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 $xs \neq [] \Longrightarrow sum\text{-}list\ (tl\ xs) = (-\ hd\ xs) + sum\text{-}list\ xs$
⟨*proof*⟩

2.2 Sums

lemma *Sum-eq-pick-changed-elem*:
assumes *finite M*
and $m \in M\ f\ m = g\ m + \Delta$
and $\bigwedge n.\ n \neq m \wedge n \in M \Longrightarrow f\ n = g\ n$
shows $(\sum x{\in}M.\ f\ x) = (\sum x{\in}M.\ g\ x) + \Delta$
⟨*proof*⟩

lemma *sum-pos-ex-elem-pos*: $(0{::}int) < (\sum m{\in}M.\ f\ m) \Longrightarrow \exists\, m{\in}M.\ 0 < f\ m$
⟨*proof*⟩

lemma *sum-if-distrib-add*: $finite\ A \Longrightarrow b \in A \Longrightarrow (\sum a{\in}A.\ if\ a{=}b\ then\ X\ b + Y\ a\ else\ X\ a) = (\sum a{\in}A.\ X\ a) + Y\ b$
⟨*proof*⟩

2.3 Partial Orders

lemma (**in** *order*) *order-finite-set-exists-foundation*:
fixes $t :: 'a$
assumes *finite M*
and $t \in M$
shows $\exists\, s{\in}M.\ s \leq t \wedge (\forall\, u{\in}M.\ \neg\ u < s)$
⟨*proof*⟩

lemma *order-finite-set-obtain-foundation*:
fixes *t* :: - :: *order*
assumes *finite M*
and $t \in M$
obtains *s* **where** $s \in M\ s \leq t\ \forall\, u{\in}M.\ \neg\ u < s$
⟨*proof*⟩

2.4 Multisets

lemma *finite-nonzero-count*: $finite\ \{t.\ count\ M\ t > 0\}$
⟨*proof*⟩

lemma *finite-count*[*simp*]: $finite\ \{t.\ count\ M\ t > i\}$
⟨*proof*⟩

2.5 Signed Multisets

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

lemma *finite-zcount-pos*[*simp*]: $finite\ \{t.\ zcount\ M\ t > 0\}$

⟨*proof*⟩

lemma *finite-zcount-neg*[*simp*]: *finite* $\{t.\ zcount\ M\ t < 0\}$
⟨*proof*⟩

lemma *pos-zcount-in-zmset*: $0 < zcount\ M\ x \Longrightarrow x \in\#_z M$
⟨*proof*⟩

lemma *zmset-elem-nonneg*: $x \in\#_z M \Longrightarrow (\bigwedge x.\ x \in\#_z M \Longrightarrow 0 \leq zcount\ M\ x) \Longrightarrow 0 < zcount\ M\ x$
⟨*proof*⟩

lemma *zero-le-sum-single*: $0 \leq zcount\ (\sum x{\in}M.\ \{\#f\ x\#\}_z)\ t$
⟨*proof*⟩

lemma *mem-zmset-of*[*simp*]: $x \in\#_z zmset\text{-}of\ M \longleftrightarrow x \in\# M$
⟨*proof*⟩

lemma *mset-neg-minus*: $mset\text{-}neg\ (abs\text{-}zmultiset\ (Mp,Mn)) = Mn - Mp$
⟨*proof*⟩

lemma *mset-pos-minus*: $mset\text{-}pos\ (abs\text{-}zmultiset\ (Mp,Mn)) = Mp - Mn$
⟨*proof*⟩

lemma *mset-neg-sum-set*: $(\bigwedge m.\ m \in M \Longrightarrow mset\text{-}neg\ (f\ m) = \{\#\}) \Longrightarrow mset\text{-}neg\ (\sum m{\in}M.\ f\ m) = \{\#\}$
⟨*proof*⟩

lemma *mset-neg-empty-iff*: $mset\text{-}neg\ M = \{\#\} \longleftrightarrow (\forall t.\ 0 \leq zcount\ M\ t)$
⟨*proof*⟩

lemma *mset-neg-zcount-nonneg*: $mset\text{-}neg\ M = \{\#\} \Longrightarrow 0 \leq zcount\ M\ t$
⟨*proof*⟩

lemma *in-zmset-conv-pos-neg-disj*: $x \in\#_z M \longleftrightarrow x \in\# mset\text{-}pos\ M \vee x \in\# mset\text{-}neg\ M$
⟨*proof*⟩

lemma *in-zmset-notin-mset-pos*[*simp*]: $x \in\#_z M \Longrightarrow x \notin\# mset\text{-}pos\ M \Longrightarrow x \in\# mset\text{-}neg\ M$
⟨*proof*⟩

lemma *in-zmset-notin-mset-neg*[*simp*]: $x \in\#_z M \Longrightarrow x \notin\# mset\text{-}neg\ M \Longrightarrow x \in\# mset\text{-}pos\ M$
⟨*proof*⟩

lemma *in-mset-pos-in-zmset*: $x \in\# mset\text{-}pos\ M \Longrightarrow x \in\#_z M$
⟨*proof*⟩

lemma *in-mset-neg-in-zmset*: $x \in\#$ *mset-neg* $M \Longrightarrow x \in\#_z M$
⟨*proof*⟩

lemma *set-zmset-eq-set-mset-union*: *set-zmset* M = *set-mset* (*mset-pos* M) $\cup$ *set-mset* (*mset-neg* M)
⟨*proof*⟩

lemma *member-mset-pos-iff-zcount*: $x \in\#$ *mset-pos* $M \longleftrightarrow 0 <$ *zcount* M x
⟨*proof*⟩

lemma *member-mset-neg-iff-zcount*: $x \in\#$ *mset-neg* $M \longleftrightarrow$ *zcount* M $x < 0$
⟨*proof*⟩

lemma *mset-pos-mset-neg-disjoint*[*simp*]: *set-mset* (*mset-pos* Δ) $\cap$ *set-mset* (*mset-neg* Δ) = {}
⟨*proof*⟩

lemma *zcount-sum*: *zcount* $(\sum M{\in}MM.\ f\ M)\ t = (\sum M{\in}MM.$ *zcount* $(f\ M)\ t)$
⟨*proof*⟩

lemma *zcount-filter-invariant*: *zcount* $\{\#\ t'{\in}\#_z M.\ t'{=}t\ \#\}\ t$ = *zcount* M t
⟨*proof*⟩

lemma *in-filter-zmset-in-zmset*[*simp*]: $x \in\#_z$ *filter-zmset* P $M \Longrightarrow x \in\#_z M$
⟨*proof*⟩

lemma *pos-filter-zmset-pos-zmset*[*simp*]: $0 <$ *zcount* (*filter-zmset* P M) $x \Longrightarrow 0 <$ *zcount* M x
⟨*proof*⟩

lemma *neg-filter-zmset-neg-zmset*[*simp*]: $0 >$ *zcount* (*filter-zmset* P M) $x \Longrightarrow 0 >$ *zcount* M x
⟨*proof*⟩

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))
⟨*proof*⟩

lemma *zcount-update-zmultiset*: *zcount* (*update-zmultiset* M t n) t' = *zcount* M t' + (*if* $t = t'$ *then* n *else* 0)
⟨*proof*⟩

lemma (**in** *order*) *order-zmset-exists-foundation*:
fixes $t :: 'a$
assumes $0 <$ *zcount* M t
shows $\exists s.\ s \le t \land 0 <$ *zcount* M $s \land (\forall u.\ 0 <$ *zcount* M $u \longrightarrow \neg\ u < s)$
⟨*proof*⟩

lemma (**in** *order*) *order-zmset-exists-foundation'*:
  fixes $t :: 'a$
  assumes $0 < zcount\ M\ t$
  shows $\exists s.\ s \leq t \wedge 0 < zcount\ M\ s \wedge (\forall u{<}s.\ zcount\ M\ u \leq 0)$
  ⟨*proof*⟩

lemma (**in** *order*) *order-zmset-exists-foundation-neg*:
  fixes $t :: 'a$
  assumes $zcount\ M\ t < 0$
  shows $\exists s.\ s \leq t \wedge zcount\ M\ s < 0 \wedge (\forall u.\ zcount\ M\ u < 0 \longrightarrow \neg\ u < s)$
  ⟨*proof*⟩

lemma (**in** *order*) *order-zmset-exists-foundation-neg'*:
  fixes $t :: 'a$
  assumes $zcount\ M\ t < 0$
  shows $\exists s.\ s \leq t \wedge zcount\ M\ s < 0 \wedge (\forall u{<}s.\ 0 \leq zcount\ M\ u)$
  ⟨*proof*⟩

lemma (**in** *order*) *elem-order-zmset-exists-foundation*:
  fixes $x :: 'a$
  assumes $x \in\#_z M$
  shows $\exists s{\in}\#_z M.\ s \leq x \wedge (\forall u{\in}\#_z M.\ \neg\ u < s)$
  ⟨*proof*⟩

2.5.1 Image of a Signed Multiset

lift-definition *image-zmset* :: $('a \Rightarrow 'b) \Rightarrow 'a\ zmultiset \Rightarrow 'b\ zmultiset$ **is**
  $\lambda f\ (M,\ N).\ (image\text{-}mset\ f\ M,\ image\text{-}mset\ f\ N)$
  ⟨*proof*⟩

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
  $\{\#e.\ x \in\#_z M\#\} \rightleftharpoons CONST\ image\text{-}zmset\ (\lambda x.\ e)\ M$

lemma *image-zmset-empty*[*simp*]: $image\text{-}zmset\ f\ \{\#\}_z = \{\#\}_z$
  ⟨*proof*⟩

lemma *image-zmset-single*[*simp*]: $image\text{-}zmset\ f\ \{\#x\#\}_z = \{\#f\ x\#\}_z$
  ⟨*proof*⟩

lemma *image-zmset-union*[*simp*]: $image\text{-}zmset\ f\ (M + N) = image\text{-}zmset\ f\ M + image\text{-}zmset\ f\ N$
  ⟨*proof*⟩

lemma *image-zmset-Diff*[*simp*]: *image-zmset* f $(A - B)$ = *image-zmset* f A − *image-zmset* f B
⟨*proof*⟩

lemma *mset-neg-image-zmset*: *mset-neg* M = {#} ⟹ *mset-neg* (*image-zmset* f M) = {#}
⟨*proof*⟩

lemma *nonneg-zcount-image-zmset*[*simp*]: ($\bigwedge t$. $0 \le$ *zcount* M t) ⟹ $0 \le$ *zcount* (*image-zmset* f M) t
⟨*proof*⟩

lemma *image-zmset-add-zmset*[*simp*]: *image-zmset* f (*add-zmset* t M) = *add-zmset* (f t) (*image-zmset* f M)
⟨*proof*⟩

lemma *pos-zcount-image-zmset*[*simp*]: ($\bigwedge t$. $0 \le$ *zcount* M t) ⟹ $0 <$ *zcount* M t ⟹ $0 <$ *zcount* (*image-zmset* f M) (f t)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *zcount-image-zmset*:
zcount (*image-zmset* f M) x = ($\sum y \in f$ −‘ {x} ∩ *set-zmset* M. *zcount* M y)
⟨*proof*⟩

lemma *zmset-empty-image-zmset-empty*: ($\bigwedge t$. *zcount* M t = 0) ⟹ *zcount* (*image-zmset* f M) t = 0
⟨*proof*⟩

lemma *in-image-zmset-in-zmset*: $t \in\#_z$ *image-zmset* f M ⟹ $\exists t.\ t \in\#_z M$
⟨*proof*⟩

lemma *zcount-image-zmset-zero*: ($\bigwedge m$. $m \in\#_z M$ ⟹ $f\ m \neq x$) ⟹ $x \notin\#_z$ *image-zmset* f M
⟨*proof*⟩

lemma *image-zmset-pre*: $t \in\#_z$ *image-zmset* f M ⟹ $\exists m.\ m \in\#_z M \wedge f\ m = t$
⟨*proof*⟩

lemma *pos-image-zmset-obtain-pre*:
($\bigwedge t$. $0 \le$ *zcount* M t) ⟹ $0 <$ *zcount* (*image-zmset* f M) t ⟹ $\exists m$. $0 <$ *zcount* M $m \wedge f\ m = t$
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *alw-relatesD*[*dest*]: *alw* (*relates P*) *s* $\Longrightarrow$ *P* (*shd s*) (*shd* (*stl s*))
⟨*proof*⟩

lemma *relatesI*[*intro*]: *P* (*shd s*) (*shd* (*stl s*)) $\Longrightarrow$ *relates P s*
⟨*proof*⟩

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

lemma *alw-relates*: *alw* (*relates P*) *s* $\longleftrightarrow$ *P* (*shd s*) (*shd* (*stl s*)) $\wedge$ *alw* (*relates P*) (*stl s*)
⟨*proof*⟩

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* $\Rightarrow$ *'t* :: *order* $\Rightarrow$ *bool* **where**
vacant-upto a t = ($\forall s.\ s \le t \longrightarrow$ *zcount a s* = *0*)

abbreviation *nonpos-upto* :: *'t delta-vec* $\Rightarrow$ *'t* :: *order* $\Rightarrow$ *bool* **where**
nonpos-upto a t $\equiv \forall s.\ s \le t \longrightarrow$ *zcount a s* $\le$ *0*

definition *supported-strong* :: *'t delta-vec* $\Rightarrow$ *'t* :: *order* $\Rightarrow$ *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* ⟷ (∀ *t*. *zcount a t* > *0* ⟶ *supported-strong a t*)
⟨*proof*⟩

definition *beta-upright* :: *'t* :: *order delta-vec* ⇒ *'t* :: *order delta-vec* ⇒ *bool* **where**
beta-upright va vb = (∀ *t*. *zcount va t* > *0* ⟶ (∃ *s*. *s* < *t* ∧ (*zcount va s* < *0* ∨ *zcount vb s* < *0*)))

lemma *beta-upright-alt*:
beta-upright va vb = (∀ *t*. *zcount va t* > *0* ⟶ (∃ *s*. *s* < *t* ∧ (*zcount va s* < *0* ∨ *zcount vb s* < *0*) ∧ *nonpos-upto va s*))
⟨*proof*⟩

