrow b \in_A$ *implied-frontier-alt c loc*)) *s*
apply (*frule spec-imp-iiws*)
apply (*drule alw-inv-implications-nonneg*)
apply (*drule* (*1*) *alw-conjI*)
apply (*erule thin-rl*)
apply (*auto elim*!: *alw-mono simp*: *implication-frontier-iff-implied-frontier-alt-vacant*)
done

lemma *antichain-eqI*: ($\bigwedge$*b*. $b \in_A A \longleftrightarrow b \in_A B$) $\Longrightarrow A = B$
by *transfer auto*

lemma *zmset-frontier-zmset-pos*: *zmset-frontier A* $\subseteq\#_z$ *zmset-pos A*

unfolding *subseteq-zmset-def*
by *transfer'* (*auto simp*: *count-mset-set-if minimal-antichain-def*)

lemma *image-mset-mono-pos*:
$\forall\, b.\ 0 \leq$ *zcount A b* $\Longrightarrow \forall\, b.\ 0 \leq$ *zcount B b* $\Longrightarrow$ *A* $\subseteq\#_z$ *B* $\Longrightarrow$ *image-zmset f A* $\subseteq\#_z$ *image-zmset f B*
unfolding *subseteq-zmset-def zcount-image-zmset*
apply (*intro allI*)
apply (*rule order-trans*[*OF sum-mono sum-mono2*])
apply *simp-all*
by (*metis Int-subset-iff antisym subsetI zcount-ne-zero-iff*)

lemma *sum-mono-subseteq*:
$(\bigwedge i.\ i \in K \Longrightarrow f\ i \subseteq\#_z\ g\ i) \Longrightarrow (\sum i \in K.\ f\ i) \subseteq\#_z (\sum i \in K.\ g\ i)$
by (*simp add*: *subseteq-zmset-def sum-mono zcount-sum*)

lemma *after-summary-zmset-frontier*:
after-summary (*zmset-frontier A*) *S* $\subseteq\#_z$ *after-summary* (*zmset-pos A*) *S*
unfolding *after-summary-def*
apply (*rule sum-mono-subseteq*)
apply (*rule image-mset-mono-pos*[*OF - - zmset-frontier-zmset-pos*])
apply *auto*
done

lemma *frontier-eqI*: $\forall\, b.\ 0 \leq$ *zcount A b* $\Longrightarrow \forall\, b.\ 0 \leq$ *zcount B b* $\Longrightarrow$
A $\subseteq\#_z$ *B* $\Longrightarrow (\bigwedge b.\ b \in\#_z B \Longrightarrow \exists\, a.\ a \in\#_z A \wedge a \leq b) \Longrightarrow$ *frontier A* = *frontier B*
apply (*transfer fixing*: *A B*)
apply (*clarsimp simp*: *minimal-antichain-def subseteq-zmset-def*)
apply *safe*
apply (*metis less-le-trans*)
apply (*metis less-le less-le-trans zcount-ne-zero-iff*)
apply (*metis antisym less-le zcount-ne-zero-iff*)
apply (*meson less-le-trans*)
done

lemma *implied-frontier-implied-frontier-alt*: *implied-frontier* (*c-pts c*) *loc* = *implied-frontier-alt c loc*
unfolding *implied-frontier-alt-def implied-frontier-def*
apply (*rule frontier-eqI*[*symmetric*])
subgoal by (*auto simp*: *zcount-sum sum-nonneg*)
subgoal by (*auto simp*: *zcount-sum sum-nonneg*)
subgoal by (*rule sum-mono-subseteq*[*OF after-summary-zmset-frontier*])
apply (*simp flip*: *zcount-ne-zero-iff add*: *zcount-sum*)
apply (*erule sum.not-neutral-contains-not-neutral*)
apply (*simp flip*: *zcount-ne-zero-iff add*: *after-summary-def zcount-sum*)
apply (*erule sum.not-neutral-contains-not-neutral*)
apply (*simp flip*: *zcount-ne-zero-iff add*: *zcount-image-zmset split*: *if-splits*)
apply (*erule sum.not-neutral-contains-not-neutral*)

apply (*simp flip*: *zcount-ne-zero-iff*)
subgoal for *u loc s t*
apply (*rule obtain-elem-frontier*[*of c-pts c loc t*])
apply (*metis le-less*)
subgoal for *a*
apply (*rule exI*[*of - results-in a s*])
apply (*rule conjI*[*rotated*])
using *results-in-mono*(*1*) **apply** *blast*
apply (*subst sum-nonneg-eq-0-iff*; *simp add*: *sum-nonneg*)
apply (*rule exI*[*of - loc*])
apply (*subst sum-nonneg-eq-0-iff*; *simp*)
apply (*rule bexI*[*of - s*])
apply *auto*
done
done
done

lemmas *alw-implication-frontier-eq-implied-frontier-vacant* =
alw-implication-frontier-eq-implied-frontier-alt-vacant[*folded implied-frontier-implied-frontier-alt*]
lemmas *implication-frontier-iff-implied-frontier-vacant* =
implication-frontier-iff-implied-frontier-alt-vacant[*folded implied-frontier-implied-frontier-alt*]

end

8 Combined Progress Tracking Protocol

lemma *fold-invar*:
assumes *finite M*
and *P z*
and $\forall z.\ \forall x{\in}M.\ P\ z \longrightarrow P\ (f\ x\ z)$
and *comp-fun-commute f*
shows *P* (*Finite-Set.fold f z M*)
proof −
interpret *commute*: *comp-fun-commute f*
by (*fact* ‹*comp-fun-commute f*›)
from *assms* **show** *?thesis*
by (*induct M arbitrary*: *z rule*: *finite-induct*) *auto*
qed

8.1 Could-result-in Relation

context *dataflow-topology* **begin**

definition *cri-less-eq* :: (*'loc* × *'t*) ⇒ (*'loc* × *'t*) ⇒ *bool* (-$\leq_p$- [*51,51*] *50*) **where**
cri-less-eq =
(λ(*loc1,t1*) (*loc2,t2*). (∃ *s*. *s* ∈$_A$ *path-summary loc1 loc2* ∧ *results-in t1 s* ≤ *t2*))

definition *cri-less* :: ('*loc* × '*t*) ⇒ ('*loc* × '*t*) ⇒ *bool* (-≺$_p$- [*51*,*51*] *50*) **where**
cri-less x y = (*x* ≤$_p$ *y* ∧ *x* ≠ *y*)

lemma *cri-asym1*: *x* ≺$_p$ *y* ⟶ ¬ *y* ≺$_p$ *x*
for *x y* **apply** (*cases x*; *cases y*)
proof (*rule ccontr*)
fix *loc1 t1 loc2 t2*
assume *as*: ¬ (*x* ≺$_p$ *y* ⟶ ¬ *y* ≺$_p$ *x*) *x* = (*loc1*, *t1*) *y* = (*loc2*, *t2*)
then have *as1*: *loc1* ≠ *loc2*
unfolding *cri-less-def cri-less-eq-def*
by *clarsimp*
(*metis add.right-neutral order.antisym order.trans le-plus*(*2*) *results-in-mono*(*2*) *results-in-zero*)
from *as* **obtain** *s1* **where** *s1*: *s1* ∈$_A$ *path-summary loc1 loc2*
results-in t1 s1 ≤ *t2*
using *cri-less-def cri-less-eq-def* **by** *auto*
then obtain *path1* **where** *path1*: *flow.path loc1 loc2 path1*
s1 = *flow.sum-path-weights path1*
path1 ≠ []
using *as1 flow.path-weight-conv-path flow.path0E* **by** *blast*
from *as* **obtain** *s2* **where** *s2*: *s2* ∈$_A$ *path-summary loc2 loc1*
results-in t2 s2 ≤ *t1*
using *cri-less-def cri-less-eq-def* **by** *auto*
then obtain *path2* **where** *path2*: *flow.path loc2 loc1 path2*
s2 = *flow.sum-path-weights path2*
path2 ≠ []
using *as1 flow.path-weight-conv-path flow.path0E* **by** *blast*
with *path1* **have** *path-total*: *flow.path loc1 loc1* (*path1*@*path2*)
using *flow.path-trans path2*(*3*) **by** *blast*
then obtain *s3* **where** *s3*: *s3* = *flow.sum-path-weights* (*path1*@*path2*) **by** *auto*
then have *s3-sum*: *s3* = *followed-by s1 s2*
using *path1 path2 flow.sum-weights-append* **by** *auto*
have ¬ *t1* < *results-in t1 s3*
using *s3-sum s1*(*2*) *s2*(*2*) *followed-by-summary*
by (*metis le-less-trans less-le-not-le results-in-mono*(*1*))
then show *False* **using** *path-total no-zero-cycle s3 path1*(*3*) *path2*(*3*) **by** *blast*
qed

lemma *cri-asym2*: *x* ≺$_p$ *y* ⟶ *x* ≠ *y*
by (*simp add*: *cri-less-def*)

sublocale *cri*: *order cri-less-eq cri-less*
apply *standard*
subgoal
using *cri-asym1 cri-asym2 cri-less-def* **by** *blast*
subgoal for *x*
unfolding *cri-less-eq-def*
using *flow.path-weight-refl results-in-zero* **by** *fastforce*
subgoal

