

Formalization of Timely Dataflow’s Progress Tracking Protocol

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

February 6, 2026

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 computations 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	3
2	Auxiliary Lemmas	3
2.1	General	3
2.2	Sums	4
2.3	Partial Orders	4
2.4	Multisets	4
2.5	Signed Multisets	4
2.5.1	Image of a Signed Multiset	7
2.6	Streams	9
2.7	Notation	9
3	Clocks Protocol	9
4	Exchange Protocol	18
4.1	Specification	18
4.2	Auxiliary Lemmas	23
4.2.1	Transition lemmas	24
4.2.2	Facts about <i>justified</i> ’ness	27
4.2.3	Facts about <i>justified-with</i> ’ness	28

4.3	Invariants	30
4.3.1	InvRecordCount	30
4.3.2	InvCapsNonneg and InvRecordsNonneg	30
4.3.3	Resulting lemmas	31
4.3.4	SafeRecordsMono	32
4.3.5	InvJustifiedII and InvJustifiedGII	32
4.3.6	InvTempJustified	34
4.3.7	InvGlobNonposImpRecordsNonpos	34
4.3.8	SafeGlobVacantUptoImpliesStickyNrec	35
4.3.9	InvGlobNonposEqVacant	35
4.3.10	InvInfoJustifiedWithII and InvInfoJustifiedWithGII	36
4.3.11	SafeGlobMono and InvMsgInGlob	37
5	Antichains	40
6	Multigraphs with Partially Ordered Weights	42
6.1	Paths	43
6.2	Path Weights	44
7	Local Progress Propagation	47
7.1	Specification	47
7.2	Auxiliary	50
7.3	Invariants	51
7.3.1	Invariant: <i>inv-imps-work-sum</i>	51
7.3.2	Invariant: <i>inv-imp-plus-work-nonneg</i>	52
7.3.3	Invariant: <i>inv-implications-nonneg</i>	52
7.4	Proof of Safety	56
7.5	A Better (More Invariant) Safety	57
7.6	Implied Frontier	59
8	Combined Progress Tracking Protocol	61
8.1	Could-result-in Relation	61
8.2	Specification	62
8.2.1	Configuration	62
8.2.2	Initial state and state transitions	62
8.3	Auxiliary Lemmas	65
8.3.1	Auxiliary Lemmas for CM Conversion	65
8.4	Exchange is a Subsystem of Tracker	68
8.5	Definitions	69
8.6	Propagate is a Subsystem of Tracker	69
8.6.1	CM Conditions	69
8.6.2	Propagate Safety and InvGlobPointstampsEq	70
8.6.3	Propagate is a Subsystem	73
8.7	Safety Proofs	73

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 $xs :: (- :: \text{group-add}) \text{ list}$
shows $xs \neq [] \implies \text{sum-list } (tl\ xs) = (-\ hd\ xs) + \text{sum-list } xs$
 $\langle \text{proof} \rangle$

2.2 Sums

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

lemma *sum-pos-ex-elem-pos*: $(0 :: \text{int}) < (\sum m \in M. f\ m) \implies \exists m \in M. 0 < f\ m$
 $\langle \text{proof} \rangle$

lemma *sum-if-distrib-add*: $\text{finite } A \implies b \in A \implies (\sum a \in A. \text{if } a=b \text{ then } X\ b + Y\ a \text{ else } X\ a) = (\sum a \in A. X\ a) + Y\ b$
 $\langle \text{proof} \rangle$

2.3 Partial Orders

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

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

2.4 Multisets

lemma *finite-nonzero-count*: $\text{finite } \{t. \text{count } M\ t > 0\}$
 $\langle \text{proof} \rangle$

lemma *finite-count[simp]*: $\text{finite } \{t. \text{count } M\ t > i\}$
 $\langle \text{proof} \rangle$

2.5 Signed Multisets

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

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

<proof>

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

lemma *pos-zcount-in-zmset*: $0 < \text{zcount } M x \implies x \in \#_z M$
<proof>

lemma *zmset-elim-nonneg*: $x \in \#_z M \implies (\bigwedge x. x \in \#_z M \implies 0 \leq \text{zcount } M x) \implies 0 < \text{zcount } M x$
<proof>

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

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

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

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

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

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

lemma *mset-neg-zcount-nonneg*: $\text{mset-neg } M = \{\#\} \implies 0 \leq \text{zcount } M t$
<proof>

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

lemma *in-zmset-notin-mset-pos[simp]*: $x \in \#_z M \implies x \notin \# \text{mset-pos } M \implies x \in \# \text{mset-neg } M$
<proof>

lemma *in-zmset-notin-mset-neg[simp]*: $x \in \#_z M \implies x \notin \# \text{mset-neg } M \implies x \in \# \text{mset-pos } M$
<proof>

lemma *in-mset-pos-in-zmset*: $x \in \# \text{mset-pos } M \implies x \in \#_z M$
<proof>