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

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

definition *init-config* :: (*'p* :: *finite*, *'t* :: *order*) *configuration* ⇒ *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* ⇒ (*'p*, *'t*) *configuration* ⇒ *'p* ⇒ *'t count-vec* ⇒ *'t count-vec* ⇒ *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**
$next\text{-}sendupd'\ c0\ c1\ p\ tt =$
$(let\ \gamma = \{\#t \in\#_z\ c\text{-}temp\ c0\ p.\ t \in tt\#\}\ in$
$\gamma \neq 0$
$\wedge\ upright\ (c\text{-}temp\ c0\ p - \gamma)$
$\wedge\ c1 = c0(\!|c\text{-}msg := (c\text{-}msg\ c0)(p := \lambda q.\ c\text{-}msg\ c0\ p\ q\ @\ [\gamma]),$
$c\text{-}temp := (c\text{-}temp\ c0)(p := c\text{-}temp\ c0\ p - \gamma)|\!))$

abbreviation *next-sendupd* **where**
$next\text{-}sendupd\ s \equiv (\exists\, p\ tt.\ next\text{-}sendupd'\ (shd\ s)\ (shd\ (stl\ s))\ p\ tt)$

definition *next-recvupd'* **where**
$next\text{-}recvupd'\ c0\ c1\ p\ q =$
$(c\text{-}msg\ c0\ p\ q \neq []$
$\wedge\ c1 = c0(\!|c\text{-}msg := (c\text{-}msg\ c0)(p := (c\text{-}msg\ c0\ p)(q := tl\ (c\text{-}msg\ c0\ p\ q))),$
$c\text{-}glob := (c\text{-}glob\ c0)(q := c\text{-}glob\ c0\ q + hd\ (c\text{-}msg\ c0\ p\ q))|\!))$

abbreviation *next-recvupd* **where**
$next\text{-}recvupd\ s \equiv (\exists\, p\ q.\ next\text{-}recvupd'\ (shd\ s)\ (shd\ (stl\ s))\ p\ q)$

definition *next* **where**
$next\ s = (next\text{-}performop\ s \vee next\text{-}sendupd\ s \vee next\text{-}recvupd\ s \vee (shd\ (stl\ s) = shd\ s))$

definition *spec* :: $('p :: finite, 't :: order)\ computation \Rightarrow bool$ **where**
$spec\ s = (holds\ init\text{-}config\ s \wedge alw\ next\ s)$

abbreviation *GlobVacantUpto* **where**
$GlobVacantUpto\ c\ q\ t \equiv vacant\text{-}upto\ (c\text{-}glob\ c\ q)\ t$

abbreviation *NrecVacantUpto* **where**
$NrecVacantUpto\ c\ t \equiv vacant\text{-}upto\ (c\text{-}records\ c)\ t$

definition *SafeGlobVacantUptoImpliesStickyNrec* :: $('p :: finite, 't :: order)\ computation \Rightarrow bool$ **where**
$SafeGlobVacantUptoImpliesStickyNrec\ s =$
$(let\ c = shd\ s\ in\ \forall\, t\ q.\ GlobVacantUpto\ c\ q\ t \longrightarrow alw\ (holds\ (\lambda c.\ NrecVacantUpto\ c\ t))\ s)$

definition *SafeStickyNrecVacantUpto* :: $('p :: finite, 't :: order)\ computation \Rightarrow bool$ **where**
$SafeStickyNrecVacantUpto\ s =$
$(let\ c = shd\ s\ in\ \forall\, t.\ NrecVacantUpto\ c\ t \longrightarrow alw\ (holds\ (\lambda c.\ NrecVacantUpto\ c\ t))\ s)$

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *upright-vec-add*:
assumes *upright v1*
and *upright v2*
shows *upright* (*v1* + *v2*)
⟨*proof*⟩

lemma *next-InvTempUpright*: *holds InvTempUpright s* ⟹ *next s* ⟹ *nxt* (*holds InvTempUpright*) *s*
⟨*proof*⟩

lemma *alw-InvTempUpright*: *spec s* ⟹ *alw* (*holds InvTempUpright*) *s*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *init-InvIncomingInfoUpright*: *init-config c* ⟹ *InvIncomingInfoUpright c*
⟨*proof*⟩

lemma *next-InvIncomingInfoUpright*: *holds InvIncomingInfoUpright s* ⟹ *next s* ⟹ *nxt* (*holds InvIncomingInfoUpright*) *s*
⟨*proof*⟩

lemma *alw-InvIncomingInfoUpright*: *spec s* ⟹ *alw* (*holds InvIncomingInfoUpright*) *s*
⟨*proof*⟩

definition *GlobalIncomingInfo* :: ($'p$ :: *finite*, $'t$) *configuration* ⇒ *nat* ⇒ $'p$ ⇒ $'p$ ⇒ $'t$ *delta-vec* **where**
GlobalIncomingInfo c k p q = ($\sum p' \in$ *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 InvGlobalRecordCount s*
⟨*proof*⟩

lemma *if-eq-same*: (*if a* = *b then f b else f a*) = *f a*
⟨*proof*⟩

lemma *next-InvGlobalRecordCount*: *holds InvGlobalRecordCount s* ⟹ *next s* ⟹ *nxt* (*holds InvGlobalRecordCount*) *s*
⟨*proof*⟩

lemma *alw-InvGlobalRecordCount*: *spec s* ⟹ *alw* (*holds InvGlobalRecordCount*) *s*
⟨*proof*⟩

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* ($\sum x \in X$. *A x*)
⟨*proof*⟩

lemma *InvIncomingInfoUpright-imp-InvGlobalIncomingInfoUpright*: *holds InvIncomingInfoUpright s* ⟹ *holds InvGlobalIncomingInfoUpright s*
⟨*proof*⟩

lemma *alw-InvGlobalIncomingInfoUpright*: *spec s* ⟹ *alw* (*holds InvGlobalIncomingInfoUpright*) *s*
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *next-nrec-pos*: *holds nrec-pos s* $\Longrightarrow$ *next s* $\Longrightarrow$ *nxt* (*holds nrec-pos*) *s*
⟨*proof*⟩

lemma *alw-nrec-pos*: *spec s* $\Longrightarrow$ *alw* (*holds nrec-pos*) *s*
⟨*proof*⟩

lemma *next-performop-vacant*:
vacant-upto (*c-records* (*shd s*)) *t* $\Longrightarrow$ *next-performop s* $\Longrightarrow$ *vacant-upto* (*c-records* (*shd* (*stl s*))) *t*
⟨*proof*⟩

lemma *next-sendupd-vacant*:
vacant-upto (*c-records* (*shd s*)) *t* $\Longrightarrow$ *next-sendupd s* $\Longrightarrow$ *vacant-upto* (*c-records* (*shd* (*stl s*))) *t*
⟨*proof*⟩

lemma *next-recvupd-vacant*:
vacant-upto (*c-records* (*shd s*)) *t* $\Longrightarrow$ *next-recvupd s* $\Longrightarrow$ *vacant-upto* (*c-records* (*shd* (*stl s*))) *t*
⟨*proof*⟩

lemma *spec-imp-SafeStickyNrecVacantUpto-aux*: *alw next s* $\Longrightarrow$ *alw SafeStickyNrecVacantUpto s*
⟨*proof*⟩

lemma *spec-imp-SafeStickyNrecVacantUpto*: *spec s* $\Longrightarrow$ *alw SafeStickyNrecVacantUpto s*
⟨*proof*⟩

lemma *invs-imp-InvGlobVacantUptoImpliesNrec*:
assumes *holds InvGlobalIncomingInfoUpright s*
assumes *holds InvGlobalRecordCount s*
assumes *holds nrec-pos s*
shows *holds InvGlobVacantUptoImpliesNrec s*
⟨*proof*⟩

lemma *spec-imp-inv1*: *spec s* $\Longrightarrow$ *alw* (*holds InvGlobVacantUptoImpliesNrec*) *s*
⟨*proof*⟩

lemma *safe2-inv1-imp-safe*: *SafeStickyNrecVacantUpto s* $\Longrightarrow$ *holds InvGlobVacantUptoImpliesNrec s* $\Longrightarrow$ *SafeGlobVacantUptoImpliesStickyNrec s*
⟨*proof*⟩

lemma *spec-imp-safe*: *spec s* $\Longrightarrow$ *alw SafeGlobVacantUptoImpliesStickyNrec s*
⟨*proof*⟩

lemma *beta-upright-0*: *beta-upright 0 vb*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *betaupright-upright-vut*:
assumes *beta-upright va vb*
and *upright vb*
and *vacant-upto* (*va* + *vb*) *t*
shows *vacant-upto va t*
⟨*proof*⟩

lemma *beta-upright-add*:
assumes *upright vb*
and *upright vc*
and *beta-upright va vb*
shows *beta-upright va* (*vb* + *vc*)
⟨*proof*⟩

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*
⟨*proof*⟩

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* $\Longrightarrow$ *P*
and *shd* (*stl s*) = *shd s* $\Longrightarrow$ *P*
shows *P*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *alw-InvInfoAtBetaUpright-aux*: *alw* (*holds InvTempUpright*) *s* $\Longrightarrow$ *alw* (*holds InvIncomingInfoUpright*) *s* $\Longrightarrow$ *holds InvInfoAtBetaUpright s* $\Longrightarrow$ *alw next s* $\Longrightarrow$ *alw* (*holds InvInfoAtBetaUpright*) *s*
⟨*proof*⟩

lemma *alw-InvInfoAtBetaUpright*: *spec s* $\Longrightarrow$ *alw* (*holds InvInfoAtBetaUpright*) *s*
⟨*proof*⟩

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*: $\bigwedge$*T*. *finite T* $\Longrightarrow$ *T* ⊂ *S* $\Longrightarrow$ *P T* $\Longrightarrow$ ∃ *s*∈*S* − *T*. *P* (*insert s T*)
shows *P S*
⟨*proof*⟩

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 $\bigwedge$*x Z*. *A* (*f x*) $\Longrightarrow$ *B* (*sum f Z*) $\Longrightarrow$ *A* (*f x* + *sum f Z*)
and $\bigwedge$*x Z*. *B* (*f x*) $\Longrightarrow$ *A* (*sum f Z*) $\Longrightarrow$ *A* (*f x* + *sum f Z*)
shows *A* ($\sum$ *x*∈*X*. *f x*)
⟨*proof*⟩

lemma *invs-imp-InvGlobalInfoAtBetaUpright*:
assumes *holds InvInfoAtBetaUpright s*
and *holds InvGlobalIncomingInfoUpright s*
and *holds InvIncomingInfoUpright s*
shows *holds InvGlobalInfoAtBetaUpright s*

⟨*proof*⟩

lemma *alw-InvGlobalInfoAtBetaUpright*: *spec s* $\Longrightarrow$ *alw* (*holds InvGlobalInfoAtBetaUpright*) *s*
⟨*proof*⟩

definition *SafeStickyGlobVacantUpto* :: (*'p* :: *finite*, *'t* :: *order*) *computation* $\Rightarrow$ *bool* **where**
SafeStickyGlobVacantUpto s = ($\forall$ *q t*. *GlobVacantUpto* (*shd s*) *q t* $\longrightarrow$ *alw* (*holds* (λ*c*. *GlobVacantUpto c q t*)) *s*)

lemma *gvut1*:
GlobVacantUpto (*shd s*) *q t* $\Longrightarrow$ *next-performop s* $\Longrightarrow$ *GlobVacantUpto* (*shd* (*stl s*)) *q t*
⟨*proof*⟩

lemma *gvut2*:
GlobVacantUpto (*shd s*) *q t* $\Longrightarrow$ *next-sendupd s* $\Longrightarrow$ *GlobVacantUpto* (*shd* (*stl s*)) *q t*
⟨*proof*⟩

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*⟩

lemma *spec-imp-SafeStickyGlobVacantUpto-aux*:
assumes
alw (*holds* (λ*c*. *InvGlobVacantUptoImpliesNrec c*)) *s* **and**
alw (*holds* (λ*c*. *InvGlobalRecordCount c*)) *s* **and**
alw (*holds* (λ*c*. *InvGlobalIncomingInfoUpright c*)) *s* **and**
alw (*holds* (λ*c*. *InvGlobalInfoAtBetaUpright c*)) *s* **and**
alw next s
shows *alw SafeStickyGlobVacantUpto s*
⟨*proof*⟩

lemma *spec-imp-SafeStickyGlobVacantUpto*: *spec s* $\Longrightarrow$ *alw SafeStickyGlobVacantUpto s*
⟨*proof*⟩

definition *SafeGlobMono* **where**
SafeGlobMono c0 c1 = ($\forall$ *p t*. *GlobVacantUpto c0 p t* $\longrightarrow$ *GlobVacantUpto c1 p t*)

lemma *alw-SafeGlobMono*: *spec s* ⟹ *alw* (*relates SafeGlobMono*) *s*
⟨*proof*⟩

4 Exchange Protocol

4.1 Specification

record ('*p*, '*t*) *configuration* =
c-temp :: '*p* ⇒ '*t zmultiset*
c-msg :: '*p* ⇒ '*p* ⇒ '*t zmultiset list*
c-glob :: '*p* ⇒ '*t zmultiset*
c-caps :: '*p* ⇒ '*t zmultiset*
c-data-msg :: ('*p* × '*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* ≡ {# *t*. (*x*,*t*) $\in\#_z$ *M* #}

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

definition *nonpos-upto* :: '*a zmultiset* ⇒ '*a* ⇒ *bool* **where**
nonpos-upto a t = (∀ *s*. *s* ≤ *t* ⟶ *zcount a s* ≤ *0*)

definition *supported* :: '*a zmultiset* ⇒ '*a* ⇒ *bool* **where**
supported a t ≡ (∃ *s*. *s* < *t* ∧ *zcount a s* < *0*)

definition *supported-strong* :: '*a zmultiset* ⇒ '*a* ⇒ *bool* **where**
supported-strong a t ≡ (∃ *s*. *s* < *t* ∧ *zcount a s* < *0* ∧ *nonpos-upto a s*)

definition *justified* **where**
justified C M = (∀ *t*. *0* < *zcount M t* ⟶ *supported M t* ∨ (∃ *t'*<*t*. *0* < *zcount C t'*) ∨ *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*)

$\langle proof \rangle$

definition *justified-with* **where**
$justified\text{-}with\ C\ M\ N =$
$(\forall\, t.\ 0 < zcount\ M\ t \longrightarrow$
$(\exists\, s{<}t.\ (zcount\ M\ s < 0 \lor zcount\ N\ s < 0)) \lor$
$(\exists\, s{<}t.\ 0 < zcount\ C\ s) \lor$
$zcount\ (M{+}N)\ t < zcount\ C\ t)$

lemma *justified-with-alt*: $justified\text{-}with\ C\ M\ N =$
$(\forall\, t.\ 0 < zcount\ M\ t \longrightarrow$
$(\exists\, s{<}t.\ (zcount\ M\ s < 0 \lor zcount\ N\ s < 0) \land (\forall\, s'{<}s.\ zcount\ M\ s' \leq 0)) \lor$
$(\exists\, s{<}t.\ 0 < zcount\ C\ s) \lor$
$zcount\ (M{+}N)\ t < zcount\ C\ t)$
$\langle proof \rangle$

definition *PositiveImplies* **where**
$PositiveImplies\ v\ w \equiv \forall\, t.\ zcount\ v\ t > 0 \longrightarrow zcount\ w\ t > 0$

— A worker can mint capabilities greater or equal to any owned capability
definition *minting-self* **where**
$minting\text{-}self\ C\ M = (\forall\, t{\in}\#M.\ \exists\, t'{\leq}t.\ 0 < zcount\ C\ t')$

— Sending messages mints a capability at a strictly greater pointstamp
definition *minting-msg* **where**
$minting\text{-}msg\ C\ M = (\forall\, (p,t){\in}\#M.\ \exists\, t'{<}t.\ 0 < zcount\ C\ t')$

definition *records* **where**
$records\ c = (\sum p{\in}UNIV.\ c\text{-}caps\ c\ p) + timestamps\ (zmset\text{-}of\ (c\text{-}data\text{-}msg\ c))$