for *x y z* **apply** (*cases x*; *cases y*; *cases z*)
proof −
fix *a b aa ba ab bb*
assume *as*: $x \leq_p y$ $y \leq_p z$ $x = (a, b)$ $y = (aa, ba)$ $z = (ab, bb)$
then obtain *s1* **where** *s1*: $s1 \in_A$ *path-summary a aa results-in b s1* $\leq$ *ba*
using *cri-less-eq-def* **by** *auto*
from *as*(*2*,*4*,*5*) **obtain** *s2* **where** *s2*: $s2 \in_A$ *path-summary aa ab results-in ba s2* $\leq$ *bb*
using *cri-less-eq-def* **by** *auto*
with *s1* **obtain** *s3* **where** *s3*: $s3 \in_A$ *path-summary a ab s3* $\leq$ *followed-by s1 s2*
using *flow.path-weight-elem-trans* **by** *blast*
with *s1 s2* **have** *results-in b s3* $\leq$ *bb*
proof −
have $\forall$ *s. results-in* (*results-in b s1*) *s* $\leq$ *results-in ba s*
by (*meson results-in-mono*(*1*) *s1*(*2*))
then show *?thesis*
by (*metis* (*no-types*) *results-in-mono*(*2*) *followed-by-summary order-trans s2*(*2*) *s3*(*2*))
qed
with *as s3* **show** *?thesis* **unfolding** *cri-less-eq-def* **by** *blast*
qed
using *cri-asym1 cri-asym2 cri-less-def* **by** *blast*

lemma *wf-cri*: *wf* {(*l*, *l'*). (*l*, *t*) $<_p$ (*l'*, *t*)}
by (*rule finite-acyclic-wf*)
(*auto intro*: *cri.acyclicI-order*[*THEN acyclic-converse*[*THEN iffD1*]])

end

8.2 Specification

8.2.1 Configuration

record ('*p*::*finite*, '*t*::*order*, '*loc*) *configuration* =
exchange-config :: ('*p*, ('*loc* × '*t*)) *Exchange.configuration*
prop-config :: '*p* ⇒ ('*loc*, '*t*) *Propagate.configuration*
init :: '*p* ⇒ *bool*

type-synonym ('*p*, '*t*, '*loc*) *computation* = ('*p*, '*t*, '*loc*) *configuration stream*

context *dataflow-topology* **begin**

definition *the-cm* **where**
the-cm c loc t n = (*THE c'*. *next-change-multiplicity' c c' loc t n*)

the-cm is not commutative in general, only if the necessary conditions hold. It can be converted to *apply-cm* for which we prove *comp-fun-commute*.

definition *apply-cm* **where**
apply-cm c loc t n =
(*let new-pointstamps* = (λ*loc'*.

```
            (if loc' = loc then update-zmultiset (c-pts c loc') t n
                           else c-pts c loc')) in
      c (| c-pts := new-pointstamps |)
        (| c-work :=
           (λloc'. c-work c loc' + frontier-changes (new-pointstamps loc') (c-pts c
loc'))|))
```

definition *cm-all'* **where**
cm-all' c0 Δ =
Finite-Set.fold (λ(loc, t) c. apply-cm c loc t (zcount Δ (loc,t))) c0 (set-zmset Δ)

definition *cm-all* **where**
cm-all c0 Δ =
Finite-Set.fold (λ(loc, t) c. the-cm c loc t (zcount Δ (loc,t))) c0 (set-zmset Δ)

definition *propagate-all c0 = while-option (λc. ∃ loc. (c-work c loc) ≠ {#}$_z$)*
(λc. SOME c'. ∃ loc t. next-propagate' c c' loc t)
c0

8.2.2 Initial state and state transitions

definition *InitConfig* :: (*'p*::*finite*, *'t*::*order*, *'loc*) *configuration* ⇒ *bool* **where**
InitConfig c =
((∀ p. init c p = False)
∧ cri.init-config (exchange-config c)
∧ (∀ p loc t. zcount (c-pts (prop-config c p) loc) t
= zcount (c-glob (exchange-config c) p) (loc, t))
∧ (∀ w. init-config (prop-config c w)))

definition *NextPerformOp'* :: (*'p*::*finite*, *'t*::*order*, *'loc*) *configuration* ⇒ (*'p*, *'t*, *'loc*) *configuration*
⇒ *'p* ⇒ (*'loc* × *'t*) *multiset* ⇒ (*'p* × (*'loc* × *'t*)) *multiset*
⇒ (*'loc* × *'t*) *multiset* ⇒ *bool* **where**
NextPerformOp' c0 c1 p Δneg Δmint-msg Δmint-self = (
cri.next-performop' (exchange-config c0) (exchange-config c1) p Δneg Δmint-msg Δmint-self
∧ unchanged prop-config c0 c1
∧ unchanged init c0 c1)

abbreviation *NextPerformOp* **where**
NextPerformOp c0 c1 ≡ ∃ p Δneg Δmint-msg Δmint-self. NextPerformOp' c0 c1 p Δneg Δmint-msg Δmint-self

definition *NextRecvCap'*
:: (*'p*::*finite*, *'t*::*order*, *'loc*) *configuration* ⇒ (*'p*, *'t*, *'loc*) *configuration* ⇒ *'p* ⇒ *'loc* × *'t* ⇒ *bool* **where**
NextRecvCap' c0 c1 p t = (
cri.next-recvcap' (exchange-config c0) (exchange-config c1) p t

∧ *unchanged prop-config c0 c1*
∧ *unchanged init c0 c1*)

abbreviation *NextRecvCap* **where**
NextRecvCap c0 c1 ≡ ∃ *p t*. *NextRecvCap′ c0 c1 p t*

definition *NextSendUpd′* :: (*′p*::*finite*, *′t*::*order*, *′loc*) *configuration* ⇒ (*′p*, *′t*, *′loc*) *configuration*
⇒ *′p* ⇒ (*′loc* × *′t*) *set* ⇒ *bool* **where**
NextSendUpd′ c0 c1 p tt = (
cri.next-sendupd′ (*exchange-config c0*) (*exchange-config c1*) *p tt*
∧ *unchanged prop-config c0 c1*
∧ *unchanged init c0 c1*)

abbreviation *NextSendUpd* **where**
NextSendUpd c0 c1 ≡ ∃ *p tt*. *NextSendUpd′ c0 c1 p tt*

definition *NextRecvUpd′* :: (*′p*::*finite*, *′t*::*order*, *′loc*) *configuration* ⇒ (*′p*, *′t*, *′loc*) *configuration*
⇒ *′p* ⇒ *′p* ⇒ *bool* **where**
NextRecvUpd′ c0 c1 p q = (
init c0 q — Once init is set we are guaranteed that the CM transitions' premises are satisfied
∧ *cri.next-recvupd′* (*exchange-config c0*) (*exchange-config c1*) *p q*
∧ *unchanged init c0 c1*
∧ (∀ *p′*. *prop-config c1 p′* =
(*if p′* = *q*
then cm-all (*prop-config c0 q*) (*hd* (*c-msg* (*exchange-config c0*) *p q*))
else prop-config c0 p′)))

abbreviation *NextRecvUpd* **where**
NextRecvUpd c0 c1 ≡ ∃ *p q*. *NextRecvUpd′ c0 c1 p q*

definition *NextPropagate′* :: (*′p*::*finite*, *′t*::*order*, *′loc*) *configuration* ⇒ (*′p*, *′t*, *′loc*) *configuration*
⇒ *′p* ⇒ *bool* **where**
NextPropagate′ c0 c1 p = (
unchanged exchange-config c0 c1
∧ *init c1* = (*init c0*)(*p* := *True*)
∧ (∀ *p′*. *Some* (*prop-config c1 p′*) =
(*if p′* = *p*
then propagate-all (*prop-config c0 p′*)
else Some (*prop-config c0 p′*))))

abbreviation *NextPropagate* **where**
NextPropagate c0 c1 ≡ ∃ *p*. *NextPropagate′ c0 c1 p*

definition *Next′* **where**
Next′ c0 c1 = (*NextPerformOp c0 c1* ∨ *NextSendUpd c0 c1* ∨ *NextRecvUpd c0*

c1 ∨ *NextPropagate c0 c1* ∨ *NextRecvCap c0 c1* ∨ *c1* = *c0*)

abbreviation *Next* **where**
Next s ≡ *Next'* (*shd s*) (*shd* (*stl s*))

definition *FullSpec* :: ('*p* :: *finite*, '*t* :: *order*, '*loc*) *computation* ⇒ *bool* **where**
FullSpec s = (*holds InitConfig s* ∧ *alw Next s*)

lemma *NextPerformOpD*:
assumes *NextPerformOp' c0 c1 p* Δ*neg* Δ*mint-msg* Δ*mint-self*
shows
cri.next-performop' (*exchange-config c0*) (*exchange-config c1*) *p* Δ*neg* Δ*mint-msg* Δ*mint-self*
unchanged prop-config c0 c1
unchanged init c0 c1
using *assms* **unfolding** *NextPerformOp'-def* **by** *simp-all*

lemma *NextSendUpdD*:
assumes *NextSendUpd' c0 c1 p tt*
shows
cri.next-sendupd' (*exchange-config c0*) (*exchange-config c1*) *p tt*
unchanged prop-config c0 c1
unchanged init c0 c1
using *assms* **unfolding** *NextSendUpd'-def* **by** *simp-all*

lemma *NextRecvUpdD*:
assumes *NextRecvUpd' c0 c1 p q*
shows
init c0 q
cri.next-recvupd' (*exchange-config c0*) (*exchange-config c1*) *p q*
unchanged init c0 c1
(∀ *p'*. *prop-config c1 p'* =
(*if p'* = *q*
then cm-all (*prop-config c0 q*) (*hd* (*c-msg* (*exchange-config c0*) *p q*))
else prop-config c0 p'))
using *assms* **unfolding** *NextRecvUpd'-def* **by** *simp-all*

lemma *NextPropagateD*:
assumes *NextPropagate' c0 c1 p*
shows
unchanged exchange-config c0 c1
init c1 = (*init c0*)(*p* := *True*)
(∀ *p'*. *Some* (*prop-config c1 p'*) =
(*if p'* = *p*
then propagate-all (*prop-config c0 p'*)
else Some (*prop-config c0 p'*)))
using *assms* **unfolding** *NextPropagate'-def* **by** *simp-all*

lemma *NextRecvCapD*:

assumes *NextRecvCap′ c0 c1 p t*
shows
cri.next-recvcap′ (*exchange-config c0*) (*exchange-config c1*) *p t*
unchanged prop-config c0 c1
unchanged init c0 c1
using *assms* **unfolding** *NextRecvCap′-def* **by** *simp-all*