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

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

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

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

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

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

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

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

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

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

lift-definition *update-zmultiset* :: $'t \text{ zmultiset} \Rightarrow 't \Rightarrow \text{int} \Rightarrow 't \text{ zmultiset}$ **is**
 $\lambda(A,B) T D. (\text{if } D > 0 \text{ then } (A + \text{replicate-mset } (\text{nat } D) T, B)$
 $\text{else } (A, B + \text{replicate-mset } (\text{nat } (-D)) T))$
 ⟨proof⟩

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

lemma (**in order**) *order-zmset-exists-foundation*:

fixes $t :: 'a$

assumes $0 < \text{zcount } M t$

shows $\exists s. s \leq t \wedge 0 < \text{zcount } M s \wedge (\forall u. 0 < \text{zcount } M u \longrightarrow \neg u < s)$

⟨proof⟩

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

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

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

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

2.5.1 Image of a Signed Multiset

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

syntax (ASCI)

-comprehension-zmset :: $'a \Rightarrow 'b \Rightarrow 'b \text{ zmset} \Rightarrow 'a \text{ zmset}$ $\langle (\{ \# \cdot / \cdot \cdot \cdot \#_z \cdot \cdot \cdot \} \rangle) \rangle$

syntax

-comprehension-zmset :: $'a \Rightarrow 'b \Rightarrow 'b \text{ zmset} \Rightarrow 'a \text{ zmset}$ $\langle (\{ \# \cdot / \cdot \cdot \cdot \in \#_z \cdot \cdot \cdot \} \rangle) \rangle$

syntax-consts

-comprehension-zmset \equiv *image-zmset*

translations

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

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

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

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

image-zmset f N
⟨proof⟩

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

lemma *mset-neg-image-zmset*: $\text{mset-neg } M = \{\#\} \implies \text{mset-neg } (\text{image-zmset } f M) = \{\#\}$
⟨proof⟩

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

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

lemma *pos-zcount-image-zmset[simp]*: $(\bigwedge t. 0 \leq \text{zcount } M t) \implies 0 < \text{zcount } M t \implies 0 < \text{zcount } (\text{image-zmset } f M) (f t)$
⟨proof⟩

lemma *set-zmset-transfer[transfer-rule]*:
(*rel-fun* (*pcr-zmultiset* (=)) (*rel-set* (=)))
($\lambda (Mp, Mn). \text{set-mset } Mp \cup \text{set-mset } Mn - \{x. \text{count } Mp x = \text{count } Mn x\}$)
set-zmset
⟨proof⟩

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

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

lemma *in-image-zmset-in-zmset*: $t \in \#_z \text{ image-zmset } f M \implies \exists t. t \in \#_z M$
⟨proof⟩

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

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

lemma *pos-image-zmset-obtain-pre*:
 $(\bigwedge t. 0 \leq \text{zcount } M t) \implies 0 < \text{zcount } (\text{image-zmset } f M) t \implies \exists m. 0 < \text{zcount } M m \wedge f m = t$

$M m \wedge f m = t$
<proof>

2.6 Streams

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

lemma *relatesD[dest]*: *relates* P $s \implies P$ (*shd* s) (*shd* (*stl* s))
<proof>

lemma *alw-relatesD[dest]*: *alw* (*relates* P) $s \implies P$ (*shd* s) (*shd* (*stl* s))
<proof>

lemma *relatesI[intro]*: P (*shd* s) (*shd* (*stl* s)) \implies *relates* P s
<proof>

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

lemma *alw-relates*: *alw* (*relates* P) $s \iff P$ (*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* ⇒ 't :: *order* ⇒ bool **where**
vacant-upto a $t = (\forall s. s \leq t \implies zcount$ a $s = 0)$

abbreviation *nonpos-upto* :: 't *delta-vec* ⇒ 't :: *order* ⇒ bool **where**

$nonpos\text{-}upto\ a\ t \equiv \forall s. s \leq t \longrightarrow zcount\ a\ s \leq 0$

definition $supported\text{-}strong :: 't\ delta\text{-}vec \Rightarrow 't :: order \Rightarrow bool$ **where**
 $supported\text{-}strong\ a\ t = (\exists s. s < t \wedge zcount\ a\ s < 0 \wedge nonpos\text{-}upto\ a\ s)$

definition $supported :: 't\ delta\text{-}vec \Rightarrow 't :: order \Rightarrow bool$ **where**
 $supported\ a\ t = (\exists s. s < t \wedge zcount\ a\ s < 0)$

definition $upright :: 't :: order\ delta\text{-}vec \Rightarrow bool$ **where**
 $upright\ a = (\forall t. zcount\ a\ t > 0 \longrightarrow supported\ a\ t)$

lemma $upright\text{-}alt: upright\ a \longleftrightarrow (\forall t. zcount\ a\ t > 0 \longrightarrow supported\text{-}strong\ a\ t)$
 $\langle proof \rangle$