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

definition *IncomingInfo* :: $('p, 'a)\ configuration \Rightarrow nat \Rightarrow 'p \Rightarrow 'p \Rightarrow 'a\ zmultiset$ **where**
$IncomingInfo\ c\ k\ p\ q \equiv sum\text{-}list\ (drop\ k\ (c\text{-}msg\ c\ p\ q)) + c\text{-}temp\ c\ p$

definition *GlobalIncomingInfo* :: $('p :: finite, 'a)\ configuration \Rightarrow nat \Rightarrow 'p \Rightarrow 'p \Rightarrow 'a\ zmultiset$ **where**
$GlobalIncomingInfo\ c\ k\ p\ q \equiv \sum p' \in UNIV.\ IncomingInfo\ c\ (if\ p' = p\ then\ k\ else\ 0)\ p'\ q$

abbreviation *GlobalIncomingInfoAt* **where**
$GlobalIncomingInfoAt\ c\ q \equiv GlobalIncomingInfo\ c\ 0\ q\ q$

definition *init-config* :: $('p :: finite, 'a)\ configuration \Rightarrow bool$ **where**
$init\text{-}config\ c \equiv$

$(\forall p.\ c\text{-}temp\ c\ p = \{\#\}_z) \wedge$
$(\forall p1\ p2.\ c\text{-}msg\ c\ p1\ p2 = []) \wedge$
— Capabilities have non-negative multiplicities
$(\forall p\ t.\ 0 \leq zcount\ (c\text{-}caps\ c\ p)\ t) \wedge$
— The pointstamps in glob are exactly those in *records*
$(\forall p.\ c\text{-}glob\ c\ p = records\ c) \wedge$
— All capabilities are being tracked
$c\text{-}data\text{-}msg\ c = \{\#\}$

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

abbreviation *next-recvcap* **where**
$next\text{-}recvcap\ c0\ c1 \equiv \exists p\ t.\ next\text{-}recvcap'\ c0\ c1\ p\ t$

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 $\forall p\ t.\ \mathit{zcount}\ (\mathit{c\text{-}caps}\ c\ p)\ t \le \mathit{zcount}\ (\mathit{records}\ c)\ t$, our condition $\forall t.\ \mathit{int}\ (\mathit{count}\ \Delta\mathit{neg}\ t) \le \mathit{zcount}\ (\mathit{c\text{-}caps}\ c0\ p)\ t$ implies the one on *nrec* in Abadi's model.

Conditions in performop:

The performop transition takes three msets of pointstamps, $\Delta\mathit{neg}$, $\Delta\mathit{mint\text{-}msg}$, and $\Delta\mathit{mint\text{-}self}$ $\Delta\mathit{neg}$ contains dropped capabilities (a subset of *c-caps*) $\Delta\mathit{mint\text{-}msg}$ 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 $\Delta\mathit{mint\text{-}self}$ contains pointstamps minted and owned by worker p

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

The $\Delta\mathit{mint\text{-}msg} \neq \{\#\} \lor \mathit{zmset\text{-}of}\ \Delta\mathit{mint\text{-}self} - \mathit{zmset\text{-}of}\ \Delta\mathit{neg} \neq \{\#\}_z$ 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* ⇒

$'p \Rightarrow 'a\ set \Rightarrow bool$ **where**
  $next\text{-}sendupd'\ c0\ c1\ p\ tt =$
    $(let\ \gamma = \{\#t \in\#_z\ c\text{-}temp\ c0\ p.\ t \in tt\#\}\ in$
      $\gamma \neq 0$
    $\wedge\ justified\ (c\text{-}caps\ c0\ p)\ (c\text{-}temp\ c0\ p - \gamma)$
    $\wedge\ c1 = c0(\!|c\text{-}msg := (c\text{-}msg\ c0)(p := \lambda q.\ c\text{-}msg\ c0\ p\ q\ @\ [\gamma]),$
            $c\text{-}temp := (c\text{-}temp\ c0)(p := c\text{-}temp\ c0\ p - \gamma)|\!))$

abbreviation *next-sendupd* **where**
  $next\text{-}sendupd\ c0\ c1 \equiv (\exists\, p\ tt.\ next\text{-}sendupd'\ c0\ c1\ p\ tt)$

definition *next-recvupd'* :: $('p::finite, 'a)\ configuration \Rightarrow ('p, 'a)\ configuration \Rightarrow$ $'p \Rightarrow 'p \Rightarrow bool$ **where**
  $next\text{-}recvupd'\ c0\ c1\ p\ q \equiv$
      $c\text{-}msg\ c0\ p\ q \neq []$
    $\wedge\ c1 = c0(\!|c\text{-}msg := (c\text{-}msg\ c0)(p := (c\text{-}msg\ c0\ p)(q := tl\ (c\text{-}msg\ c0\ p\ q))),$
            $c\text{-}glob := (c\text{-}glob\ c0)(q := c\text{-}glob\ c0\ q + hd\ (c\text{-}msg\ c0\ p\ q))|\!)$

abbreviation *next-recvupd* **where**
  $next\text{-}recvupd\ c0\ c1 \equiv (\exists\, p\ q.\ next\text{-}recvupd'\ c0\ c1\ p\ q)$

definition *next'* **where**
  $next'\ c0\ c1 = (next\text{-}performop\ c0\ c1 \vee next\text{-}sendupd\ c0\ c1 \vee next\text{-}recvupd\ c0\ c1$ $\vee\ next\text{-}recvcap\ c0\ c1 \vee c1 = c0)$

abbreviation *next* **where**
  $next\ s \equiv next'\ (shd\ s)\ (shd\ (stl\ s))$

definition *spec* :: $('p :: finite, 'a)\ computation \Rightarrow bool$ **where**
  $spec\ s \equiv holds\ init\text{-}config\ s \wedge alw\ next\ s$

abbreviation *GlobVacantUpto* **where**
  $GlobVacantUpto\ c\ q\ t \equiv vacant\text{-}upto\ (c\text{-}glob\ c\ q)\ t$

abbreviation *GlobNonposUpto* **where**
  $GlobNonposUpto\ c\ q\ t \equiv nonpos\text{-}upto\ (c\text{-}glob\ c\ q)\ t$

abbreviation *RecordsVacantUpto* **where**
  $RecordsVacantUpto\ c\ t \equiv vacant\text{-}upto\ (records\ c)\ t$

definition *SafeGlobVacantUptoImpliesStickyNrec* :: $('p :: finite, 'a)\ computation$ $\Rightarrow bool$ **where**
  $SafeGlobVacantUptoImpliesStickyNrec\ s =$
    $(let\ c = shd\ s\ in\ \forall\, t\ q.\ GlobVacantUpto\ c\ q\ t \longrightarrow alw\ (holds\ (\lambda c.\ RecordsVacantUpto\ c\ t))\ s)$

4.2 Auxiliary Lemmas

lemma *finite-induct-select* [*consumes 1*, *case-names empty select*]:

assumes *finite S*
and *empty*: P {}
and *select*: $\bigwedge T.$ *finite* $T \Longrightarrow T \subset S \Longrightarrow P\ T \Longrightarrow \exists s{\in}S - T.\ P\ (insert\ s\ T)$
shows $P\ S$
⟨*proof*⟩

lemma *finite-induct-decompose-sum*:
fixes $f :: 'c \Rightarrow ('b :: comm\text{-}monoid\text{-}add)$
assumes *finite X*
and $x{\in}X$
and $A\ (f\ x)$
and $\forall Z.\ B\ (sum\ f\ Z)$
and $\bigwedge x\ Z.\ A\ (f\ x) \Longrightarrow B\ (sum\ f\ Z) \Longrightarrow A\ (f\ x + sum\ f\ Z)$
and $\bigwedge x\ Z.\ B\ (f\ x) \Longrightarrow A\ (sum\ f\ Z) \Longrightarrow A\ (f\ x + sum\ f\ Z)$
shows $A\ (\sum x{\in}X.\ f\ x)$
⟨*proof*⟩

lemma *minting-msg-add-records*: *minting-msg C1 M* $\Longrightarrow \forall t.\ 0 \leq$ *zcount C2 t* $\Longrightarrow$ *minting-msg* (*C1*+*C2*) *M*
⟨*proof*⟩

lemma *add-less*: $(a{::}int) < c \Longrightarrow b \leq 0 \Longrightarrow a + b < c$
⟨*proof*⟩

lemma *disj3-split*: $P \vee Q \vee R \Longrightarrow (P \Longrightarrow thesis) \Longrightarrow (\neg P \wedge Q \Longrightarrow thesis) \Longrightarrow (\neg P \Longrightarrow \neg Q \Longrightarrow R \Longrightarrow thesis) \Longrightarrow thesis$
⟨*proof*⟩

lemma *filter-zmset-conclude-predicate*: $0 <$ *zcount* {# $x \in\#_z M.\ P\ x$ #} $x \Longrightarrow 0 <$ *zcount M x* $\Longrightarrow P\ x$
⟨*proof*⟩

lemma *alw-holds2*: *alw* (*holds P*) *ss* = (*P* (*shd ss*) ∧ *alw* (*holds P*) (*stl ss*))
⟨*proof*⟩

lemma *zmset-of-remove1-mset*: $x \in\# M \Longrightarrow$ *zmset-of* (*remove1-mset x M*) = *zmset-of M* − $\{\#x\#\}_z$
⟨*proof*⟩

lemma *timestamps-zmset-of-pair-image*[*simp*]: *timestamps* (*zmset-of* {# (*c*,*t*). *t* ∈# *M* #}) = *zmset-of M*
⟨*proof*⟩

lemma *timestamps-image-zmset-fst*[*simp*]: *timestamps* {# (*f x*, *t*). (*x*, *t*) $\in\#_z$ *M* #} = *timestamps M*
⟨*proof*⟩

lemma *lift-invariant-to-spec*:
assumes ($\bigwedge c.$ *init-config c* $\Longrightarrow P\ c$)

and $(\bigwedge s.\ \mathit{holds}\ P\ s \Longrightarrow \mathit{next}\ s \Longrightarrow \mathit{nxt}\ (\mathit{holds}\ P)\ s)$
shows $\mathit{spec}\ s \Longrightarrow \mathit{alw}\ (\mathit{holds}\ P)\ s$
⟨*proof*⟩

lemma *timestamps-sum-distrib*[*simp*]: $(\sum p \in A.\ \mathit{timestamps}\ (f\ p)) = \mathit{timestamps}\ (\sum p \in A.\ f\ p)$
⟨*proof*⟩

lemma *timestamps-zmset-of*[*simp*]: *timestamps* (*zmset-of* M) = *zmset-of* {# t. $(p,t) \in\#\ M$ #}
⟨*proof*⟩

lemma *vacant-upto-add*: *vacant-upto* $a\ t \Longrightarrow$ *vacant-upto* $b\ t \Longrightarrow$ *vacant-upto* $(a+b)\ t$
⟨*proof*⟩

lemma *nonpos-upto-add*: *nonpos-upto* $a\ t \Longrightarrow$ *nonpos-upto* $b\ t \Longrightarrow$ *nonpos-upto* $(a+b)\ t$
⟨*proof*⟩

lemma *nonzero-lt-gtD*: $(n::\text{-}::\mathit{linorder}) \neq 0 \Longrightarrow 0 < n \vee n < 0$
⟨*proof*⟩

lemma *zero-lt-diff*: $(0::\mathit{int}) < a - b \Longrightarrow b \geq 0 \Longrightarrow 0 < a$
⟨*proof*⟩

lemma *zero-lt-add-disj*: $0 < (a::\mathit{int}) + b \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 < a \vee 0 < b$
⟨*proof*⟩

4.2.1 Transition lemmas

lemma *next-performopD*:
assumes *next-performop′ c0 c1 p* Δ*neg* Δ*mint-msg* Δ*mint-self*
shows
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*
minting-self (*c-caps c0 p*) Δ*mint-self*
minting-msg (*c-caps c0 p*) Δ*mint-msg*
c-temp c1 = (*c-temp c0*)(*p* := *c-temp c0 p* + (*timestamps* (*zmset-of* Δ*mint-msg*) + *zmset-of* Δ*mint-self* − *zmset-of* Δ*neg*))
c-msg c1 = *c-msg c0*
c-glob c1 = *c-glob c0*
c-data-msg c1 = *c-data-msg c0* + Δ*mint-msg*
c-caps c1 = (*c-caps c0*)(*p* := *c-caps c0 p* + (*zmset-of* Δ*mint-self* − *zmset-of* Δ*neg*))
⟨*proof*⟩

lemma *next-performop-complexD*:
assumes *next-performop′ c0 c1 p* Δ*neg* Δ*mint-msg* Δ*mint-self*

shows
$records\ c1 = records\ c0 + (timestamps\ (zmset\text{-}of\ \Delta mint\text{-}msg) + zmset\text{-}of\ \Delta mint\text{-}self - zmset\text{-}of\ \Delta neg)$
$GlobalIncomingInfoAt\ c1\ q = GlobalIncomingInfoAt\ c0\ q + (timestamps\ (zmset\text{-}of\ \Delta mint\text{-}msg) + zmset\text{-}of\ \Delta mint\text{-}self - zmset\text{-}of\ \Delta neg)$
$IncomingInfo\ c1\ k\ p'\ q = (if\ p' = p$
$then\ IncomingInfo\ c0\ k\ p'\ q + (timestamps\ (zmset\text{-}of\ \Delta mint\text{-}msg) + zmset\text{-}of\ \Delta mint\text{-}self - zmset\text{-}of\ \Delta neg)$
$else\ IncomingInfo\ c0\ k\ p'\ q)$
$\forall\, t'{<}t.\ zcount\ (c\text{-}caps\ c0\ p)\ t' = 0 \Longrightarrow zcount\ (timestamps\ (zmset\text{-}of\ \Delta mint\text{-}msg))\ t = 0$
$InfoAt\ c1\ k\ p'\ q = InfoAt\ c0\ k\ p'\ q$
$\langle proof \rangle$

lemma *next-sendupdD*:
assumes $next\text{-}sendupd'\ c0\ c1\ p\ tt$
shows
$\{\#t \in\#_z\ c\text{-}temp\ c0\ p.\ t \in tt\#\} \neq \{\#\}_z$
$justified\ (c\text{-}caps\ c0\ p)\ (c\text{-}temp\ c0\ p - \{\#t \in\#_z\ c\text{-}temp\ c0\ p.\ t \in tt\#\})$
$c\text{-}temp\ c1\ p' = (if\ p' = p\ then\ c\text{-}temp\ c0\ p - \{\#t \in\#_z\ c\text{-}temp\ c0\ p.\ t \in tt\#\}\ else\ c\text{-}temp\ c0\ p')$
$c\text{-}msg\ c1 = (\lambda p'\ q.\ if\ p' = p\ then\ c\text{-}msg\ c0\ p\ q\ @\ [\{\#t \in\#_z\ c\text{-}temp\ c0\ p.\ t \in tt\#\}]\ else\ c\text{-}msg\ c0\ p'\ q)$
$c\text{-}glob\ c1 = c\text{-}glob\ c0$
$c\text{-}caps\ c1 = c\text{-}caps\ c0$
$c\text{-}data\text{-}msg\ c1 = c\text{-}data\text{-}msg\ c0$
$\langle proof \rangle$