8.3 Auxiliary Lemmas

8.3.1 Auxiliary Lemmas for CM Conversion

lemma *apply-cm-is-cm*:
$\exists\, t'.\ t' \in_A$ *frontier* (*c-imp c loc*) $\wedge\ t' \leq t \Longrightarrow n \neq 0 \Longrightarrow$ *next-change-multiplicity′ c* (*apply-cm c loc t n*) *loc t n*
by (*auto simp*: *next-change-multiplicity′-def apply-cm-def intro*!: *Propagate.configuration.equality*)

lemma *update-zmultiset-commute*:
update-zmultiset (*update-zmultiset M t′ n′*) *t n* = *update-zmultiset* (*update-zmultiset M t n*) *t′ n′*
by *transfer* (*auto simp*: *equiv-zmset-def split*: *if-splits*)

lemma *apply-cm-commute*: *apply-cm* (*apply-cm c loc t n*) *loc′ t′ n′* = *apply-cm* (*apply-cm c loc′ t′ n′*) *loc t n*
unfolding *apply-cm-def Let-def*
by (*auto intro*!: *Propagate.configuration.equality simp*: *update-zmultiset-commute*)

lemma *comp-fun-commute-apply-cm*[*simp*]: *comp-fun-commute* (λ(*loc*, *t*) *c*. *apply-cm c loc t* (*f loc t*))
by (*auto intro*!: *Propagate.configuration.equality simp*: *update-zmultiset-commute comp-fun-commute-def o-def apply-cm-commute*)

lemma *ex-cm-imp-conds*:
assumes $\exists\, c'.$ *next-change-multiplicity′ c c′ loc t n*
shows $\exists\, t'.\ t' \in_A$ *frontier* (*c-imp c loc*) $\wedge\ t' \leq t\ n \neq 0$
using *assms* **by** (*auto simp*: *next-change-multiplicity′-def*)

lemma *the-cm-eq-apply-cm*:
assumes $\exists\, c'.$ *next-change-multiplicity′ c c′ loc t n*
shows *the-cm c loc t n* = *apply-cm c loc t n*
proof −
from *assms* **have** *cond*: $\exists\, t'.\ t' \in_A$ *frontier* (*c-imp c loc*) $\wedge\ t' \leq t\ n \neq 0$
using *ex-cm-imp-conds* **by** *blast*+
show *?thesis*
unfolding *the-cm-def*
apply (*rule the1-equality*)
apply (*rule next-change-multiplicity′-unique*[*OF cond*(*2*,*1*)])
unfolding *apply-cm-def next-change-multiplicity′-def*
using *cond* **apply** (*auto intro*!: *Propagate.configuration.equality*)
done

qed

lemma *apply-cm-preserves-cond*:
assumes $\forall (loc,t) \in$ *set-zmset* $\Delta.\ \exists\, t'.\ t' \in_A$ *frontier* (*c-imp c0 loc*) $\wedge\ t' \leq t$
shows $\forall (loc,t) \in$ *set-zmset* $\Delta.\ \exists\, t'.\ t' \in_A$ *frontier* (*c-imp* (*apply-cm c0 loc'* t'' *n*) *loc*) $\wedge\ t' \leq t$
using *assms* **by** (*auto simp*: *apply-cm-def*)

lemma *cm-all-eq-cm-all'*:
assumes $\forall (loc,t) \in$ *set-zmset* $\Delta.\ \exists\, t'.\ t' \in_A$ *frontier* (*c-imp c0 loc*) $\wedge\ t' \leq t$
shows *cm-all c0* Δ = *cm-all' c0* Δ
unfolding *cm-all-def cm-all'-def*
apply (*rule fold-closed-eq*[**where** $B = \{c.\ \forall (loc,t) \in$ *set-zmset* $\Delta.\ \exists\, t'.\ t' \in_A$ *frontier* (*c-imp c loc*) $\wedge\ t' \leq t\}$])
subgoal for *a* Δ
apply (*cases a*)
apply *simp*
apply (*rule the-cm-eq-apply-cm*)
apply (*rule ex1-implies-ex*)
apply (*rule next-change-multiplicity'-unique*)
apply *auto*
done
subgoal for *a* Δ
apply (*cases a*)
apply *simp*
apply (*rule apply-cm-preserves-cond*)
apply *auto*
done
subgoal
using *assms* **by** *simp*
done

lemma *cm-eq-the-cm*:
assumes *next-change-multiplicity' c c' loc t n*
shows *the-cm c loc t n* = *c'*
proof −
from *assms* **have** *cond*: $\exists\, u.\ u \in_A$ *frontier* (*c-imp c loc*) $\wedge\ u \leq t\ n \neq 0$
unfolding *next-change-multiplicity'-def* **by** *auto*
then show *?thesis*
using *next-change-multiplicity'-unique*[*OF cond*(*2*,*1*)] *assms the-cm-def*
by *auto*
qed

lemma *zcount-ps-apply-cm*:
zcount (*c-pts* (*apply-cm c loc t n*) *loc'*) *t'* = *zcount* (*c-pts c loc'*) *t'* + (*if loc* = *loc'* $\wedge$ *t* = *t' then n else 0*)
by (*auto simp*: *apply-cm-def zcount-update-zmultiset*)

lemma *zcount-pointstamps-update*: *zcount* (*c-pts* (*c*⦇*c-pts*:=*M*⦈) *loc*) *x* = *zcount*

```
(M loc) x
  by auto

lemma nop: loc1 ≠ loc2 ∨ t1 ≠ t2 ⟶
    zcount (c-pts (apply-cm c loc2 t2 (zcount Δ (loc2, t2))) loc1) t1 =
    zcount (c-pts c loc1) t1
  apply (simp add: apply-cm-def)
  using zcount-update-zmultiset
  by (simp add: zcount-update-zmultiset)

lemma fold-nop:
  zcount (c-pts (Finite-Set.fold (λ(loc', t') c. apply-cm c loc' t' (zcount Δ' (loc',
t'))) c
                        (set-zmset Δ − {(loc, t)})) loc) t
  = zcount (c-pts c loc) t
proof −
  have finite (set-zmset Δ− {(loc, t)}) using finite-set-zmset by blast
  then show ?thesis
  proof (induct set-zmset Δ − {(loc, t)} arbitrary: Δ rule: finite-induct)
    case empty
    then show ?case using fold-empty by (auto simp del: diff-shunt)
  next
    let ?f = (λ(loc', t') c. apply-cm c loc' t' (zcount Δ' (loc',t')))
    case (insert x F)
    obtain loc' t' where x-pair: x = (loc', t') by (meson surj-pair)
    from insert have nonmember: x ≠ (loc, t) by auto
    then have finite-s: finite F using insert by auto
    interpret commute: comp-fun-commute ?f
      by (simp add: comp-fun-commute-def comp-def apply-cm-commute)
    from finite-s have step1:
      Finite-Set.fold ?f c (insert x F) = ?f x (Finite-Set.fold ?f c F)
    by (metis (mono-tags, lifting) commute.fold-insert insert.hyps(1) insert.hyps(2))
    from nonmember have step2:
      zcount (c-pts (?f x (Finite-Set.fold ?f c F)) loc) t
         = zcount (c-pts (Finite-Set.fold ?f c F) loc) t
      using x-pair
      by (metis (mono-tags, lifting) case-prod-conv nop)
    with step1 and x-pair have final:
      zcount (c-pts (Finite-Set.fold ?f c (insert x F)) loc) t
         = zcount (c-pts (Finite-Set.fold ?f c F) loc) t
      by simp
     from insert(2,4) obtain Δ2 where Δ2: Δ2 = filter-zmset (λy. y≠x) Δ by
blast
    then have F = set-zmset Δ2 − {(loc, t)}
    proof −
      from Δ2 have ∗: x ∉#_z Δ2 by (simp add: not-in-iff-zmset)
      from insert(4) and nonmember have ∗∗: x ∈#_z Δ by blast
      from Δ2 ∗∗ have ∗∗∗: ∀ y. y ∈#_z Δ ⟷ (y ∈#_z Δ2 ∨ y = x)
        by (metis (mono-tags, lifting) count-filter-zmset zcount-ne-zero-iff)
```

```
    with *** have ∀ y. (y ∈ set-zmset Δ = (y ∈ (set-zmset Δ2 ∪ {x}))) by blast
    then have set-zmset Δ = set-zmset Δ2 ∪ {x} by (auto simp add: set-eq-iff)
    with insert(2,3,4) * show ?thesis
      by (metis (mono-tags, lifting) Diff-insert Diff-insert2 Diff-insert-absorb
Un-empty-right Un-insert-right)
  qed
  with final insert show ?case by metis
 qed
qed

lemma zcount-pointstamps-cm-all':
 zcount (c-pts (cm-all' c Δ) loc) x
= zcount (c-pts c loc) x + zcount Δ (loc,x)
proof −
 let ?f = (λ(loc', t') c. apply-cm c loc' t' (zcount Δ (loc',t')))
 have ?thesis
 proof (cases zcount Δ (loc,x) = 0)
  case case-nonmember: True
   then have set-zmsetΔ: set-zmset Δ − {(loc, x)} = set-zmset Δ using
zcount-eq-zero-iff by fastforce
  have zcount (c-pts (cm-all' c Δ) loc) x
        = zcount (c-pts c loc) x
   unfolding cm-all'-def
   apply (subst set-zmsetΔ[symmetric])
   apply (simp add: fold-nop)
   done
  with case-nonmember show ?thesis by auto
 next
  case case-member: False
  then have fold-rec: Finite-Set.fold ?f c (set-zmset Δ)
        = ?f (loc, x) (Finite-Set.fold ?f c (set-zmset Δ − {(loc, x)}))
  proof −
   interpret commute: comp-fun-commute (λ(loc, t) c. apply-cm c loc t (f loc
t)) for f
    by (fact comp-fun-commute-apply-cm)
   have (loc, x) ∈#z Δ
    by (meson case-member zcount-inI)
   then show ?thesis
    by (intro commute.fold-rec) simp-all
  qed
  have zcount (c-pts (Finite-Set.fold ?f c (set-zmset Δ − {(loc, x)})) loc) x
      = zcount (c-pts c loc) x by (simp add: fold-nop)
  then have zcount (c-pts (Finite-Set.fold ?f c (set-zmset Δ)) loc) x
        = zcount (c-pts (?f (loc, x) c) loc) x
   using fold-nop fold-rec by (simp add: zcount-ps-apply-cm)
  then show ?thesis
   by (simp add: zcount-ps-apply-cm cm-all'-def)
 qed
 then show ?thesis ..
```