lemma *next-sendupd-complexD*:
assumes $next\text{-}sendupd'\ c0\ c1\ p\ tt$
shows
$records\ c1 = records\ c0$
$IncomingInfo\ c1\ 0 = IncomingInfo\ c0\ 0$
$IncomingInfo\ c1\ k\ p'\ q = (if\ p' = p \wedge length\ (c\text{-}msg\ c0\ p\ q) < k$
$then\ IncomingInfo\ c0\ k\ p'\ q - \{\#t \in\#_z\ c\text{-}temp\ c0\ p'.\ t \in tt\#\}$
$else\ IncomingInfo\ c0\ k\ p'\ q)$
$k \leq length\ (c\text{-}msg\ c0\ p\ q) \Longrightarrow IncomingInfo\ c1\ k\ p'\ q = IncomingInfo\ c0\ k\ p'\ q$
$length\ (c\text{-}msg\ c0\ p\ q) < k \Longrightarrow$
$IncomingInfo\ c1\ k\ p'\ q = (if\ p' = p$
$then\ IncomingInfo\ c0\ k\ p'\ q - \{\#t \in\#_z\ c\text{-}temp\ c0\ p'.\ t \in tt\#\}$
$else\ IncomingInfo\ c0\ k\ p'\ q)$
$GlobalIncomingInfoAt\ c1\ q = GlobalIncomingInfoAt\ c0\ q$
$InfoAt\ c1\ k\ p'\ q = (if\ p' = p \wedge k = length\ (c\text{-}msg\ c0\ p\ q)\ then\ \{\#t \in\#_z\ c\text{-}temp\ c0\ p'.\ t \in tt\#\}\ else\ InfoAt\ c0\ k\ p'\ q)$
$\langle proof \rangle$

lemma *next-recvupdD*:

assumes *next-recvupd′ c0 c1 p q*
shows
c-msg c0 p q $\neq$ []
c-temp c1 = *c-temp c0*
c-msg c1 = ($\lambda p'$ q'. *if* $p' = p \wedge q' = q$ *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*
⟨*proof*⟩

lemma *next-recvupd-complexD*:
assumes *next-recvupd′ c0 c1 p q*
shows
records c1 = *records c0*
IncomingInfo c1 0 p' q' = (*if* $p' = p \wedge q' = q$ *then IncomingInfo c0 0* p' q' − *hd* (*c-msg c0 p q*) *else IncomingInfo c0 0* p' q')
IncomingInfo c1 k p' q' = (*if* $p' = p \wedge q' = q$
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* $p' = p \wedge q' = q$ *then InfoAt c0* (*k*+*1*) *p q else InfoAt c0 k* p' q')
⟨*proof*⟩

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*)#}
⟨*proof*⟩

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*⟩

lemma *ex-next-recvupd*:
assumes *c-msg c0 p q* $\neq$ []
shows ∃ *c1*. *next-recvupd′ c0 c1 p q*

⟨*proof*⟩

4.2.2 Facts about *justified*'ness

lemma *justified-empty*[*simp*]: *justified* $\{\#\}_z$ $\{\#\}_z$
⟨*proof*⟩

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

lemma *justified-leastI*:
assumes $\forall t.\ 0 < zcount\ M\ t \longrightarrow (\forall t'<t.\ zcount\ M\ t' \leq 0) \longrightarrow supported\text{-}strong\ M\ t \vee (\exists t'<t.\ 0 < zcount\ C\ t') \vee (zcount\ M\ t < zcount\ C\ t)$
shows *justified C M*
⟨*proof*⟩

lemma *justified-add*:
assumes *justified C1 M1*
and *justified C2 M2*
and $\forall t.\ 0 \leq zcount\ C1\ t$
and $\forall t.\ 0 \leq zcount\ C2\ t$
shows *justified* (*C1*+*C2*) (*M1*+*M2*)
⟨*proof*⟩

lemma *justified-sum*:
assumes $\forall p{\in}P.\ justified\ (f\ p)\ (g\ p)$
and $\forall p{\in}P.\ \forall t.\ 0 \leq zcount\ (f\ p)\ t$
shows $justified\ (\sum p{\in}P.\ f\ p)\ (\sum p{\in}P.\ g\ p)$
⟨*proof*⟩

lemma *justified-add-records*:
assumes *justified C M*
and $\forall t.\ 0 \leq zcount\ C'\ t$
shows $justified\ (C+C')\ M$
⟨*proof*⟩

lemma *justified-add-zmset-records*:
assumes *justified C M*
shows *justified* (*add-zmset t C*) *M*
⟨*proof*⟩

lemma *justified-diff*:
assumes *justified C M*
and $\forall t.\ 0 \leq zcount\ C\ t$
and $\forall t.\ count\ \Delta\ t \leq zcount\ C\ t$
shows $justified\ (C - zmset\text{-}of\ \Delta)\ (M - zmset\text{-}of\ \Delta)$
⟨*proof*⟩

lemma *justified-add-msg-delta*:
assumes *justified C M*
and $minting\text{-}msg\ C\ \Delta$

and $\forall t.\ 0 \leq zcount\ C\ t$
shows $justified\ C\ (M + timestamps\ (zmset\text{-}of\ \Delta))$
⟨*proof*⟩

lemma *justified-add-same*:
assumes $justified\ C\ M$
and $minting\text{-}self\ C\ \Delta$
and $\forall t.\ 0 \leq zcount\ C\ t$
shows $justified\ (C + zmset\text{-}of\ \Delta)\ (M + zmset\text{-}of\ \Delta)$
⟨*proof*⟩

4.2.3 Facts about *justified-with*'ness

lemma *justified-with-add-records*:
assumes $justified\text{-}with\ C1\ M\ N$
and $\forall t.\ 0 \leq zcount\ C2\ t$
shows $justified\text{-}with\ (C1{+}C2)\ M\ N$
⟨*proof*⟩

lemma *justified-with-leastI*:
assumes
$(\forall t.\ 0 < zcount\ M\ t \longrightarrow (\forall t'{<}t.\ zcount\ M\ t' \leq 0) \longrightarrow$
$(\exists s{<}t.\ (zcount\ M\ s < 0 \lor zcount\ N\ s < 0) \land (\forall s'{<}s.\ zcount\ M\ s' \leq 0)) \lor$
$(\exists s{<}t.\ 0 < zcount\ C\ s) \lor$
$zcount\ (M{+}N)\ t < zcount\ C\ t)$
shows $justified\text{-}with\ C\ M\ N$
⟨*proof*⟩

lemma *justified-with-add*:
assumes $justified\text{-}with\ C1\ M\ N1$
and $justified\ C1\ N1$
and $justified\ C2\ N2$
and $\forall t.\ 0 \leq zcount\ C1\ t$
and $\forall t.\ 0 \leq zcount\ C2\ t$
shows $justified\text{-}with\ (C1{+}C2)\ M\ (N1{+}N2)$
⟨*proof*⟩

lemma *justified-with-sum'*:
assumes $finite\ X\ X{\neq}\{\}$
and $\forall x{\in}X.\ justified\text{-}with\ (C\ x)\ M\ (N\ x)$
and $\forall x{\in}X.\ justified\ (C\ x)\ (N\ x)$
and $\forall x{\in}X.\ \forall t.\ 0 \leq zcount\ (C\ x)\ t$
shows $justified\text{-}with\ (\sum x{\in}X.\ C\ x)\ M\ (\sum x{\in}X.\ N\ x)$
⟨*proof*⟩

lemma *justified-with-sum*:
assumes $finite\ X\ X{\neq}\{\}$
and $x \in X$
and $justified\text{-}with\ (C\ x)\ M\ (N\ x)$

and $\forall x \in X.$ *justified* $(C\ x)\ (N\ x)$
and $\forall x \in X.\ \forall t.\ 0 \le$ *zcount* $(C\ x)\ t$
shows *justified-with* $(\sum x \in X.\ C\ x)\ M\ (\sum x \in X.\ N\ x)$
⟨*proof*⟩

lemma *justified-with-add-same*:
assumes *justified-with* $C\ M\ N$
and $\forall t.\ 0 \le$ *zcount* $C\ t$
shows *justified-with* $(C +$ *zmset-of* $\Delta)\ M\ (N +$ *zmset-of* $\Delta)$
⟨*proof*⟩

lemma *justified-with-add-msg-delta*:
assumes *justified-with* $C\ M\ N$
and *minting-msg* $C\ \Delta$
and $\forall t.\ 0 \le$ *zcount* $C\ t$
shows *justified-with* $C\ M\ (N +$ *timestamps* (*zmset-of* Δ))
⟨*proof*⟩

lemma *justified-with-diff*:
assumes *justified-with* $C\ M\ N$
and $\forall t.\ 0 \le$ *zcount* $C\ t$
and $\forall t.$ *count* $\Delta\ t \le$ *zcount* $C\ t$
and *justified* $C\ N$
shows *justified-with* $(C -$ *zmset-of* $\Delta)\ M\ (N -$ *zmset-of* $\Delta)$
⟨*proof*⟩

lemma *PositiveImplies-justified-with*:
assumes *justified* $C\ (M+N)$
and *PositiveImplies* $M\ (M+N)$
shows *justified-with* $C\ M\ N$
⟨*proof*⟩

lemma *justified-with-add-zmset*[*simp*]:
assumes *justified-with* $C\ M\ N$
shows *justified-with* (*add-zmset* $c\ C$) $M\ N$
⟨*proof*⟩

lemma *next-performop′-preserves-justified-with*:
assumes *justified-with* (*c-caps c0 p*) $M\ N$
and *next-performop′ c0 c1 p* Δ*neg* Δ*mint-msg* Δ*mint-self*
and $\forall t.\ 0 \le$ *zcount* (*c-caps c0 p*) t
and *justified* (*c-caps c0 p*) N
shows *justified-with* (*c-caps c0 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*)
⟨*proof*⟩

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 *GlobalIncomingInfoAt 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*
⟨*proof*⟩

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*⟩

lemma *sendupd-preserves-InvRecordCount*:
assumes *InvRecordCount c0*
and *next-sendupd′ c0 c1 p tt*
shows *InvRecordCount c1*
⟨*proof*⟩

lemma *recvupd-preserves-InvRecordCount*:
assumes *InvRecordCount c0*
and *next-recvupd′ c0 c1 p q*
shows *InvRecordCount c1*
⟨*proof*⟩

lemma *recvcap-preserves-InvRecordCount*:
assumes *InvRecordCount c0*
and *next-recvcap′ c0 c1 p t*
shows *InvRecordCount c1*
⟨*proof*⟩

lemma *next-preserves-InvRecordCount*: *InvRecordCount c0* ⟹ *next′ c0 c1* ⟹ *InvRecordCount c1*
⟨*proof*⟩

lemma *alw-InvRecordCount*: *spec s* ⟹ *alw* (*holds InvRecordCount*) *s*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *performop-preserves-InvCapsNonneg*:
assumes *InvCapsNonneg c0*
and *next-performop′ c0 c1 p* Δ_m Δ_{p1} Δ_{p2}
shows *InvCapsNonneg c1*
⟨*proof*⟩

lemma *sendupd-performs-InvCapsNonneg*:
assumes *InvCapsNonneg c0*
and *next-sendupd′ c0 c1 p tt*
shows *InvCapsNonneg c1*
⟨*proof*⟩

lemma *recvupd-preserves-InvCapsNonneg*:
assumes *InvCapsNonneg c0*
and *next-recvupd′ c0 c1 p q*
shows *InvCapsNonneg c1*
⟨*proof*⟩

lemma *recvcap-preserves-InvCapsNonneg*:
assumes *InvCapsNonneg c0*
and *next-recvcap′ c0 c1 p t*
shows *InvCapsNonneg c1*
⟨*proof*⟩

lemma *next-preserves-InvCapsNonneg*: *holds InvCapsNonneg s* ⟹ *next s* ⟹ *nxt* (*holds InvCapsNonneg*) *s*
⟨*proof*⟩

lemma *alw-InvCapsNonneg*: *spec s* ⟹ *alw* (*holds InvCapsNonneg*) *s*
⟨*proof*⟩

lemma *alw-InvRecordsNonneg*: *spec s* ⟹ *alw* (*holds InvRecordsNonneg*) *s*
⟨*proof*⟩

4.3.3 Resulting lemmas

lemma *pos-caps-pos-records*:
assumes *InvCapsNonneg c*
shows *0* < *zcount* (*c-caps c p*) *x* ⟹ *0* < *zcount* (*records c*) *x*
⟨*proof*⟩

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*⟩

lemma *next'-preserves-RecordsVacantUpto*:
fixes *c0* :: ('*p*::*finite*, '*a*) *configuration*
shows *InvCapsNonneg c0* ⟹ *InvRecordsNonneg c1* ⟹ *RecordsVacantUpto c0 t* ⟹ *next' c0 c1* ⟹ *RecordsVacantUpto c1 t*
⟨*proof*⟩

lemma *alw-next-implies-alw-SafeRecordsMono*:
alw next s ⟹ *alw* (*holds InvCapsNonneg*) *s* ⟹ *alw* (*holds InvRecordsNonneg*) *s* ⟹ *alw SafeRecordsMono s*
⟨*proof*⟩

lemma *alw-SafeRecordsMono*: *spec s* ⟹ *alw SafeRecordsMono s*
⟨*proof*⟩

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*⟩

lemma *InvJustifiedII-implies-InvJustifiedGII*:
assumes *InvJustifiedII c*
and *InvCapsNonneg c*
shows *InvJustifiedGII c*
⟨*proof*⟩

lemma *init-config-implies-InvJustifiedII*: *init-config c* ⟹ *InvJustifiedII c*
⟨*proof*⟩

lemma *performop-preserves-InvJustifiedII*:
assumes *InvJustifiedII c0*
and *next-performop′ c0 c1 p* Δ*neg* Δ*mint-msg* Δ*mint-self*
and *InvCapsNonneg c0*
shows *InvJustifiedII c1*
⟨*proof*⟩

lemma *sendupd-preserves-InvJustifiedII*:
assumes *InvJustifiedII c0*
and *next-sendupd′ c0 c1 p tt*
shows *InvJustifiedII c1*
⟨*proof*⟩

lemma *recvupd-preserves-InvJustifiedII*:
assumes *InvJustifiedII c0*
and *next-recvupd′ c0 c1 p q*
shows *InvJustifiedII c1*
⟨*proof*⟩

lemma *recvcap-preserves-InvJustifiedII*:
assumes *InvJustifiedII c0*
and *next-recvcap′ c0 c1 p t*
shows *InvJustifiedII c1*
⟨*proof*⟩

lemma *next′-preserves-InvJustifiedII*:
InvCapsNonneg c0 ⟹ *InvJustifiedII c0* ⟹ *next′ c0 c1* ⟹ *InvJustifiedII c1*
⟨*proof*⟩

lemma *alw-InvJustifiedII*: *spec s* ⟹ *alw* (*holds InvJustifiedII*) *s*
⟨*proof*⟩

lemma *alw-InvJustifiedGII*: *spec s* ⟹ *alw* (*holds InvJustifiedGII*) *s*
⟨*proof*⟩

4.3.6 InvTempJustified

definition *InvTempJustified* **where**
InvTempJustified c = (∀ *p*. *justified* (*c-caps c p*) (*c-temp c p*))

lemma *init-config-implies-InvTempJustified*: *init-config c* ⟹ *InvTempJustified c*
⟨*proof*⟩

lemma *recvcap-preserves-InvTempJustified*:
assumes *InvTempJustified c0*
and *next-recvcap′ c0 c1 p t*
shows *InvTempJustified c1*
⟨*proof*⟩

lemma *recvupd-preserves-InvTempJustified*:
assumes *InvTempJustified c0*
and *next-recvupd′ c0 c1 p t*
shows *InvTempJustified c1*
⟨*proof*⟩

lemma *sendupd-preserves-InvTempJustified*:
assumes *InvTempJustified c0*
and *next-sendupd′ c0 c1 p tt*
shows *InvTempJustified c1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *next′-preserves-InvTempJustified*:
InvCapsNonneg c0 ⟹ *InvTempJustified c0* ⟹ *next′ c0 c1* ⟹ *InvTempJustified c1*
⟨*proof*⟩

lemma *alw-InvTempJustified*: *spec s* ⟹ *alw* (*holds InvTempJustified*) *s*
⟨*proof*⟩

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* ⇒ *bool* **where**
InvGlobNonposImpRecordsNonpos c = (∀ *t q*. *nonpos-upto* (*c-glob c q*) *t* ⟶ *nonpos-upto* (*records c*) *t*)

definition *InvGlobVacantImpRecordsVacant* :: ('*p* :: *finite*, '*a*) *configuration* ⇒ *bool*
where
InvGlobVacantImpRecordsVacant c = (∀ *t q*. *GlobVacantUpto c q t* ⟶ *RecordsVacantUpto c t*)

lemma *invs-imp-InvGlobNonposImpRecordsNonpos*:
assumes *InvJustifiedGII c*
and *InvRecordCount c*
and *InvRecordsNonneg c*
shows *InvGlobNonposImpRecordsNonpos c*
⟨*proof*⟩

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*⟩

lemma *alw-InvGlobNonposImpRecordsNonpos*: *spec s* ⟹ *alw* (*holds InvGlobNonposImpRecordsNonpos*) *s*
⟨*proof*⟩

lemma *alw-InvGlobVacantImpRecordsVacant*: *spec s* ⟹ *alw* (*holds InvGlobVacantImpRecordsVacant*) *s*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *alw-SafeGlobVacantUptoImpliesStickyNrec*:
spec s ⟹ *alw SafeGlobVacantUptoImpliesStickyNrec s*
⟨*proof*⟩

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*⟩

lemma *alw-InvGlobNonposEqVacant*: *spec s* ⟹ *alw* (*holds InvGlobNonposEqVacant*) *s*
⟨*proof*⟩

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* ⟹ *InvInfoJustifiedWithII c*
⟨*proof*⟩

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*; ∀ *t*. *0* ≤ *zcount* (*c-caps ?c0.0 ?p*) *t*; *justified* (*c-caps ?c0.0 ?p*) *?N*⟧ ⟹ *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*
⟨*proof*⟩

lemma *sendupd-preserves-InvInfoJustifiedWithII*:
assumes *InvInfoJustifiedWithII c0*
and *next-sendupd′ c0 c1 p′ tt*
and *InvTempJustified c0*
shows *InvInfoJustifiedWithII c1*
⟨*proof*⟩

lemma *recvupd-preserves-InvInfoJustifiedWithII*:
assumes *InvInfoJustifiedWithII c0*
and *next-recvupd′ c0 c1 p q*
shows *InvInfoJustifiedWithII c1*
⟨*proof*⟩

lemma *recvcap-preserves-InvInfoJustifiedWithII*:
assumes *InvInfoJustifiedWithII c0*
and *next-recvcap′ c0 c1 p t*
shows *InvInfoJustifiedWithII c1*
⟨*proof*⟩

lemma *invs-imp-InvInfoJustifiedWithGII*:
assumes *InvInfoJustifiedWithII c*
and *InvJustifiedII c*
and *InvCapsNonneg c*
shows *InvInfoJustifiedWithGII c*
⟨*proof*⟩

lemma *next′-preserves-InvInfoJustifiedWithII*:
assumes *InvInfoJustifiedWithII c0*
and *next′ c0 c1*
and *InvCapsNonneg c0*
and *InvJustifiedII c0*
and *InvTempJustified c0*
shows *InvInfoJustifiedWithII c1*
⟨*proof*⟩

lemma *alw-InvInfoJustifiedWithII*: $spec\ s \Longrightarrow alw\ (holds\ InvInfoJustifiedWithII)\ s$
⟨*proof*⟩

lemma *alw-InvInfoJustifiedWithGII*: $spec\ s \Longrightarrow alw\ (holds\ InvInfoJustifiedWithGII)\ s$
⟨*proof*⟩

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**
$SafeGlobMono\ c0\ c1 = (\forall\, p\ t.\ GlobVacantUpto\ c0\ p\ t \longrightarrow GlobVacantUpto\ c1\ p\ t)$

definition *InvMsgInGlob* **where**
$InvMsgInGlob\ c = (\forall\, p\ q\ t.\ c\text{-}msg\ c\ p\ q \neq [] \longrightarrow t \in\#_z hd\ (c\text{-}msg\ c\ p\ q) \longrightarrow (\exists\, t' \le t.\ 0 < zcount\ (c\text{-}glob\ c\ q)\ t'))$

lemma *not-InvMsgInGlob-imp-not-SafeGlobMono*:
assumes ¬ *InvMsgInGlob c0*
and *InvGlobNonposEqVacant c0*
shows ∃ *c1. next-recvupd c0 c1* ∧ ¬ *SafeGlobMono c0 c1*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *next′-imp-SafeGlobMono*:
assumes *next′ c0 c1*
and *InvInfoJustifiedWithGII c0*
and *InvGlobNonposEqVacant c1*
and *InvGlobVacantImpRecordsVacant c0*
and *InvRecordCount c0*
shows *SafeGlobMono c0 c1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *alw-SafeGlobMono*: *spec s* $\Longrightarrow$ *alw* (*relates SafeGlobMono*) *s*
⟨*proof*⟩

lemma *alw-InvMsgInGlob*: *spec s* $\Longrightarrow$ *alw* (*holds InvMsgInGlob*) *s*
⟨*proof*⟩

lemma *SafeGlobMono-preserves-vacant*:
assumes $\forall t' \le t.$ *zcount* (*c-glob c0 q*) $t' = 0$
and (λ*c0 c1. SafeGlobMono c0 c1*)** *c0 c1*
shows $\forall t' \le t.$ *zcount* (*c-glob c1 q*) $t' = 0$
⟨*proof*⟩

lemma *rtranclp-all-imp-rel*: $r^{**}\ x\ y \Longrightarrow \forall\, a\ b.\ r\ a\ b \longrightarrow r'\ a\ b \Longrightarrow r'^{**}\ x\ y$
⟨*proof*⟩

lemma *rtranclp-rel-and-invar*: $r^{**}\ x\ y \Longrightarrow Q\ x \Longrightarrow \forall\, a\ b.\ Q\ a \wedge r\ a\ b \longrightarrow P\ a\ b \wedge Q\ b \Longrightarrow (\lambda x\ y.\ P\ x\ y \wedge Q\ y)^{**}\ x\ y$
⟨*proof*⟩

lemma *rtranclp-invar-conclude-last*: $(\lambda x\ y.\ P\ x\ y \wedge Q\ y)^{**}\ x\ y \Longrightarrow Q\ x \Longrightarrow Q\ y$
⟨*proof*⟩

lemma *InvCapsNonneg-imp-InvRecordsNonneg*: *InvCapsNonneg c* $\Longrightarrow$ *InvRecordsNonneg c*
⟨*proof*⟩

lemma *invs-imp-msg-in-glob*:
fixes *c* :: (*'p*::*finite*, *'a*) *configuration*
assumes $M \in$ *set* (*c-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 $\exists t' \le t.\ 0 <$ *zcount* (*c-glob c q*) t'
⟨*proof*⟩

lemma *alw-msg-glob*: *spec s* $\Longrightarrow$
alw (*holds* ($\lambda c.\ \forall p\ q\ t.\ (\exists M \in$ *set* (*c-msg c p q*). $t \in\#_z M) \longrightarrow (\exists t' \le t.\ 0 <$ *zcount* (*c-glob c q*) t'))) *s*
⟨*proof*⟩

end

5 Antichains

definition *incomparable* **where**
  $incomparable\ A = (\forall x \in A.\ \forall y \in A.\ x \neq y \longrightarrow \neg\ x < y \wedge \neg\ y < x)$

lemma *incomparable-empty*[*simp*, *intro*]: *incomparable* {}
  ⟨*proof*⟩

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

setup-lifting *type-definition-antichain*

lift-definition *member-antichain* :: *'a* :: *order* ⇒ *'a antichain* ⇒ *bool* ((-/ $\in_A$ -) [*51*, *51*] *50*) **is** *Set.member* ⟨*proof*⟩

abbreviation *not-member-antichain* :: *'a* :: *order* ⇒ *'a antichain* ⇒ *bool* ((-/ $\notin_A$ -) [*51*, *51*] *50*) **where**
  $x \notin_A A \equiv \neg\ x \in_A A$

lift-definition *empty-antichain* :: *'a* :: *order antichain* ($\{\}_A$) **is** {} ⟨*proof*⟩

lemma *mem-antichain-nonempty*[*simp*]: $s \in_A A \Longrightarrow A \neq \{\}_A$
  ⟨*proof*⟩

definition *minimal-antichain* $A = \{x \in A.\ \neg(\exists y \in A.\ y < x)\}$

lemma *in-minimal-antichain*: $x \in$ *minimal-antichain* $A \longleftrightarrow x \in A \wedge \neg(\exists y \in A.\ y < x)$
  ⟨*proof*⟩

lemma *in-antichain-minimal-antichain*[*simp*]: *finite M* $\Longrightarrow x \in_A$ *antichain* (*minimal-antichain M*) $\longleftrightarrow x \in$ *minimal-antichain M*
  ⟨*proof*⟩

lemma *incomparable-minimal-antichain*[*simp*]: *incomparable* (*minimal-antichain A*)
  ⟨*proof*⟩

lemma *finite-minimal-antichain*[*simp*]: *finite A* ⟹ *finite* (*minimal-antichain A*)
  ⟨*proof*⟩

lemma *finite-set-antichain*[*simp*, *intro*]: *finite* (*set-antichain A*)
  ⟨*proof*⟩

lemma *minimal-antichain-subset*: *minimal-antichain* $A \subseteq A$

⟨*proof*⟩

lift-definition *frontier* :: *'t* :: *order zmultiset* ⇒ *'t antichain* **is** $\lambda M.$ *minimal-antichain* $\{t.$ *zcount* $M\ t > 0\}$
⟨*proof*⟩

lemma *member-frontier-pos-zmset*: $t \in_A$ *frontier* $M \Longrightarrow 0 <$ *zcount* $M\ t$
⟨*proof*⟩

lemma *frontier-comparable-False*[*simp*]: $x \in_A$ *frontier* $M \Longrightarrow y \in_A$ *frontier* $M \Longrightarrow x < y \Longrightarrow$ *False*
⟨*proof*⟩

lemma *minimal-antichain-idempotent*[*simp*]: *minimal-antichain* (*minimal-antichain* A) = *minimal-antichain* A
⟨*proof*⟩

instantiation *antichain* :: (*order*) *minus* **begin**
lift-definition *minus-antichain* :: *'a antichain* ⇒ *'a antichain* ⇒ *'a antichain* **is** $(-)$
⟨*proof*⟩
instance ⟨*proof*⟩
end

instantiation *antichain* :: (*order*) *plus* **begin**
lift-definition *plus-antichain* :: *'a antichain* ⇒ *'a antichain* ⇒ *'a antichain* **is** $\lambda M\ N.$ *minimal-antichain* $(M \cup N)$
⟨*proof*⟩
instance ⟨*proof*⟩
end

lemma *antichain-add-commute*: (M :: *'a* :: *order antichain*) + N = N + M
⟨*proof*⟩

lift-definition *filter-antichain* :: (*'a* :: *order* ⇒ *bool*) ⇒ *'a antichain* ⇒ *'a antichain* **is** *Set.filter*
⟨*proof*⟩

syntax (*ASCII*)
-ACCollect :: *pttrn* ⇒ *'a* :: *order antichain* ⇒ *bool* ⇒ *'a antichain* ((*1*{- $:_A$ -./ -}))
syntax
-ACCollect :: *pttrn* ⇒ *'a* :: *order antichain* ⇒ *bool* ⇒ *'a antichain* ((*1*{- $\in_A$ -./ -}))
translations
$\{x \in_A M.\ P\}$ == *CONST filter-antichain* $(\lambda x.\ P)\ M$

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

lemma *minimal-antichain-empty*[*simp*]: *minimal-antichain* {} = {}
⟨*proof*⟩

lemma *minimal-antichain-singleton*[*simp*]: *minimal-antichain* {*x*::- ::*order*} = {*x*}
⟨*proof*⟩

lemma *minimal-antichain-nonempty*:
finite $A \Longrightarrow$ (*t*::-::*order*) $\in A \Longrightarrow$ *minimal-antichain* $A \neq \{\}$
⟨*proof*⟩

lemma *minimal-antichain-member*:
finite $A \Longrightarrow$ (*t*::-::*order*) $\in A \Longrightarrow \exists\, t'.\ t' \in$ *minimal-antichain* $A \wedge t' \leq t$
⟨*proof*⟩

lemma *minimal-antichain-union*: *minimal-antichain* ((*A*::(- :: *order*) *set*) $\cup$ *B*) $\subseteq$ *minimal-antichain* (*minimal-antichain* *A* $\cup$ *minimal-antichain* *B*)
⟨*proof*⟩

lemma *ac-Diff-iff*: $c \in_A A - B \longleftrightarrow c \in_A A \wedge c \notin_A B$
⟨*proof*⟩

lemma *ac-DiffD2*: $c \in_A A - B \Longrightarrow c \in_A B \Longrightarrow P$
⟨*proof*⟩

lemma *ac-notin-Diff*: $\neg\ x \in_A A - B \Longrightarrow \neg\ x \in_A A \vee x \in_A B$
⟨*proof*⟩

lemma *ac-eq-iff*: $A = B \longleftrightarrow (\forall\, x.\ x \in_A A \longleftrightarrow x \in_A B)$
⟨*proof*⟩

lemma *antichain-obtain-foundation*:
assumes $t \in_A M$
obtains *s* **where** $s \in_A M \wedge s \leq t \wedge (\forall\, u.\ u \in_A M \longrightarrow \neg\ u < s)$
⟨*proof*⟩

lemma *set-antichain1*[*simp*]: $x \in$ *set-antichain* $X \Longrightarrow x \in_A X$
⟨*proof*⟩

lemma *set-antichain2*[*simp*]: $x \in_A X \Longrightarrow x \in$ *set-antichain* X
⟨*proof*⟩