qed

lemma *implications-apply-cm*[*simp*]: *c-imp* (*apply-cm c loc t n*) = *c-imp c*
by (*auto simp*: *apply-cm-def*)

lemma *implications-cm-all*[*simp*]:
c-imp (*cm-all' c* Δ) = *c-imp c*
unfolding *cm-all'-def Let-def*
apply (*rule fold-invar*[*OF finite-set-zmset*])
apply *auto*
done

lemma *lift-cm-inv-cm-all'*:
assumes (⋀*c0 c1 loc t n*. *P c0* ⟹ *next-change-multiplicity' c0 c1 loc t n* ⟹ *P c1*)
and ∀(*loc*,*t*)∈#$_z$Δ. ∃ *t'*. *t'* ∈$_A$ *frontier* (*c-imp c0 loc*) ∧ *t'* ≤ *t*
and *P c0*
shows *P* (*cm-all' c0* Δ)
proof −
let *?cond-invar* = λ*c*. ∀(*loc*, *t*)∈#$_z$Δ. ∃ *t'*. *t'* ∈$_A$ *frontier* (*c-imp c loc*) ∧ *t'* ≤ *t*
let *?invar* = λ*c*. *?cond-invar c* ∧ *P c*
show *?thesis*
unfolding *cm-all'-def*
apply (*rule conjunct2*[*OF fold-invar*[*OF finite-set-zmset*, *of ?invar*]])
using *assms*(*2*,*3*) **apply** *simp*
subgoal
apply *safe*
apply *auto* []
apply (*rule assms*(*1*))
apply *simp*
apply (*rule apply-cm-is-cm*)
apply *auto*
done
apply *simp*
done
qed

lemma *lift-cm-inv-cm-all*:
assumes ⋀*c0 c1 loc t n*. *P c0* ⟹ *next-change-multiplicity' c0 c1 loc t n* ⟹ *P c1*
and ∀(*loc*,*t*)∈#$_z$Δ. ∃ *t'*. *t'* ∈$_A$ *frontier* (*c-imp c0 loc*) ∧ *t'* ≤ *t*
and *P c0*
shows *P* (*cm-all c0* Δ)
apply (*subst cm-all-eq-cm-all'*)
using *assms*(*2*) **apply** *simp*
using *assms* **apply** (*rule lift-cm-inv-cm-all'*)
apply *simp-all*
done

lemma *obtain-min-worklist*:
assumes $(a\ (loc'::(\text{-} :: finite))::((t :: order)\ zmultiset)) \neq \{\#\}_z$
obtains *loc t*
where $t \in\#_z\ a\ loc$
and $\forall\, t'\ loc'.\ t' \in\#_z\ a\ loc' \longrightarrow \neg\ t' < t$
using *assms*
proof *atomize-elim*
obtain *f* **where** *f*: $f = minimal\text{-}antichain\ (\bigcup loc'.\ set\text{-}zmset\ (a\ loc'))$
by *blast*
from *assms* **obtain** x' **where** x': $x' \in (\bigcup loc'.\ set\text{-}zmset\ (a\ loc'))$
using *pos-zcount-in-zmset* **by** *fastforce*
from *assms* **have** *finite* $(\bigcup loc'.\ set\text{-}zmset\ (a\ loc'))$
using *finite-UNIV* **by** *blast*
with x' **have** $f \neq \{\}$ **unfolding** *f*
using *minimal-antichain-nonempty* **by** *blast*
then obtain *t* **where** *t*: $t \in f\ (\forall\, s{\in}f.\ \neg\ s < t)$
using *ex-min-if-finite f minimal-antichain-def* **by** *fastforce*
with *f* **have** *thesis1*: $\forall\, t'\ loc'.\ t' \in\#_z\ a\ loc' \longrightarrow \neg\ t' < t\ \exists\, loc.\ t \in\#_z\ a\ loc$
by (*simp add*: *minimal-antichain-def*)+
then show $\exists\, t\ loc.\ t \in\#_z\ a\ loc \wedge (\forall\, t'\ loc'.\ t' \in\#_z\ a\ loc' \longrightarrow \neg\ t' < t)$ **by** *blast*
qed

lemma *propagate-pointstamps-eq*:
assumes *c-work c loc* $\neq \{\#\}_z$
shows *c-pts c* = *c-pts* (*SOME c'*. $\exists$ *loc t*. *next-propagate' c c' loc t*)
proof −
from *assms* **obtain** *loc' t* **where** *loc't*: $t \in\#_z$ *c-work c loc'*
$\forall\, t'\ loc'.\ t' \in\#_z$ *c-work c loc'* $\longrightarrow \neg\ t' < t$
apply (*rule obtain-min-worklist*[*of c-work c loc*]) **by** *blast*
let *?imps* = $\lambda locX.$ *if locX* = *loc' then c-imp c locX* + $\{\#t' \in\#_z$ *c-work c locX*. $t' = t\#\}$
else c-imp c locX
let *?wl* = $\lambda locX.$ *if locX* = *loc' then* $\{\#t' \in\#_z$ *c-work c locX*. $t' \neq t\#\}$
else c-work c locX
+ *after-summary*
(*frontier-changes* (*?imps loc'*) (*c-imp c loc'*))
(*summary loc' locX*)
let *?c* = *c*(|*c-imp* := *?imps*, *c-work* := *?wl*|)
from *loc't assms* **have** *propagate*: $\exists\, c'.\ \exists\, loc\ t.$ *next-propagate' c c' loc t*
by (*intro exI*[*of* - *?c*] *exI*[*of* - *loc'*] *exI*[*of* - *t*])
(*auto simp add*: *next-propagate'-def intro*!: *Propagate.configuration.equality*)
{ fix *c' loc t*
assume *next-propagate' c c' loc t*
then have *c-pts c* = *c-pts c'*
by (*simp add*: *next-propagate'-def*)
}
with *propagate* **show** *?thesis*
by (*simp add*: *verit-sko-ex'*)

qed

lemma *propagate-all-imp-InvGlobPointstampsEq*:
Some c1 = propagate-all c0 ⟹ c-pts c0 = c-pts c1
unfolding *propagate-all-def*
using *while-option-rule*[**where** *P*=(λ*c*. *c-pts c0 = c-pts c*)
and *b*=(λ*c*. ∃ *loc*. *c-work c loc* ≠ {#}$_z$)
and *c*=(λ*c*. *SOME c'*. ∃ *loc t*. *next-propagate' c c' loc t*)]
propagate-pointstamps-eq **by** (*metis* (*no-types*, *lifting*))