6 Multigraphs with Partially Ordered Weights

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

abbreviation (*input*) *LBL* **where**
LBL ≡ λ(*s*, *l*, *t*). *l*

abbreviation (*input*) *TO* **where**
TO ≡ λ(*s*, *l*, *t*). *t*

notation *subseq* (**infix** ⪯ *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'*
⟨*proof*⟩

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* $\in_A$ *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*)
⟨*proof*⟩

lemma *path-take-from*: *path l1 l2 xs* ⟹ *m* < *length xs* ⟹ *FROM* (*xs* ! *m*) = *l2'* ⟹ *path l1 l2'* (*take m xs*)
⟨*proof*⟩

lemma *path-take-to*: *path l1 l2 xs* ⟹ *m* < *length xs* ⟹ *TO* (*xs* ! *m*) = *l2'* ⟹ *path l1 l2'* (*take* (*m*+*1*) *xs*)
⟨*proof*⟩

lemma *path-determines-loc*: *path l1 l2 xs* ⟹ *path l1 l3 xs* ⟹ *l2* = *l3*
⟨*proof*⟩

lemma *path-first-loc*: *path loc loc' xs* ⟹ *xs* ≠ [] ⟹ *FROM* (*xs* ! *0*) = *loc*
⟨*proof*⟩

lemma *path-to-eq-from*: *path loc1 loc2 xs* ⟹ *i* + *1* < *length xs* ⟹ *FROM* (*xs* ! (*i*+*1*)) = *TO* (*xs* ! *i*)
⟨*proof*⟩

lemma *path-singleton*[*intro*, *simp*]: *s* $\in_A$ *weights l1 l2* ⟹ *path l1 l2* [(*l1*,*s*,*l2*)]

⟨*proof*⟩

lemma *path-appendE*: $path\ l1\ l3\ (xs\ @\ ys) \Longrightarrow \exists\, l2.\ path\ l2\ l3\ ys \wedge path\ l1\ l2\ xs$
⟨*proof*⟩

lemma *path-replace-prefix*:
$path\ l1\ l3\ (xs\ @\ zs) \Longrightarrow path\ l1\ l2\ ys \Longrightarrow path\ l1\ l2\ xs \Longrightarrow path\ l1\ l3\ (ys\ @\ zs)$
⟨*proof*⟩

lemma *drop-subseq*: $n \leq length\ xs \Longrightarrow drop\ n\ xs \preceq xs$
⟨*proof*⟩

lemma *take-subseq*[*simp*, *intro*]: $take\ n\ xs \preceq xs$
⟨*proof*⟩

lemma *map-take-subseq*[*simp*, *intro*]: $map\ f\ (take\ n\ xs) \preceq map\ f\ xs$
⟨*proof*⟩

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*⟩

lemma *path-edge*: $(l1', lbl, l2') \in set\ xs \Longrightarrow path\ l1\ l2\ xs \Longrightarrow lbl \in_A weights\ l1'\ l2'$
⟨*proof*⟩

6.2 Path Weights

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

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

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

lemma *sum-le-zero*[*simp*]: $(s::'lbl) \leq 0 \longleftrightarrow s = 0$
⟨*proof*⟩

lemma *sum-le-zeroD*[*dest*]: $(x::'lbl) \leq 0 \Longrightarrow x = 0$
⟨*proof*⟩

lemma *foldr-plus-mono*: $(n::'lbl) \leq m \Longrightarrow foldr\ (+)\ xs\ n \leq foldr\ (+)\ xs\ m$
⟨*proof*⟩

lemma *sum-weights-append*:
$sum\text{-}weights\ (ys\ @\ xs) = sum\text{-}weights\ ys + sum\text{-}weights\ xs$
⟨*proof*⟩

lemma *sum-summary-prepend-le*: *sum-path-weights ys* ≤ *sum-path-weights xs* ⟹ *sum-path-weights* (*zs* @ *ys*) ≤ *sum-path-weights* (*zs* @ *xs*)
⟨*proof*⟩

lemma *sum-summary-append-le*: *sum-path-weights ys* ≤ *sum-path-weights xs* ⟹ *sum-path-weights* (*ys* @ *zs*) ≤ *sum-path-weights* (*xs* @ *zs*)
⟨*proof*⟩

lemma *foldr-plus-zero-le*: *foldr* (+) *xs* (*0*::′*lbl*) ≤ *foldr* (+) *xs a*
⟨*proof*⟩

lemma *subseq-sum-weights-le*:
assumes *xs* ⪯ *ys*
shows *sum-weights xs* ≤ *sum-weights ys*
⟨*proof*⟩

lemma *subseq-sum-path-weights-le*:
map LBL xs ⪯ *map LBL ys* ⟹ *sum-path-weights xs* ≤ *sum-path-weights ys*
⟨*proof*⟩

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

lemma *sum-weights-append-singleton*:
sum-weights (*xs* @ [*x*]) = *sum-weights xs* + *x*
⟨*proof*⟩

lemma *sum-path-weights-append-singleton*:
sum-path-weights (*xs* @ [(*l*,*x*,*l*′)]) = *sum-path-weights xs* + *x*
⟨*proof*⟩

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* $\in_A$ *weights l1 l2*))
⟨*proof*⟩

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

lemma *finite-summaries*: *finite* {*xs*. *distinct xs* ∧ (∀ (*l1*, *s*, *l2*) ∈ *set xs*. *s* $\in_A$ *weights l1 l2*)}
⟨*proof*⟩

lemma *finite-minimal-antichain-path-weightp*:
finite (*minimal-antichain* {*x*. *path-weightp l1 l2 x*})

⟨*proof*⟩

lift-definition *path-weight* :: $'vtx \Rightarrow 'vtx \Rightarrow 'lbl$ *antichain*
is λ*l1 l2. minimal-antichain* {*x. path-weightp l1 l2 x*}
⟨*proof*⟩

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

lemma *in-path-weight*: $s \in_A$ *path-weight loc1 loc2* ⟷ *s* ∈ *minimal-antichain* {*s. path-weightp loc1 loc2 s*}
⟨*proof*⟩

lemma *path-weight-refl*[*simp*]: $0 \in_A$ *path-weight loc loc*
⟨*proof*⟩

lemma *zero-in-minimal-antichain*[*simp*]: (*0*::*'lbl*) ∈ *S* ⟹ *0* ∈ *minimal-antichain S*
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *finite-path-weightp-distinct*[*simp*, *intro*]: *finite* {*xs. path-weightp-distinct l1 l2 xs*}
⟨*proof*⟩

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

lemma *path-weightp-distinct-member*:
s ∈ {*s. path-weightp l1 l2 s*} ⟹ ∃ *u. u* ∈ {*s. path-weightp-distinct l1 l2 s*} ∧ *u* ≤ *s*
⟨*proof*⟩

lemma *minimal-antichain-path-weightp-member*:
s ∈ {*xs. path-weightp l1 l2 xs*} ⟹ ∃ *u. u* ∈ *minimal-antichain* {*xs. path-weightp l1 l2 xs*} ∧ *u* ≤ *s*
⟨*proof*⟩

lemma *path-path-weight*: *path l1 l2 xs* ⟹ ∃ *s. s* $\in_A$ *path-weight l1 l2* ∧ *s* ≤ *sum-path-weights xs*
⟨*proof*⟩

lemma *path-weight-conv-path*:
$s \in_A$ *path-weight l1 l2* $\Longrightarrow \exists$ *xs. path l1 l2 xs* $\wedge$ *s* = *sum-path-weights xs* $\wedge$ ($\forall$ *ys. path l1 l2 ys* $\longrightarrow \neg$ *sum-path-weights ys* < *sum-path-weights xs*)
⟨*proof*⟩

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

lemma *path-weight-path*: $s \in_A$ *path-weight loc1 loc2* $\Longrightarrow$
($\bigwedge$*xs. optimal-path loc1 loc2 xs* $\Longrightarrow$ *distinct xs* $\Longrightarrow$ *sum-path-weights xs* = *s* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
⟨*proof*⟩

lemma *path-weight-elem-trans*:
$s \in_A$ *path-weight l1 l2* $\Longrightarrow s' \in_A$ *path-weight l2 l3* $\Longrightarrow \exists$ *u. u* $\in_A$ *path-weight l1 l3* $\wedge$ $u \leq s + s'$
⟨*proof*⟩

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*: $t1 \leq t2 \Longrightarrow s1 \leq s2 \Longrightarrow$ *results-in t1 s1* $\leq$ *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* $\Longrightarrow$ *xs* $\neq$ [] $\Longrightarrow$ *s* = *sum-path-weights xs* $\Longrightarrow$ *t* < *results-in t s*
begin

lemma *results-in-mono*:
$t1 \leq t2 \Longrightarrow$ *results-in t1 s* $\leq$ *results-in t2 s*
$s1 \leq s2 \Longrightarrow$ *results-in t s1* $\leq$ *results-in t s2*
⟨*proof*⟩

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* $\in_A$ *path-summary loc1 loc2*
⟶ (∃ *t'*≤*results-in t s*. *t'* $\in_A$ *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 ≡ ($\sum$ *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 ≠ *0* ∧
— change can only happen at timestamps not in advance of implication-frontier
(∃ *t'*. *t'* $\in_A$ *frontier* (*c-imp c0 loc*) ∧ *t'* ≤ *t*) ∧
— at loc, t is added to pointstamps n times
c1 = *c0*(|*c-pts* := (*c-pts c0*)(*loc* := *update-zmultiset* (*c-pts c0 loc*) *t n*),
— worklist at loc is adjusted by frontier changes
c-work := (*c-work c0*)(*loc* := *c-work c0 loc* +
frontier-changes (*update-zmultiset* (*c-pts c0 loc*) *t n*) (*c-pts c0 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 *loc′* ≠ *loc*
shows *c-work c1 loc′* = *c-work c0 loc′*
⟨*proof*⟩

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* ∈#$_z$ *c-work c0 loc* ∧
(∀ *t′ loc′*. *t′* ∈#$_z$ *c-work c0 loc′* ⟶ ¬ *t′* < *t*) ∧
c1 = *c0*⦇*c-imp* := (*c-imp c0*)(*loc* := *c-imp c0 loc* + {#*t′* ∈#$_z$ *c-work c0 loc*. *t′* = *t*#}),
c-work := (λ*loc′*.
— worklist entries for t are removed from loc's worklist
if loc′ = *loc then* {#*t′* ∈#$_z$ *c-work c0 loc′*. *t′* ≠ *t*#}
— worklists at other locations change by the loc's frontier change after adding summaries
else c-work c0 loc′
+ *after-summary*
(*frontier-changes* (*c-imp c0 loc* + {#*t′* ∈#$_z$ *c-work c0 loc*. *t′* = *t*#}) (*c-imp c0 loc*))
(*summary loc 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*
⟨*proof*⟩

7.2 Auxiliary

lemma *next-change-multiplicity′-unique*:
assumes $n \neq 0$
and $\exists t'. t' \in_A$ *frontier* (*c-imp c loc*) $\wedge t' \leq t$
shows $\exists! c'.$ *next-change-multiplicity′ c c′ loc t n*
⟨*proof*⟩

lemma *frontier-change-zmset-frontier*:
assumes $t \in_A$ *frontier M1* $-$ *frontier M0*
shows *zcount* (*zmset-frontier M1*) $t = 1 \wedge$ *zcount* (*zmset-frontier M0*) $t = 0$
⟨*proof*⟩

lemma *frontier-empty*[*simp*]: *frontier* $\{\#\}_z = \{\}_A$
⟨*proof*⟩

lemma *zmset-frontier-empty*[*simp*]: *zmset-frontier* $\{\#\}_z = \{\#\}_z$
⟨*proof*⟩

lemma *after-summary-empty*[*simp*]: *after-summary* $\{\#\}_z$ $S = \{\#\}_z$
⟨*proof*⟩

lemma *after-summary-empty-summary*[*simp*]: *after-summary M* $\{\}_A = \{\#\}_z$
⟨*proof*⟩

lemma *mem-frontier-diff*:
assumes $t \in_A$ *frontier M* $-$ *frontier N*
shows *zcount* (*frontier-changes M N*) $t = 1$
⟨*proof*⟩

lemma *mem-frontier-diff′*:
assumes $t \in_A$ *frontier N* $-$ *frontier M*
shows *zcount* (*frontier-changes M N*) $t = -1$
⟨*proof*⟩

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*⟩

lemma *mset-neg-after-summary*: *mset-neg M* $= \{\#\} \Longrightarrow$ *mset-neg* (*after-summary M S*) $= \{\#\}$
⟨*proof*⟩
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*

($frontier\text{-}changes$ ($c\text{-}imp$ $c1$ loc) ($c\text{-}imp$ $c0$ loc))
($summary$ loc loc')
⟨*proof*⟩

lemma *after-summary-union*: *after-summary* (M + N) S = *after-summary* M S + *after-summary* N S
⟨*proof*⟩

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'∈$UNIV$. *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 ≡
∀ 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
⟨*proof*⟩

lemma *union-frontiers-nonneg*: $0 \le$ *zcount* (*union-frontiers* c loc) t
⟨*proof*⟩

lemma *next-p-union-frontier-change*:
assumes *next-propagate'* $c0$ $c1$ loc t
and *summary* loc loc' $\neq \{\}_A$
shows *union-frontiers* $c1$ loc' =
union-frontiers $c0$ loc'
+ *after-summary*
(*frontier-changes* (*c-imp* $c1$ loc) (*c-imp* $c0$ loc))
(*summary* loc loc')
⟨*proof*⟩
lemma *init-imp-inv-imps-work-sum*: *init-config* c ⟹ *inv-imps-work-sum* c
⟨*proof*⟩
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*⟩
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*⟩

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*
⟨*proof*⟩

lemma *spec-imp-iiws*: *spec s* $\Longrightarrow$ *alw* (*holds inv-imps-work-sum*) *s*
⟨*proof*⟩

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*⟩

lemma *spec-imp-iipwn*: *spec s* $\Longrightarrow$ *alw* (*holds inv-imp-plus-work-nonneg*) *s*
⟨*proof*⟩

7.3.3 Invariant: *inv-implications-nonneg*

lemma *init-imp-inv-implications-nonneg*:
assumes *init-config c*
shows *inv-implications-nonneg c*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *alw-inv-implications-nonneg*: *spec s* ⟹ *alw* (*holds inv-implications-nonneg*) *s*
⟨*proof*⟩

lemma *after-summary-Diff*: *after-summary* (*M* − *N*) *S* = *after-summary M S* − *after-summary N S*
⟨*proof*⟩

lemma *mem-zmset-frontier*: $x \in\#_z$ *zmset-frontier M* ⟷ $x \in_A$ *frontier M*
⟨*proof*⟩

lemma *obtain-frontier-elem*:
assumes *0* < *zcount M t*
obtains *u* **where** $u \in_A$ *frontier M u* ≤ *t*
⟨*proof*⟩

lemma *frontier-unionD*: $t \in_A$ *frontier* (*M*+*N*) ⟹ *0* < *zcount M t* ∨ *0* < *zcount N t*
⟨*proof*⟩