lemma *exists-next-propagate'*:
assumes *c-work c loc* ≠ {#}$_z$
shows ∃ *c' loc t*. *next-propagate' c c' loc t*
proof −
from *assms* **obtain** *loc' t* **where** *loc't*:
t ∈#$_z$ *c-work c loc'*
∀ *t' loc'*. *t'* ∈#$_z$ *c-work c loc'* ⟶ ¬ *t'* < *t*
by (*rule obtain-min-worklist*)
let *?imps* = λ*locX*. *if locX = loc' then c-imp c locX* + {#*t'* ∈#$_z$ *c-work c locX*. *t'* = *t*#}
else c-imp c locX
let *?wl* = λ*locX*. *if locX = loc' then* {#*t'* ∈#$_z$ *c-work c locX*. *t'* ≠ *t*#}
else c-work c locX
+ *after-summary*
(*frontier-changes* (*?imps loc'*) (*c-imp c loc'*))
(*summary loc' locX*)
let *?c* = *c*(|*c-imp* := *?imps*, *c-work* := *?wl*|)
from *loc't assms* **show** *?thesis*
by (*intro exI*[*of* - *?c*] *exI*[*of* - *loc'*] *exI*[*of* - *t*])
(*auto simp*: *next-propagate'-def intro*!: *Propagate.configuration.equality*)
qed

lemma *lift-propagate-inv-propagate-all*:
assumes (⋀*c0 c1 loc t*. *P c0* ⟹ *next-propagate' c0 c1 loc t* ⟹ *P c1*)
and *P c0*
and *propagate-all c0 = Some c1*
shows *P c1*
apply (*rule while-option-rule*[*of P* -, *rotated*, *OF assms*(*3*)[*unfolded propagate-all-def*], *OF assms*(*2*)])
apply *clarify*
subgoal for *c loc*
apply (*drule exists-next-propagate'*)
apply (*simp add*: *assms*(*1*) *verit-sko-ex'*)
done
done

8.4 Exchange is a Subsystem of Tracker

Steps in the Tracker are valid steps in the Exchange protocol.

lemma *next-imp-exchange-next*:
Next′ c0 c1 ⟹ *cri.next′* (*exchange-config c0*) (*exchange-config c1*)
unfolding *Next′-def cri.next′-def NextPerformOp′-def NextRecvUpd′-def NextSendUpd′-def NextPropagate′-def NextRecvCap′-def*
by *auto*

lemma *alw-next-imp-exchange-next*: *alw Next s* ⟹ *alw cri.next* (*smap exchange-config s*)
by (*coinduction arbitrary*: *s rule*: *alw.coinduct*) (*auto dest*: *alwD next-imp-exchange-next*)

Any Tracker trace is a valid Exchange trace

lemma *spec-imp-exchange-spec*: *FullSpec s* ⟹ *cri.spec* (*smap exchange-config s*)
unfolding *cri.spec-def FullSpec-def*
by (*auto simp*: *InitConfig-def intro*: *alw-next-imp-exchange-next*)

lemma *lift-exchange-invariant*:
assumes ⋀*s*. *cri.spec s* ⟹ *alw* (*holds P*) *s*
shows *FullSpec s* ⟹ *alw* (λ*s*. *P* (*exchange-config* (*shd s*))) *s*
proof −
assume *FullSpec s*
note *spec-imp-exchange-spec*[*OF this*]
note *assms*[*OF this*]
then show *?thesis*
by (*auto simp*: *alw-holds-smap-conv-comp*)
qed

Lifted Exchange invariants

lemmas
exch-alw-InvCapsNonneg = *lift-exchange-invariant*[*OF cri.alw-InvCapsNonneg*, *unfolded atomize-imp*, *simplified*, *folded atomize-imp*] **and**
exch-alw-InvRecordCount = *lift-exchange-invariant*[*OF cri.alw-InvRecordCount*, *simplified atomize-imp*, *simplified*, *folded atomize-imp*] **and**
exch-alw-InvRecordsNonneg = *lift-exchange-invariant*[*OF cri.alw-InvRecordsNonneg*, *simplified atomize-imp*, *simplified*, *folded atomize-imp*] **and**
exch-alw-InvGlobVacantImpRecordsVacant = *lift-exchange-invariant*[*OF cri.alw-InvGlobVacantImpRecordsVa*
simplified atomize-imp, *simplified*, *folded atomize-imp*] **and**
exch-alw-InvGlobNonposImpRecordsNonpos = *lift-exchange-invariant*[*OF cri.alw-InvGlobNonposImpRecordsN*
simplified atomize-imp, *simplified*, *folded atomize-imp*] **and**
exch-alw-InvJustifiedGII = *lift-exchange-invariant*[*OF cri.alw-InvJustifiedGII*, *simplified atomize-imp*, *simplified*, *folded atomize-imp*] **and**
exch-alw-InvJustifiedII = *lift-exchange-invariant*[*OF cri.alw-InvJustifiedII*, *simplified atomize-imp*, *simplified*, *folded atomize-imp*] **and**
exch-alw-InvGlobNonposEqVacant = *lift-exchange-invariant*[*OF cri.alw-InvGlobNonposEqVacant*, *simplified atomize-imp*, *simplified*, *folded atomize-imp*] **and**
exch-alw-InvMsgInGlob = *lift-exchange-invariant*[*OF cri.alw-InvMsgInGlob*, *simplified atomize-imp*, *simplified*, *folded atomize-imp*] **and**
exch-alw-InvTempJustified = *lift-exchange-invariant*[*OF cri.alw-InvTempJustified*, *simplified atomize-imp*, *simplified*, *folded atomize-imp*]

8.5 Definitions

definition *safe-combined* :: (*'p*::*finite*, *'t*::*order*, *'loc*) *configuration* ⇒ *bool* **where**
safe-combined c ≡ ∀ *loc1 loc2 t s p*.
zcount (*cri.records* (*exchange-config c*)) (*loc1*, *t*) > *0* ∧ *s* ∈$_A$ *path-summary loc1 loc2* ∧ *init c p*
⟶ (∃ *t'*. *t'* ∈$_A$ *frontier* (*c-imp* (*prop-config c p*) *loc2*) ∧ *t'* ≤ *results-in t s*)

definition *safe-combined2* :: (*'p*::*finite*, *'t*::*order*, *'loc*) *configuration* ⇒ *bool* **where**
safe-combined2 c ≡ ∀ *loc1 loc2 t s p1 p2*.
zcount (*c-caps* (*exchange-config c*) *p1*) (*loc1*, *t*) > *0* ∧ *s* ∈$_A$ *path-summary loc1 loc2* ∧ *init c p2*
⟶ (∃ *t'*. *t'* ∈$_A$ *frontier* (*c-imp* (*prop-config c p2*) *loc2*) ∧ *t'* ≤ *results-in t s*)

definition *InvGlobPointstampsEq* :: (*'p* :: *finite*, *'t* :: *order*, *'loc*) *configuration* ⇒ *bool* **where**
InvGlobPointstampsEq c = (
(∀ *p loc t*. *zcount* (*c-pts* (*prop-config c p*) *loc*) *t*
= *zcount* (*c-glob* (*exchange-config c*) *p*) (*loc*, *t*)))

lemma *safe-combined-implies-safe-combined2*:
assumes *cri.InvCapsNonneg* (*exchange-config c*)
and *safe-combined c*
shows *safe-combined2 c*
unfolding *safe-combined2-def*
apply *clarify*
subgoal for *loc1 loc2 t s p1 p2*
apply (*rule assms*(*2*)[*unfolded safe-combined-def*, *rule-format*, *of loc1 t s loc2 p2*])
apply (*simp add*: *cri.records-def*)
apply (*rule add-pos-nonneg*)
apply (*subst zcount-sum*)
apply (*rule sum-pos*[**where** *y* = *p1*])
using *assms*(*1*)
apply (*simp-all add*: *cri.InvCapsNonneg-def*)
done
done

8.6 Propagate is a Subsystem of Tracker

8.6.1 CM Conditions

definition *InvMsgCMConditions* **where**
InvMsgCMConditions c = (∀ *p q*.
init c q ⟶ *c-msg* (*exchange-config c*) *p q* ≠ [] ⟶
(∀ (*loc*,*t*) ∈#$_z$ (*hd* (*c-msg* (*exchange-config c*) *p q*)). ∃ *t'*. *t'* ∈$_A$ *frontier* (*c-imp* (*prop-config c q*) *loc*) ∧ *t'* ≤ *t*))

Pointstamps in incoming messages all satisfy the CM premise, which is re-

quired during NextRecvUpd' steps.

lemma *msg-is-cm-safe*:
fixes *c* :: ('*p*::*finite*, '*t*::*order*, '*loc*) *configuration*
assumes *safe* (*prop-config c q*)
and *InvGlobPointstampsEq c*
and *cri.InvMsgInGlob* (*exchange-config c*)
and *c-msg* (*exchange-config c*) *p q* ≠ []
shows ∀(*loc*,*t*) ∈#$_z$ (*hd* (*c-msg* (*exchange-config c*) *p q*)). ∃ *t'*. *t'* ∈$_A$ *frontier* (*c-imp* (*prop-config c q*) *loc*) ∧ *t'* ≤ *t*
using *assms*(*3*)[*unfolded cri.InvMsgInGlob-def*, *rule-format*, *OF assms*(*4*)] *assms*(*1*)[*unfolded safe-def*, *rule-format*]
apply (*clarsimp simp*: *cri-less-eq-def assms*(*2*)[*unfolded InvGlobPointstampsEq-def*, *rule-format*, *symmetric*])
using *order-trans* **apply** *blast*
done