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*⟩

lemma *obtain-elem-frontier*:
assumes *0* < *zcount M t*
obtains *s* **where** *s* ≤ *t* ∧ $s \in_A$ *frontier M*
⟨*proof*⟩

lemma *obtain-elem-zmset-frontier*:
assumes *0* < *zcount M t*
obtains *s* **where** *s* ≤ *t* ∧ *0* < *zcount* (*zmset-frontier M*) *s*
⟨*proof*⟩

lemma *ps-in-imps-wl*:
  assumes *inv-imps-work-sum c*
    and $0 < zcount\ (c\text{-}pts\ c\ loc)\ t$
  obtains *s* **where** $s \leq t \wedge 0 < zcount\ (c\text{-}imp\ c\ loc + c\text{-}work\ c\ loc)\ s$
⟨*proof*⟩

lemma *zero-le-after-summary-single*[*simp*]: $0 \leq zcount\ (after\text{-}summary\ \{\#t\#\}_z\ S)\ x$
  ⟨*proof*⟩

lemma *one-le-zcount-after-summary*: $s \in_A S \Longrightarrow 1 \leq zcount\ (after\text{-}summary\ \{\#t\#\}_z\ S)\ (results\text{-}in\ t\ s)$
  ⟨*proof*⟩

lemma *zero-lt-zcount-after-summary*: $s \in_A S \Longrightarrow 0 < zcount\ (after\text{-}summary\ \{\#t\#\}_z\ S)\ (results\text{-}in\ t\ s)$
  ⟨*proof*⟩

lemma *pos-zcount-after-summary*:
  $(\bigwedge t.\ 0 \leq zcount\ M\ t) \Longrightarrow 0 < zcount\ M\ t \Longrightarrow s \in_A S \Longrightarrow 0 < zcount\ (after\text{-}summary\ M\ S)\ (results\text{-}in\ t\ s)$
  ⟨*proof*⟩

lemma *after-summary-nonneg*: $(\bigwedge t.\ 0 \leq zcount\ M\ t) \Longrightarrow 0 \leq zcount\ (after\text{-}summary\ M\ S)\ t$
  ⟨*proof*⟩

lemma *after-summary-zmset-of-nonneg*[*simp*, *intro*]: $0 \leq zcount\ (after\text{-}summary\ (zmset\text{-}of\ M)\ S)\ t$
  ⟨*proof*⟩

lemma *pos-zcount-union-frontiers*:
  $zcount\ (after\text{-}summary\ (zmset\text{-}frontier\ (c\text{-}imp\ c\ l1))\ (summary\ l1\ l2))\ (results\text{-}in\ t\ s)$
    $\leq zcount\ (union\text{-}frontiers\ c\ l2)\ (results\text{-}in\ t\ s)$
  ⟨*proof*⟩

lemma *after-summary-Sum-fun*: $finite\ MM \Longrightarrow after\text{-}summary\ (\sum M{\in}MM.\ f\ M)\ A = (\sum M{\in}MM.\ after\text{-}summary\ (f\ M)\ A)$
  ⟨*proof*⟩

lemma *after-summary-obtain-pre*:
  assumes $\bigwedge t.\ 0 \leq zcount\ M\ t$
    and $0 < zcount\ (after\text{-}summary\ M\ S)\ t$
  obtains $t'\ s$ **where** $0 < zcount\ M\ t'\ results\text{-}in\ t'\ s = t\ s \in_A S$
  ⟨*proof*⟩

lemma *empty-antichain*[*dest*]: $x \in_A antichain\ \{\} \Longrightarrow False$
  ⟨*proof*⟩

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

lemma *impWitnessPathEx*:
assumes *t* $\in_A$ *frontier* (*c-imp c loc2*)
shows (∃ *loc1 xs*. *impWitnessPath c loc1 loc2 xs t*)
⟨*proof*⟩

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* $\in_A$ *summary loc1 loc2*}
⟨*proof*⟩

lemma *longestImpWitnessPathEx*:
assumes *t* $\in_A$ *frontier* (*c-imp c loc2*)
shows (∃ *loc1 xs*. *longestImpWitnessPath c loc1 loc2 xs t*)
⟨*proof*⟩

lemma *path-first-loc*: *path l1 l2 xs* ⟹ *xs* ≠ [] ⟹ *xs* ! *0* = (*l1'*,*s*,*l2'*) ⟹ *l1* = *l1'*
⟨*proof*⟩

lemma *find-witness-from-frontier*:
assumes *t* $\in_A$ *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'* $\in_A$ *frontier* (*c-pts c loc1*) ∨ *0* > *zcount* (*c-work c loc1*) *t'*))
⟨*proof*⟩

lemma *implication-implies-pointstamp*:
assumes *t* $\in_A$ *frontier* (*c-imp c loc*)
and *inv-imps-work-sum c*
shows ∃ *t' loc' s*. *s* $\in_A$ *path-summary loc' loc* ∧ *t* ≥ *results-in t' s* ∧
(*t'* $\in_A$ *frontier* (*c-pts c loc'*) ∨ *0* > *zcount* (*c-work c loc'*) *t'*)
⟨*proof*⟩

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*

⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

lemma *image-zmset-id*[*simp*]: {#*x*. $x \in\#_z$ *M*#} = *M*
⟨*proof*⟩

lemma *sum-pos*: *finite M* $\Longrightarrow$ $\forall x \in M$. $0 \le f\,x$ $\Longrightarrow$ $y \in M$ $\Longrightarrow$ $0 < (f\,y$::-::*ordered-comm-monoid-add*) $\Longrightarrow$ $0 < (\sum x \in M.\ f\,x)$
⟨*proof*⟩

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 $\bigwedge$*loc*. *c-work c loc* = $\{\#\}_z$
and $0 <$ *zcount M t*
shows $\exists s.\ s \le t \wedge s \in_A$ *frontier* (*c-imp c loc2*)
⟨*proof*⟩

lemma *loc-imps-fw-M-nonneg*[*simp*]:
assumes *loc-imps-fw c loc1 loc2 M xs*
and *inv-implications-nonneg c*
shows $0 \le$ *zcount M t*
⟨*proof*⟩

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 $0 < zcount\ (c\text{-}imp\ c\ loc1)\ t$
  shows $0 < zcount\ M\ (results\text{-}in\ t\ (sum\text{-}path\text{-}weights\ xs))$
  ⟨*proof*⟩

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

lemma *impl-safe*:
  assumes *inv-imps-work-sum c*
    and *inv-implications-nonneg c*
    and $\bigwedge loc.\ c\text{-}work\ c\ loc = \{\#\}_z$
  shows *impl-safe c*
  ⟨*proof*⟩

lemma *cm-preserves-impl-safe*:
  assumes *impl-safe c0*
    and *next-change-multiplicity' c0 c1 loc t n*
  shows *impl-safe c1*
  ⟨*proof*⟩

lemma *cm-preserves-safe*:
  assumes *safe c0*
    and *impl-safe c0*
    and *next-change-multiplicity' c0 c1 loc t n*
  shows *safe c1*
⟨*proof*⟩

7.5 A Better (More Invariant) Safety

definition *worklists-vacant-to* :: ('*loc*, '*t*) *configuration* ⇒ '*t* ⇒ *bool* **where**
  worklists-vacant-to c t =
    $(\forall\, loc1\ loc2\ s\ t'.\ s \in_A path\text{-}summary\ loc1\ loc2 \wedge t' \in\#_z c\text{-}work\ c\ loc1 \longrightarrow \neg\ results\text{-}in\ t'\ s \leq t)$

definition *inv-safe* :: ('*loc*, '*t*) *configuration* ⇒ *bool* **where**
  $inv\text{-}safe\ c = (\forall\, loc1\ loc2\ t\ s.\ 0 < zcount\ (c\text{-}pts\ c\ loc1)\ t$
                $\wedge\ s \in_A path\text{-}summary\ loc1\ loc2$
                $\wedge\ worklists\text{-}vacant\text{-}to\ c\ (results\text{-}in\ t\ s)$
          $\longrightarrow (\exists\, t' \leq results\text{-}in\ t\ s.\ t' \in_A frontier\ (c\text{-}imp\ c\ loc2)))$

Intuition: Unlike safe, *inv-safe* is an invariant because it only claims the safety property $t' \in_A frontier\ (c\text{-}imp\ c\ loc2)$ 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*: $\forall\, y \in \#_z N.\ \neg\ y \leq x \Longrightarrow x \in_A$ *frontier* $(M - N) \longleftrightarrow x \in_A$ *frontier M*
⟨*proof*⟩

lemma *worklists-vacant-to-trans*:
worklists-vacant-to c t $\Longrightarrow t' \leq t \Longrightarrow$ *worklists-vacant-to c t'*
⟨*proof*⟩

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 $\exists\, s \leq t.\ s \in_A$ *frontier* (*c-imp c loc2*)
⟨*proof*⟩

lemma *inv-safe*:
assumes *inv-imps-work-sum c*
and *inv-implications-nonneg c*
shows *inv-safe c*
⟨*proof*⟩

lemma *alw-conjI*: *alw P s* $\Longrightarrow$ *alw Q s* $\Longrightarrow$ *alw* $(\lambda s.\ P\ s \wedge Q\ s)\ s$
⟨*proof*⟩

lemma *alw-inv-safe*: *spec s* $\Longrightarrow$ *alw* (*holds inv-safe*) *s*
⟨*proof*⟩

lemma *empty-worklists-vacant-to*: $\forall$ *loc. c-work c loc* $= \{\#\}_z \Longrightarrow$ *worklists-vacant-to c t*
⟨*proof*⟩

lemma *inv-safe-safe*: $(\bigwedge$*loc. c-work c loc* $= \{\#\}_z) \Longrightarrow$ *inv-safe c* $\Longrightarrow$ *safe c*
⟨*proof*⟩

lemma *safe*:
assumes *inv-imps-work-sum c*
and *inv-implications-nonneg c*
and $\bigwedge$*loc. c-work c loc* $= \{\#\}_z$
shows *safe c*
⟨*proof*⟩

7.6 Implied Frontier

abbreviation *zmset-pos* **where** *zmset-pos M* $\equiv$ *zmset-of* (*mset-pos M*)

definition *implied-frontier* **where**

implied-frontier P loc = *frontier* ($\sum$ *loc*′∈*UNIV*. *after-summary* (*zmset-pos* (*P loc*′)) (*path-summary loc*′ *loc*))

definition *implied-frontier-alt* **where**
implied-frontier-alt c loc = *frontier* ($\sum$ *loc*′∈*UNIV*. *after-summary* (*zmset-frontier* (*c-pts c loc*′)) (*path-summary loc*′ *loc*))

lemma *in-frontier-least*: $x \in_A$ *frontier M* ⟹ ∀ *y*. *0* < *zcount M y* ⟶ ¬ *y* < *x*
⟨*proof*⟩

lemma *in-frontier-trans*: *0* < *zcount M y* ⟹ $x \in_A$ *frontier M* ⟹ *y* ≤ *x* ⟹ *y* $\in_A$ *frontier M*
⟨*proof*⟩

lemma *implied-frontier-alt-least*:
assumes $b \in_A$ *implied-frontier-alt c loc2*
shows ∀ *loc a*′ *s*′. *a*′ $\in_A$ *frontier* (*c-pts c loc*) ⟶ *s*′ $\in_A$ *path-summary loc loc2* ⟶ ¬ *results-in a*′ *s*′ < *b*
⟨*proof*⟩

lemma *implied-frontier-alt-in-pointstamps*:
assumes $b \in_A$ *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*
⟨*proof*⟩

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*⟩

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 $b \in_A$ *frontier* (*c-imp c loc2*)
shows $b \in_A$ *implied-frontier-alt c loc2*
⟨*proof*⟩

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*) ⟷ $b \in_A$ *implied-frontier-alt c loc*
⟨*proof*⟩

lemma *next-propagate-implied-frontier-alt-def*:
next-propagate c c' $\Longrightarrow$ *implied-frontier-alt c loc* = *implied-frontier-alt c' loc*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *antichain-eqI*: ($\bigwedge$*b. b* $\in_A$ *A* $\longleftrightarrow$ *b* $\in_A$ *B*) $\Longrightarrow$ *A* = *B*
⟨*proof*⟩

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

lemma *image-mset-mono-pos*:
$\forall$*b. 0* $\le$ *zcount A b* $\Longrightarrow$ $\forall$*b. 0* $\le$ *zcount B b* $\Longrightarrow$ *A* $\subseteq\#_z$ *B* $\Longrightarrow$ *image-zmset f A* $\subseteq\#_z$ *image-zmset f B*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *after-summary-zmset-frontier*:
after-summary (*zmset-frontier A*) *S* $\subseteq\#_z$ *after-summary* (*zmset-pos A*) *S*
⟨*proof*⟩

lemma *frontier-eqI*: $\forall$*b. 0* $\le$ *zcount A b* $\Longrightarrow$ $\forall$*b. 0* $\le$ *zcount B b* $\Longrightarrow$
A $\subseteq\#_z$ *B* $\Longrightarrow$ ($\bigwedge$*b. b* $\in\#_z$ *B* $\Longrightarrow$ $\exists$*a. a* $\in\#_z$ *A* $\wedge$ *a* $\le$ *b*) $\Longrightarrow$ *frontier A* = *frontier B*
⟨*proof*⟩

lemma *implied-frontier-implied-frontier-alt*: *implied-frontier* (*c-pts c*) *loc* = *implied-frontier-alt c loc*
⟨*proof*⟩

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 $P\ (\mathit{Finite\text{-}Set.fold}\ f\ z\ M)$
⟨*proof*⟩

8.1 Could-result-in Relation

context *dataflow-topology* **begin**

definition *cri-less-eq* :: $('loc \times 't) \Rightarrow ('loc \times 't) \Rightarrow bool$ ($\text{-}{\leq_p}\text{-}$ [*51,51*] *50*) **where**
cri-less-eq =
$(\lambda(loc1,t1)\ (loc2,t2).\ (\exists s.\ s \in_A \mathit{path\text{-}summary}\ loc1\ loc2 \wedge \mathit{results\text{-}in}\ t1\ s \leq t2))$

definition *cri-less* :: $('loc \times 't) \Rightarrow ('loc \times 't) \Rightarrow bool$ ($\text{-}{<_p}\text{-}$ [*51,51*] *50*) **where**
$\mathit{cri\text{-}less}\ x\ y = (x \leq_p y \wedge x \neq y)$

lemma *cri-asym1*: $x <_p y \longrightarrow \neg\ y <_p x$
for *x y* ⟨*proof*⟩

lemma *cri-asym2*: $x <_p y \longrightarrow x \neq y$
⟨*proof*⟩

sublocale *cri*: *order cri-less-eq cri-less*
⟨*proof*⟩

lemma *wf-cri*: $wf\ \{(l, l').\ (l, t) <_p (l', t)\}$
⟨*proof*⟩

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* :=
(λ*loc'*. *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* (λ*c*. ∃ *loc*. (*c-work c loc*) ≠ $\{\#\}_z$)
(λ*c*. *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*