8.6.2 Propagate Safety and InvGlobPointstampsEq

To be able to use the *msg-is-cm-safe* lemma at all times and show that Propagate is a subsystem we need to prove that the specification implies Propagate's safe and the InvGlobPointstampsEq. Both of these depend on the CM conditions being satisfied during the NextRecvUpd' step and the safety proof additionally depends on other Propagate invariants, which means that we need to prove all of these jointly.

abbreviation *prop-invs* **where**
prop-invs c ≡ *inv-implications-nonneg c* ∧ *inv-imps-work-sum c*

abbreviation *prop-safe* **where**
prop-safe c ≡ *impl-safe c* ∧ *safe c*

definition *inv-init-imp-prop-safe* **where**
inv-init-imp-prop-safe c = (∀ *p*. *init c p* ⟶ *prop-safe* (*prop-config c p*))

lemma *NextRecvUpd'-preserves-prop-safe*:
assumes *prop-safe* (*prop-config c0 q*)
and *InvGlobPointstampsEq c0*
and *cri.InvMsgInGlob* (*exchange-config c0*)
and *NextRecvUpd' c0 c1 p q*
shows *prop-safe* (*prop-config c1 q*)
proof −
have *safe*: *safe* (*prop-config c0 q*)
using *assms*(*1*) **by** *blast*
note *recvupd-change* = *cri.next-recvupdD*(*1*)[*OF NextRecvUpdD*(*2*)[*OF assms*(*4*)]]
note *cm-conds* = *msg-is-cm-safe*[*OF safe assms*(*2*,*3*) *recvupd-change*]
have *safes*:
prop-safe c0 ⟹ *next-change-multiplicity' c0 c1 loc t n* ⟹ *prop-safe c1* **for** *c0 c1 loc t n*
using

```
      cm-preserves-safe
      cm-preserves-impl-safe
    by auto
  show prop-safe (prop-config c1 q)
    using
      lift-cm-inv-cm-all[rotated, OF cm-conds, of prop-safe, OF assms(1)]
      safes
      NextRecvUpdD(4)[OF assms(4)]
    by metis
qed

lemma NextRecvUpd'-preserves-InvGlobPointstampsEq:
  assumes impl-safe (prop-config c0 q) ∧ safe (prop-config c0 q)
    and   InvGlobPointstampsEq c0
    and   cri.InvMsgInGlob (exchange-config c0)
    and   NextRecvUpd' c0 c1 p q
  shows   InvGlobPointstampsEq c1
proof −
  have safe: safe (prop-config c0 q)
    using assms(1) by blast
 note recvupd-change = cri.next-recvupdD(1)[OF NextRecvUpdD(2)[OF assms(4)]]
  note cm-conds = msg-is-cm-safe[OF safe assms(2,3) recvupd-change]
  show InvGlobPointstampsEq c1
    using
      assms(2,4)
      cm-conds
   unfolding NextRecvUpd'-def cri.next-recvupd'-def Let-def InvGlobPointstampsEq-def
    by (clarsimp simp: zcount-pointstamps-cm-all' cm-all-eq-cm-all')+
qed

— Whenever some worker p propagates it ends up in a Propagate-safe state
lemma NextPropagate'-causes-safe:
  assumes NextPropagate' c0 c1 p
    and   inv-imps-work-sum (prop-config c1 p)
    and   inv-implications-nonneg (prop-config c1 p)
  shows   safe (prop-config c1 p) impl-safe (prop-config c1 p)
proof −
  from assms(1) have Some (prop-config c1 p) = propagate-all (prop-config c0 p)
    by (simp add: NextPropagate'-def)
  then have wl: c-work (prop-config c1 p) loc = {#}_z for loc
    unfolding propagate-all-def
    by (subst (asm) eq-commute) (auto dest: while-option-stop)
  show safe (prop-config c1 p) impl-safe (prop-config c1 p)
    by (rule safe[OF assms(2,3) wl]) (rule impl-safe[OF assms(2,3) wl])
qed

— NextPropagate' preserves Propagate-safe at all workers
lemma NextPropagate'-preserves-safe:
  assumes NextPropagate' c0 c1 q
```

```
    and   inv-imps-work-sum (prop-config c1 p)
    and   inv-implications-nonneg (prop-config c1 p)
    and   safe (prop-config c0 p)
  shows   safe (prop-config c1 p)
  apply (cases p=q)
  subgoal
    using assms(1–3) by (auto intro: NextPropagate'-causes-safe)
  subgoal
    using assms(1,4) by (auto dest: spec[of - p] simp: NextPropagate'-def)
  done

lemma NextPropagate'-preserves-impl-safe:
  assumes NextPropagate' c0 c1 q
    and   inv-imps-work-sum (prop-config c1 p)
    and   inv-implications-nonneg (prop-config c1 p)
    and   impl-safe (prop-config c0 p)
  shows   impl-safe (prop-config c1 p)
  apply (cases p=q)
  subgoal
    using assms(1–3) by (auto intro: NextPropagate'-causes-safe)
  subgoal
    using assms(1,4) by (auto dest: spec[of - p] simp: NextPropagate'-def)
  done

lemma NextRecvUpd'-preserves-inv-init-imp-prop-safe:
  assumes cri.InvMsgInGlob (exchange-config c0)
    and   inv-init-imp-prop-safe c0
    and   InvGlobPointstampsEq c0
    and   NextRecvUpd' c0 c1 p q
  shows   inv-init-imp-prop-safe c1
  using assms(2) unfolding inv-init-imp-prop-safe-def
  apply clarify
  subgoal for p
    apply (cases p=q)
    subgoal
      apply (drule spec[of -p])
      apply (simp add: NextRecvUpdD(1)[OF assms(4)])
      apply (drule NextRecvUpd'-preserves-prop-safe[OF - assms(3,1,4)])
      apply simp
      done
    subgoal
      using NextRecvUpdD(3,4)[OF assms(4)] by fastforce
    done
  done

lemma NextRecvUpd'-preserves-prop-invs:
  assumes cri.InvMsgInGlob (exchange-config c0)
    and   inv-init-imp-prop-safe c0
    and   ∀ p. prop-invs (prop-config c0 p)
```

```
    and   InvGlobPointstampsEq c0
    and   NextRecvUpd′ c0 c1 p q
  shows   ∀ p. prop-invs (prop-config c1 p)
proof −
  have safe: safe (prop-config c0 q)
    using NextRecvUpdD(1) assms(2,5) inv-init-imp-prop-safe-def by blast
 note recvupd-change = cri.next-recvupdD(1)[OF NextRecvUpdD(2)[OF assms(5)]]
  note cm-conds = msg-is-cm-safe[OF safe assms(4,1) recvupd-change]
  have invs:
   prop-invs c0 ⟹ next-change-multiplicity′ c0 c1 loc t n ⟹ prop-invs c1 for c0
c1 loc t n
    using
      cm-preserves-inv-imps-work-sum
      cm-preserves-inv-implications-nonneg
    by auto
  show ∀ q. prop-invs (prop-config c1 q)
    apply rule
    subgoal for q′
      apply (cases q′=q)
      subgoal
        using
       lift-cm-inv-cm-all[rotated, OF cm-conds, of prop-invs, OF assms(3)[rule-format]]
          invs
          NextRecvUpdD(4)[OF assms(5)]
        by metis
      subgoal
        using NextRecvUpdD(4) assms(3) assms(5) by fastforce
      done
    done
qed

lemma NextPropagate′-preserves-prop-invs:
  assumes prop-invs (prop-config c0 q)
    and   NextPropagate′ c0 c1 p
  shows   prop-invs (prop-config c1 q)
  apply (cases p=q)
  subgoal
    using
      assms(1)
      lift-propagate-inv-propagate-all[
        of prop-invs,
        rotated 2,
      OF NextPropagateD(3)[OF assms(2), rule-format, of p, simplified, symmetric]]
   by (simp add: iiws-imp-iipwn p-preserves-inv-implications-nonneg p-preserves-inv-imps-work-sum)
  subgoal
    by (metis NextPropagateD(3) assms(1) assms(2) option.simps(1))
  done

lemma NextPropagate′-preserves-inv-init-imp-prop-safe:
```

```
  assumes prop-invs (prop-config c0 p)
    and   inv-init-imp-prop-safe c0
    and   NextPropagate' c0 c1 p
  shows   inv-init-imp-prop-safe c1
  using assms(2) unfolding inv-init-imp-prop-safe-def
  apply clarsimp
  subgoal for p'
    apply (cases p'=p)
    subgoal
      using NextPropagate'-preserves-prop-invs[OF assms(1,3)]
      using NextPropagate'-causes-safe(1,2)[OF assms(3)] by blast
    subgoal
      using NextPropagateD(2,3)[OF assms(3)]
      by (auto dest: spec[of - p'])
    done
  done

lemma Next'-preserves-invs:
  assumes cri.InvMsgInGlob (exchange-config c0)
    and   inv-init-imp-prop-safe c0
    and   InvGlobPointstampsEq c0
    and   Next' c0 c1
    and   ∀ p. prop-invs (prop-config c0 p)
  shows
    inv-init-imp-prop-safe c1
    ∀ p. prop-invs (prop-config c1 p)
    InvGlobPointstampsEq c1
  subgoal
    using assms(4) unfolding Next'-def
    using assms(2)
    apply (elim disjE)
    subgoal
      unfolding inv-init-imp-prop-safe-def
      using NextPerformOpD(2,3) by fastforce
    subgoal
      unfolding inv-init-imp-prop-safe-def
      using NextSendUpdD(2,3) by fastforce
    subgoal
      using NextRecvUpd'-preserves-inv-init-imp-prop-safe[OF assms(1,2,3)]
      by blast
    subgoal
      using NextPropagate'-preserves-inv-init-imp-prop-safe assms(5) by blast
    subgoal
      unfolding inv-init-imp-prop-safe-def
      using NextRecvCapD(2,3) by fastforce
    subgoal by simp
    done
  subgoal
    using assms(4) unfolding Next'-def
```