$\Rightarrow$ ($'loc \times 't$) *multiset* $\Rightarrow$ *bool* **where**
NextPerformOp′ c0 c1 p Δ*neg* Δ*mint-msg* Δ*mint-self* = (
cri.next-performop′ (*exchange-config c0*) (*exchange-config c1*) *p* Δ*neg* Δ*mint-msg* Δ*mint-self*
$\wedge$ *unchanged prop-config c0 c1*
$\wedge$ *unchanged init c0 c1*)

abbreviation *NextPerformOp* **where**
NextPerformOp c0 c1 $\equiv$ $\exists$ *p* Δ*neg* Δ*mint-msg* Δ*mint-self*. *NextPerformOp′ c0 c1 p* Δ*neg* Δ*mint-msg* Δ*mint-self*

definition *NextRecvCap′*
:: ($'p$::*finite*, $'t$::*order*, $'loc$) *configuration* $\Rightarrow$ ($'p$, $'t$, $'loc$) *configuration* $\Rightarrow$ $'p$ $\Rightarrow$ $'loc \times 't$ $\Rightarrow$ *bool* **where**
NextRecvCap′ c0 c1 p t = (
cri.next-recvcap′ (*exchange-config c0*) (*exchange-config c1*) *p t*
$\wedge$ *unchanged prop-config c0 c1*
$\wedge$ *unchanged init c0 c1*)

abbreviation *NextRecvCap* **where**
NextRecvCap c0 c1 $\equiv$ $\exists$ *p t*. *NextRecvCap′ c0 c1 p t*

definition *NextSendUpd′* :: ($'p$::*finite*, $'t$::*order*, $'loc$) *configuration* $\Rightarrow$ ($'p$, $'t$, $'loc$) *configuration*
$\Rightarrow$ $'p$ $\Rightarrow$ ($'loc \times 't$) *set* $\Rightarrow$ *bool* **where**
NextSendUpd′ c0 c1 p tt = (
cri.next-sendupd′ (*exchange-config c0*) (*exchange-config c1*) *p tt*
$\wedge$ *unchanged prop-config c0 c1*
$\wedge$ *unchanged init c0 c1*)

abbreviation *NextSendUpd* **where**
NextSendUpd c0 c1 $\equiv$ $\exists$ *p tt*. *NextSendUpd′ c0 c1 p tt*

definition *NextRecvUpd′* :: ($'p$::*finite*, $'t$::*order*, $'loc$) *configuration* $\Rightarrow$ ($'p$, $'t$, $'loc$) *configuration*
$\Rightarrow$ $'p$ $\Rightarrow$ $'p$ $\Rightarrow$ *bool* **where**
NextRecvUpd′ c0 c1 p q = (
init c0 q — Once init is set we are guaranteed that the CM transitions' premises are satisfied
$\wedge$ *cri.next-recvupd′* (*exchange-config c0*) (*exchange-config c1*) *p q*
$\wedge$ *unchanged init c0 c1*
$\wedge$ ($\forall$ *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 $\equiv$ $\exists$ *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
⟨*proof*⟩

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
⟨*proof*⟩

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′))
⟨*proof*⟩

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′*)))
⟨*proof*⟩

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
⟨*proof*⟩

8.3 Auxiliary Lemmas

8.3.1 Auxiliary Lemmas for CM Conversion

lemma *apply-cm-is-cm*:
∃ *t′*. *t′* $\in_A$ *frontier* (*c-imp c loc*) ∧ *t′* ≤ *t* ⟹ *n* ≠ *0* ⟹ *next-change-multiplicity′ c* (*apply-cm c loc t n*) *loc t n*
⟨*proof*⟩

lemma *update-zmultiset-commute*:
update-zmultiset (*update-zmultiset M t′ n′*) *t n* = *update-zmultiset* (*update-zmultiset M t n*) *t′ n′*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *comp-fun-commute-apply-cm*[*simp*]: *comp-fun-commute* (λ(*loc*, *t*) *c*. *apply-cm c loc t* (*f loc t*))
⟨*proof*⟩

lemma *ex-cm-imp-conds*:
assumes ∃ *c′*. *next-change-multiplicity′ c c′ loc t n*
shows ∃ *t′*. *t′* $\in_A$ *frontier* (*c-imp c loc*) ∧ *t′* ≤ *t n* ≠ *0*
⟨*proof*⟩

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*⟩

lemma *apply-cm-preserves-cond*:
assumes $\forall$ (*loc*,*t*)$\in$*set-zmset* Δ. $\exists t'$. $t' \in_A$ *frontier* (*c-imp c0 loc*) $\wedge$ $t' \leq t$
shows $\forall$ (*loc*,*t*)$\in$*set-zmset* Δ. $\exists t'$. $t' \in_A$ *frontier* (*c-imp* (*apply-cm c0 loc' t'' n*) *loc*) $\wedge$ $t' \leq t$
⟨*proof*⟩

lemma *cm-all-eq-cm-all'*:
assumes $\forall$ (*loc*,*t*)$\in$*set-zmset* Δ. $\exists t'$. $t' \in_A$ *frontier* (*c-imp c0 loc*) $\wedge$ $t' \leq t$
shows *cm-all c0* Δ = *cm-all' c0* Δ
⟨*proof*⟩

lemma *cm-eq-the-cm*:
assumes *next-change-multiplicity' c c' loc t n*
shows *the-cm c loc t n = c'*
⟨*proof*⟩

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*)
⟨*proof*⟩

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

lemma *nop*: *loc1* $\neq$ *loc2* $\vee$ *t1* $\neq$ *t2* $\longrightarrow$
zcount (*c-pts* (*apply-cm c loc2 t2* (*zcount* Δ (*loc2*, *t2*))) *loc1*) *t1* =
zcount (*c-pts c loc1*) *t1*
⟨*proof*⟩

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*⟩

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*⟩

lemma *implications-apply-cm*[*simp*]: *c-imp* (*apply-cm c loc t n*) = *c-imp c*
⟨*proof*⟩

lemma *implications-cm-all*[*simp*]:
$c\text{-}imp\ (cm\text{-}all'\ c\ \Delta) = c\text{-}imp\ c$
⟨*proof*⟩

lemma *lift-cm-inv-cm-all'*:
assumes $(\bigwedge c0\ c1\ loc\ t\ n.\ P\ c0 \Longrightarrow next\text{-}change\text{-}multiplicity'\ c0\ c1\ loc\ t\ n \Longrightarrow P\ c1)$
and $\forall (loc,t)\in\#_z\Delta.\ \exists t'.\ t' \in_A frontier\ (c\text{-}imp\ c0\ loc) \wedge t' \le t$
and $P\ c0$
shows $P\ (cm\text{-}all'\ c0\ \Delta)$
⟨*proof*⟩

lemma *lift-cm-inv-cm-all*:
assumes $\bigwedge c0\ c1\ loc\ t\ n.\ P\ c0 \Longrightarrow next\text{-}change\text{-}multiplicity'\ c0\ c1\ loc\ t\ n \Longrightarrow P\ c1$
and $\forall (loc,t)\in\#_z\Delta.\ \exists t'.\ t' \in_A frontier\ (c\text{-}imp\ c0\ loc) \wedge t' \le t$
and $P\ c0$
shows $P\ (cm\text{-}all\ c0\ \Delta)$
⟨*proof*⟩

lemma *obtain-min-worklist*:
assumes $(a\ (loc'::(\text{-} :: finite))::(('t :: order)\ zmultiset)) \neq \{\#\}_z$
obtains $loc\ t$
where $t \in\#_z\ a\ loc$
and $\forall t'\ loc'.\ t' \in\#_z\ a\ loc' \longrightarrow \neg\ t' < t$
⟨*proof*⟩

lemma *propagate-pointstamps-eq*:
assumes $c\text{-}work\ c\ loc \neq \{\#\}_z$
shows $c\text{-}pts\ c = c\text{-}pts\ (SOME\ c'.\ \exists loc\ t.\ next\text{-}propagate'\ c\ c'\ loc\ t)$
⟨*proof*⟩

lemma *propagate-all-imp-InvGlobPointstampsEq*:
$Some\ c1 = propagate\text{-}all\ c0 \Longrightarrow c\text{-}pts\ c0 = c\text{-}pts\ c1$
⟨*proof*⟩

lemma *exists-next-propagate'*:
assumes $c\text{-}work\ c\ loc \neq \{\#\}_z$
shows $\exists c'\ loc\ t.\ next\text{-}propagate'\ c\ c'\ loc\ t$
⟨*proof*⟩

lemma *lift-propagate-inv-propagate-all*:
assumes $(\bigwedge c0\ c1\ loc\ t.\ P\ c0 \Longrightarrow next\text{-}propagate'\ c0\ c1\ loc\ t \Longrightarrow P\ c1)$
and $P\ c0$
and $propagate\text{-}all\ c0 = Some\ c1$
shows $P\ c1$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *alw-next-imp-exchange-next*: *alw Next s* ⟹ *alw cri.next* (*smap exchange-config s*)
⟨*proof*⟩

Any Tracker trace is a valid Exchange trace

lemma *spec-imp-exchange-spec*: *FullSpec s* ⟹ *cri.spec* (*smap exchange-config s*)
⟨*proof*⟩

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*⟩

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* $\in_A$ *path-summary loc1 loc2* ∧ *init c p*

$\longrightarrow (\exists\, t'.\ t' \in_A$ *frontier* (*c-imp* (*prop-config c p*) *loc2*) $\wedge\ t' \leq$ *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* ∧ $s \in_A$ *path-summary loc1 loc2* ∧ *init c p2*
$\longrightarrow (\exists\, t'.\ t' \in_A$ *frontier* (*c-imp* (*prop-config c p2*) *loc2*) $\wedge\ t' \leq$ *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*
⟨*proof*⟩

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*) $\in\#_z$ (*hd* (*c-msg* (*exchange-config c*) *p q*)). $\exists\, t'.\ t' \in_A$ *frontier* (*c-imp* (*prop-config c q*) *loc*) $\wedge\ t' \leq t$))

Pointstamps in incoming messages all satisfy the CM premise, which is required 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*) $\in\#_z$ (*hd* (*c-msg* (*exchange-config c*) *p q*)). $\exists\, t'.\ t' \in_A$ *frontier* (*c-imp* (*prop-config c q*) *loc*) $\wedge\ t' \leq t$
⟨*proof*⟩

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*⟩

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*⟩
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*⟩
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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *NextPropagate′-preserves-prop-invs*:
assumes *prop-invs* (*prop-config c0 q*)
and *NextPropagate′ c0 c1 p*
shows *prop-invs* (*prop-config c1 q*)
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *init-imp-InvGlobPointstampsEq*: *InitConfig c* ⟹ *InvGlobPointstampsEq c*
⟨*proof*⟩

lemma *init-imp-inv-init-imp-prop-safe*: *InitConfig c* ⟹ *inv-init-imp-prop-safe c*
⟨*proof*⟩

lemma *init-imp-prop-invs*: *InitConfig c* ⟹ ∀ *p. prop-invs* (*prop-config c p*)

⟨*proof*⟩

abbreviation *all-invs* **where**
$\textit{all-invs } c \equiv \textit{InvGlobPointstampsEq } c \wedge \textit{inv-init-imp-prop-safe } c \wedge (\forall p.\ \textit{prop-invs } (\textit{prop-config } c\ p))$

lemma $\textit{alw-Next}'\textit{-alw-invs}$:
assumes *holds all-invs s*
and $\textit{alw } (\textit{holds } (\lambda c.\ \textit{cri.InvMsgInGlob } (\textit{exchange-config } c)))\ s$
and *alw Next s*
shows *alw* (*holds all-invs*) *s*
⟨*proof*⟩

lemma *alw-invs*: $\textit{FullSpec } s \Longrightarrow \textit{alw } (\textit{holds all-invs})\ s$
⟨*proof*⟩

lemma *alw-InvGlobPointstampsEq*: $\textit{FullSpec } s \Longrightarrow \textit{alw } (\textit{holds InvGlobPointstampsEq})\ s$
⟨*proof*⟩

lemma *alw-inv-init-imp-prop-safe*: $\textit{FullSpec } s \Longrightarrow \textit{alw } (\textit{holds inv-init-imp-prop-safe})\ s$
⟨*proof*⟩

lemma *alw-holds-conv-shd*: $\textit{alw } (\textit{holds } \varphi)\ s = \textit{alw } (\lambda s.\ \varphi\ (\textit{shd } s))\ s$
⟨*proof*⟩

lemma *alw-prop-invs*: $\textit{FullSpec } s \Longrightarrow \textit{alw } (\textit{holds } (\lambda c.\ \forall p.\ \textit{prop-invs } (\textit{prop-config } c\ p)))\ s$
⟨*proof*⟩

lemma *nrec-pts-delayed*:
assumes *cri.InvGlobNonposImpRecordsNonpos* (*exchange-config c*)
and $\textit{zcount } (\textit{cri.records } (\textit{exchange-config } c))\ x > 0$
shows $\exists x'.\ x' \leq_p x \wedge \textit{zcount } (\textit{c-glob } (\textit{exchange-config } c)\ p)\ x' > 0$
⟨*proof*⟩

lemma *help-lemma*:
assumes $0 < \textit{zcount } (\textit{c-pts } (\textit{prop-config } c\ p)\ \textit{loc0})\ \textit{t0}$
and $(\textit{loc0}, \textit{t0}) \leq_p (\textit{loc1}, \textit{t1})$
and $\textit{s2} \in_A \textit{path-summary loc1 loc2}$
and *safe* (*prop-config c p*)
shows $\exists\ \textit{t2}.\ (\textit{t2} \leq \textit{results-in t1 s2}$
$\wedge\ \textit{t2} \in_A \textit{frontier } (\textit{c-imp } (\textit{prop-config } c\ p)\ \textit{loc2}))$
⟨*proof*⟩
lemma $\textit{lift-prop-inv-NextPropagate}'$:
assumes $(\bigwedge c0\ c1\ loc\ t.\ P\ c0 \Longrightarrow \textit{next-propagate}'\ c0\ c1\ loc\ t \Longrightarrow P\ c1)$
shows $P\ (\textit{prop-config } c0\ p') \Longrightarrow \textit{NextPropagate}'\ c0\ c1\ p \Longrightarrow P\ (\textit{prop-config } c1\ p')$

⟨*proof*⟩

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′*)
⟨*proof*⟩

lemma *NextPropagate′-next′*:
assumes *NextPropagate′ c0 c1 p*
shows $next'^{++}$ (*prop-config c0 q*) (*prop-config c1 q*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *alw-safe-combined*: *FullSpec s* ⟹ *alw* (*holds safe-combined*) *s*
⟨*proof*⟩

lemma *alw-safe-combined2*: *FullSpec* $s \Longrightarrow$ *alw* (*holds safe-combined2*) *s*
⟨*proof*⟩

lemma *alw-implication-frontier-eq-implied-frontier*:
FullSpec $s \Longrightarrow$
alw (*holds* (λc. *worklists-vacant-to* (*prop-config c p*) $b \longrightarrow$
$b \in_A$ *frontier* (*c-imp* (*prop-config c p*) *loc*) $\longleftrightarrow b \in_A$ *implied-frontier* (*c-pts* (*prop-config c p*)) *loc*)) *s*
⟨*proof*⟩

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.