using *assms*(*5*)
apply (*elim disjE*)
subgoal
using *NextPerformOpD*(*2*) **by** *fastforce*
subgoal
using *NextSendUpdD*(*2*) **by** *fastforce*
subgoal
using *assms*(*1*,*2*,*3*) *NextRecvUpd′-preserves-prop-invs* **by** *blast*
subgoal
using *NextPropagate′-preserves-prop-invs* **by** *blast*
subgoal
unfolding *inv-init-imp-prop-safe-def*
using *NextRecvCapD*(*2*,*3*) **by** *fastforce*
subgoal by *simp*
done
subgoal
using *assms*(*4*) **unfolding** *Next′-def*
using *assms*(*3*)
apply (*elim disjE*)
subgoal
by (*metis InvGlobPointstampsEq-def NextPerformOpD*(*1*,*2*) *cri.next-performopD*(*7*))
subgoal
by (*metis InvGlobPointstampsEq-def NextSendUpdD*(*1*,*2*) *cri.next-sendupdD*(*5*))
subgoal
using *NextRecvUpdD*(*1*) *NextRecvUpd′-preserves-InvGlobPointstampsEq assms*(*1*,*2*)
inv-init-imp-prop-safe-def **by** *blast*
subgoal
unfolding *NextPropagate′-def InvGlobPointstampsEq-def*
using *propagate-all-imp-InvGlobPointstampsEq*
by (*metis option.inject*)
subgoal
by (*metis InvGlobPointstampsEq-def NextRecvCapD*(*1*) *NextRecvCapD*(*2*)
cri.next-recvcapD(*4*))
subgoal by *simp*
done
done

lemma *init-imp-InvGlobPointstampsEq*: *InitConfig c* ⟹ *InvGlobPointstampsEq*
c
by (*simp add*: *InitConfig-def cri.init-config-def InvGlobPointstampsEq-def*)

lemma *init-imp-inv-init-imp-prop-safe*: *InitConfig c* ⟹ *inv-init-imp-prop-safe c*
by (*simp add*: *inv-init-imp-prop-safe-def InitConfig-def*)

lemma *init-imp-prop-invs*: *InitConfig c* ⟹ ∀ *p*. *prop-invs* (*prop-config c p*)
by (*simp add*: *InitConfig-def init-imp-inv-implications-nonneg init-imp-inv-imps-work-sum*)

abbreviation *all-invs* **where**
all-invs c ≡ *InvGlobPointstampsEq c* ∧ *inv-init-imp-prop-safe c* ∧ (∀ *p*. *prop-invs*

(*prop-config c p*))

lemma *alw-Next'-alw-invs*:
assumes *holds all-invs s*
and *alw* (*holds* (λ*c*. *cri.InvMsgInGlob* (*exchange-config c*))) *s*
and *alw Next s*
shows *alw* (*holds all-invs*) *s*
using *assms*
apply (*coinduction arbitrary*: *s*)
apply *clarsimp*
apply *safe*
apply (*metis* (*mono-tags*, *lifting*) *alw-holds2 Next'-preserves-invs*(*3*) *alwD*)
apply (*metis* (*mono-tags*, *lifting*) *alw-holds2 Next'-preserves-invs*(*1*) *alwD*)
apply (*metis* (*mono-tags*, *lifting*) *alw-holds2 Next'-preserves-invs*(*2*) *alwD*)
apply (*metis* (*mono-tags*, *lifting*) *alw-holds2 Next'-preserves-invs*(*2*) *alwD*)
apply *auto*
done

lemma *alw-invs*: *FullSpec s* ⟹ *alw* (*holds all-invs*) *s*
apply (*frule exch-alw-InvMsgInGlob*)
unfolding *FullSpec-def*
apply *clarsimp*
apply (*frule init-imp-InvGlobPointstampsEq*)
apply (*frule init-imp-inv-init-imp-prop-safe*)
apply (*drule init-imp-prop-invs*)
by (*simp add*: *alw-Next'-alw-invs alw-mono*)

lemma *alw-InvGlobPointstampsEq*: *FullSpec s* ⟹ *alw* (*holds InvGlobPointstampsEq*) *s*
using *alw-invs alw-mono holds-mono* **by** *blast*

lemma *alw-inv-init-imp-prop-safe*: *FullSpec s* ⟹ *alw* (*holds inv-init-imp-prop-safe*) *s*
using *alw-invs alw-mono holds-mono* **by** *blast*

lemma *alw-holds-conv-shd*: *alw* (*holds* φ) *s* = *alw* (λ*s*. φ (*shd s*)) *s*
by (*simp add*: *alw-iff-sdrop*)

lemma *alw-prop-invs*: *FullSpec s* ⟹ *alw* (*holds* (λ*c*. ∀ *p*. *prop-invs* (*prop-config c p*))) *s*
by (*auto*
intro: *alw-mono*[*of holds all-invs s holds* (λ*c*. ∀ *p*. *prop-invs* (*prop-config c p*))]
dest: *alw-invs*)

lemma *nrec-pts-delayed*:
assumes *cri.InvGlobNonposImpRecordsNonpos* (*exchange-config c*)
and *zcount* (*cri.records* (*exchange-config c*)) *x* > *0*
shows ∃ *x'*. *x'* ≤$_p$ *x* ∧ *zcount* (*c-glob* (*exchange-config c*) *p*) *x'* > *0*
proof −

from *assms* **have** *r*: $\forall\, p.\ \neg$ *cri.nonpos-upto* (*c-glob* (*exchange-config c*) *p*) *x*
unfolding *cri.InvGlobNonposImpRecordsNonpos-def cri.nonpos-upto-def*
by (*metis linorder-not-less cri.order.order-iff-strict*)
show *?thesis*
using *r*[*rule-format*, *of p*]
by (*auto simp*: *cri.nonpos-upto-def not-le*)
qed

lemma *help-lemma*:
assumes *0* < *zcount* (*c-pts* (*prop-config c p*) *loc0*) *t0*
and (*loc0*, *t0*) $\leq_p$ (*loc1*, *t1*)
and *s2* $\in_A$ *path-summary loc1 loc2*
and *safe* (*prop-config c p*)
shows $\exists$ *t2*. (*t2* $\leq$ *results-in t1 s2*
$\wedge$ *t2* $\in_A$ *frontier* (*c-imp* (*prop-config c p*) *loc2*))
proof −
from *assms*(*2*) **obtain** *s1* **where** *s1*: *s1* $\in_A$ *path-summary loc0 loc1 results-in t0 s1* $\leq$ *t1*
by (*auto simp add*: *cri-less-eq-def*)
from *s1*(*1*) *assms*(*3*) **obtain** *s-full* **where** *s-full*: *s-full* $\in_A$ *path-summary loc0 loc2 s-full* $\leq$ *followed-by s1 s2*
using *flow.path-weight-elem-trans* **by** *blast*
from *s-full*(*1*) *assms*(*1*,*4*) **obtain** *t2* **where** *t2*:
t2 $\in_A$ *frontier* (*c-imp* (*prop-config c p*) *loc2*) *t2* $\leq$ *results-in t0 s-full*
unfolding *safe-def* **by** *blast*
from *t2*(*2*) **and** *s-full*(*2*) **have** *t2* $\leq$ *results-in* (*results-in t0 s1*) *s2*
by (*metis followed-by-summary order-trans results-in-mono*(*2*))
with *s1*(*2*) **have** *t2* $\leq$ *results-in t1 s2* **by** (*meson order.trans results-in-mono*(*1*))
with *t2*(*1*) **show** *?thesis* **by** *auto*
qed

— Lift an invariant's preservation proof over *next-propagate′* to NextPropagate' transitions
lemma *lift-prop-inv-NextPropagate′*:
assumes ($\bigwedge$*c0 c1 loc t*. *P c0* $\Longrightarrow$ *next-propagate′ c0 c1 loc t* $\Longrightarrow$ *P c1*)
shows *P* (*prop-config c0 p′*) $\Longrightarrow$ *NextPropagate′ c0 c1 p* $\Longrightarrow$ *P* (*prop-config c1 p′*)
proof −
assume *pc0*: *P* (*prop-config c0 p′*)
assume *np*: *NextPropagate′ c0 c1 p*
have *n-p*: ($\bigwedge$*c0 c1*. *P c0* $\Longrightarrow$ *next-propagate c0 c1* $\Longrightarrow$ *P c1*)
using *assms* **by** *auto*
let *?f* = λ*c*. *SOME c′*. *next-propagate c c′*
let *?b* = λ*c*. $\exists$ *loc*. *c-work c loc* $\neq$ $\{\#\}_z$
from *np* **have** *pc1*: *Some* (*prop-config c1 p*) = *propagate-all* (*prop-config c0 p*)
by (*simp add*: *NextPropagate′-def*)
show *?thesis*
apply (*cases p′*=*p*)
subgoal

apply (*rule while-option-rule*[*of P ?b ?f prop-config c0 p*])
apply (*rule n-p*)
apply *assumption*
apply (*rule iffD1*[*OF verit-sko-ex*])
apply (*elim exE*)
apply (*rule exists-next-propagate′*)
apply *assumption*
using *pc1* **apply** (*simp add*: *propagate-all-def*)
using *pc0* **apply** *simp*
done
subgoal
using *np pc0* **by** (*auto simp*: *NextPropagate′-def dest*!: *spec*[*of - p′*])
done
qed

8.6.3 Propagate is a Subsystem

lemma *NextRecvUpd′-next′*:
assumes *safe* (*prop-config c0 q*)
and *InvGlobPointstampsEq c0*
and *cri.InvMsgInGlob* (*exchange-config c0*)
and *NextRecvUpd′ c0 c1 p q*
shows *next′*$^{++}$ (*prop-config c0 q′*) (*prop-config c1 q′*)
apply (*subst NextRecvUpdD*(*4*)[*OF assms*(*4*), *rule-format*])
apply *simp*
apply *safe*
subgoal
apply (*subst cm-all-eq-cm-all′*)
apply *clarsimp*
apply (*drule assms*(*3*)[*unfolded cri.InvMsgInGlob-def*, *rule-format*, *OF cri.next-recvupdD*(*1*)[*OF NextRecvUpdD*(*2*)[*OF assms*(*4*)]]])
apply *clarsimp*
subgoal for *loc t loc′ t′*
apply (*subst* (*asm*) *assms*(*2*)[*unfolded InvGlobPointstampsEq-def*, *rule-format*, *symmetric*])
apply (*clarsimp simp*: *cri-less-eq-def*)
subgoal for *s*
using *assms*(*1*)[*unfolded safe-def*, *rule-format*, *of loc′ t′ s loc*]
apply −
apply (*drule meta-mp*)
apply *simp*
apply *clarsimp*
subgoal for *t″*
apply (*clarsimp intro*!: *exI*[*of - t″*])
using *order-trans* **apply** *blast*
done
done
done
apply (*rule lift-cm-inv-cm-all′*)

```
      apply (rule tranclp.intros(2))
       apply (auto simp: next'-def) [2]
     apply clarsimp
   apply (drule assms(3)[unfolded cri.InvMsgInGlob-def, rule-format, OF cri.next-recvupdD(1)[OF
NextRecvUpdD(2)[OF assms(4)]]])
     apply clarsimp
    subgoal for loc t loc' t'
     apply (subst (asm) assms(2)[unfolded InvGlobPointstampsEq-def, rule-format,
symmetric])
      apply (clarsimp simp: cri-less-eq-def)
      subgoal for s
        using assms(1)[unfolded safe-def, rule-format, of loc' t' s loc]
        apply −
        apply (drule meta-mp)
         apply simp
        apply clarsimp
        subgoal for t''
          apply (clarsimp intro!: exI[of - t''])
          using order-trans apply blast
          done
        done
      done
    apply (auto simp: next'-def)
    done
  apply (auto simp: next'-def)
  done

lemma NextPropagate'-next':
  assumes NextPropagate' c0 c1 p
  shows   next'++ (prop-config c0 q) (prop-config c1 q)
  apply (cases p=q)
  subgoal
    apply (rule lift-propagate-inv-propagate-all[of - prop-config c0 p])
     apply (rule tranclp.intros(2))
      apply (auto simp: next'-def NextPropagateD(3)[OF assms, rule-format])
    done
  subgoal
   by (metis NextPropagateD(3) assms next'-def option.simps(1) tranclp.intros(1))
  done

lemma next-imp-propagate-next:
  assumes inv-init-imp-prop-safe c0
    and   InvGlobPointstampsEq c0
    and   cri.InvMsgInGlob (exchange-config c0)
  shows   Next' c0 c1 ⟹ next'++ (prop-config c0 p) (prop-config c1 p)
  unfolding Next'-def NextPerformOp'-def NextSendUpd'-def NextRecvCap'-def
  apply safe
  subgoal by (auto simp: next'-def)
  subgoal by (auto simp: next'-def)
```

subgoal for p' q
using *assms*(*1*)[*unfolded inv-init-imp-prop-safe-def*, *rule-format*, *of q*]
apply −
apply (*drule meta-mp*)
apply (*rule NextRecvUpdD*(*1*))
apply *simp*
apply (*cases q*=*p*)
apply (*auto dest*!: *NextRecvUpd'-next'*[*rotated*, *OF assms*(*2*−)]) []
apply (*auto simp add*: *NextRecvUpd'-def next'-def*)
done
subgoal by (*rule NextPropagate'-next'*)
subgoal by (*auto simp*: *next'-def*)
subgoal by (*auto simp*: *next'-def*)
done

lemma *alw-next-imp-propagate-next*:
assumes *alw* (*holds inv-init-imp-prop-safe*) *s*
and *alw* (*holds InvGlobPointstampsEq*) *s*
and *alw* (*holds cri.InvMsgInGlob*) (*smap exchange-config s*)
and *alw Next s*
shows *alw* (*relates* (*next'*$^{++}$)) (*smap* (λ*s*. *prop-config s p*) *s*)
using *assms* **by** (*coinduction arbitrary*: *s rule*: *alw.coinduct*) (*auto intro*!: *next-imp-propagate-next simp*: *relates-def alw-holds-smap-conv-comp*)

Any Tracker trace is a valid Propagate trace (using the transitive closure of next, since tracker may take multiple propagate steps at once).

lemma *spec-imp-propagate-spec*: *FullSpec s* ⟹ (*holds init-config aand alw* (*relates* (*next'*$^{++}$))) (*smap* (λ*c*. *prop-config c p*) *s*)
apply (*frule alw-inv-init-imp-prop-safe*)
apply (*frule alw-InvGlobPointstampsEq*)
apply (*frule spec-imp-exchange-spec*)
apply (*drule cri.alw-InvMsgInGlob*)
apply (*auto intro*!: *alw-next-imp-propagate-next simp*: *FullSpec-def InitConfig-def*)
done

8.7 Safety Proofs

lemma *safe-satisfied*:
assumes *cri.InvGlobNonposImpRecordsNonpos* (*exchange-config c*)
and *inv-init-imp-prop-safe c*
and *InvGlobPointstampsEq c*
shows *safe-combined c*
proof −
{
fix *loc1 loc2 t s p*
assume *as*: *0* < *zcount* (*cri.records* (*exchange-config c*)) (*loc1*, *t*)
s ∈$_A$ *path-summary loc1 loc2 init c p*
obtain *loc0 t0* **where** *delayed*:
(*loc0*, *t0*) ≤$_p$ (*loc1*, *t*) *0* < *zcount* (*c-glob* (*exchange-config c*) *p*) (*loc0*, *t0*)

```
    using nrec-pts-delayed[OF assms(1) as(1)]
    by fast
  with as(2,3) assms(2) have
    ∃ t2. t2 ≤ results-in t s ∧ t2 ∈A frontier (c-imp (prop-config c p) loc2)
    using help-lemma delayed
    by (metis InvGlobPointstampsEq-def assms(3) inv-init-imp-prop-safe-def)
 }
 then show ?thesis
  unfolding safe-combined-def by blast
qed

lemma alw-safe-combined: FullSpec s ⟹ alw (holds safe-combined) s
 apply (frule alw-inv-init-imp-prop-safe)
 apply (frule exch-alw-InvGlobNonposImpRecordsNonpos)
 apply (drule alw-InvGlobPointstampsEq)
 apply (coinduction arbitrary: s rule: alw.coinduct)
 apply clarsimp
 apply (rule conjI)
  apply (metis alwD alw-holds2 safe-satisfied)
 apply (rule disjI1)
 apply blast
 done

lemma alw-safe-combined2: FullSpec s ⟹ alw (holds safe-combined2) s
 apply (frule exch-alw-InvCapsNonneg)
 apply (drule alw-safe-combined)
 apply (simp add: alw-iff-sdrop safe-combined-implies-safe-combined2)
 done

lemma alw-implication-frontier-eq-implied-frontier:
 FullSpec s ⟹
  alw (holds (λc. worklists-vacant-to (prop-config c p) b ⟶
    b ∈A frontier (c-imp (prop-config c p) loc) ⟷ b ∈A implied-frontier (c-pts
(prop-config c p)) loc)) s
 by (drule alw-prop-invs)
    (auto simp: implication-frontier-iff-implied-frontier-vacant all-imp-alw elim:
alw-mp)

end
```

References

[1] Github: Timely dataflow.

[2] M. Abadi, F. McSherry, D. G. Murray, and T. L. Rodeheffer. Formal analysis of a distributed algorithm for tracking progress. In D. Beyer and

M. Boreale, editors, *FMOODS/FORTE 2013*, volume 7892 of *LNCS*, pages 5–19. Springer, 2013.

[3] M. Brun, S. Decova, A. Lattuada, and D. Traytel. Verified progress tracking for timely dataflow. In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. To appear.

[4] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In M. Kaminsky and M. Dahlin, editors, *SOSP 2013*, pages 439–455. ACM, 2013.