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

lemma $beta\text{-}upright\text{-}alt:$
 $beta\text{-}upright\ va\ vb = (\forall t. zcount\ va\ t > 0 \longrightarrow (\exists s. s < t \wedge (zcount\ va\ s < 0 \vee zcount\ vb\ s < 0) \wedge nonpos\text{-}upto\ va\ s))$
 $\langle proof \rangle$

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

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

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

definition $next\text{-}performop' :: ('p, 't :: order)\ configuration \Rightarrow ('p, 't)\ configuration$
 $\Rightarrow 'p \Rightarrow 't\ count\text{-}vec \Rightarrow 't\ count\text{-}vec \Rightarrow bool$ **where**
 $next\text{-}performop'\ c0\ c1\ p\ c\ r =$
 $(let\ \Delta = zmsset\text{-}of\ r - zmsset\text{-}of\ c\ in$
 $(\forall t. int\ (count\ c\ t) \leq zcount\ (c\text{-}records\ c0)\ t)$
 $\wedge\ upright\ \Delta$
 $\wedge\ c1 = c0 \upharpoonright c\text{-}records := c\text{-}records\ c0 + \Delta,$
 $c\text{-}temp := (c\text{-}temp\ c0)(p := c\text{-}temp\ c0\ p + \Delta))$

abbreviation $next\text{-}performop$ **where**

$next\text{-performop } s \equiv (\exists p (c :: 't :: order\ count\text{-vec}) (r :: 't\ count\text{-vec}). next\text{-performop}' (shd\ s) (shd\ (stl\ s))\ p\ c\ r)$

definition $next\text{-sendupd}'$ **where**

$next\text{-sendupd}'\ c0\ c1\ p\ tt =$
 $(let\ \gamma = \{\#t \in \#_z\ c\text{-temp}\ c0\ p.\ t \in tt\#\}$ **in**
 $\gamma \neq 0$
 $\wedge\ upright\ (c\text{-temp}\ c0\ p - \gamma)$
 $\wedge\ c1 = c0 \llbracket c\text{-msg} := (c\text{-msg}\ c0)(p := \lambda q.\ c\text{-msg}\ c0\ p\ q\ @\ [\gamma]),$
 $c\text{-temp} := (c\text{-temp}\ c0)(p := c\text{-temp}\ c0\ p - \gamma) \rrbracket$)

abbreviation $next\text{-sendupd}$ **where**

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

definition $next\text{-recvupd}'$ **where**

$next\text{-recvupd}'\ c0\ c1\ p\ q =$
 $(c\text{-msg}\ c0\ p\ q \neq \square)$
 $\wedge\ c1 = c0 \llbracket c\text{-msg} := (c\text{-msg}\ c0)(p := (c\text{-msg}\ c0\ p)(q := tl\ (c\text{-msg}\ c0\ p\ q))),$
 $c\text{-glob} := (c\text{-glob}\ c0)(q := c\text{-glob}\ c0\ q + hd\ (c\text{-msg}\ c0\ p\ q)) \rrbracket$

abbreviation $next\text{-recvupd}$ **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 $Glob\ VacantUpto$ **where**

$Glob\ VacantUpto\ c\ q\ t \equiv vacant\text{-upto } (c\text{-glob } c\ q)\ t$

abbreviation $Nrec\ VacantUpto$ **where**

$Nrec\ VacantUpto\ c\ t \equiv vacant\text{-upto } (c\text{-records } c)\ t$

definition $SafeGlob\ VacantUpto\ Implies\ StickyNrec :: ('p :: finite, 't :: order)\ computation \Rightarrow bool$ **where**

$SafeGlob\ VacantUpto\ Implies\ StickyNrec\ s =$
 $(let\ c = shd\ s\ in\ \forall t\ q.\ Glob\ VacantUpto\ c\ q\ t \longrightarrow alw\ (holds\ (\lambda c.\ Nrec\ VacantUpto\ c\ t))\ s)$

definition $SafeStickyNrec\ VacantUpto :: ('p :: finite, 't :: order)\ computation \Rightarrow bool$ **where**

$SafeStickyNrec\ VacantUpto\ s =$
 $(let\ c = shd\ s\ in\ \forall t.\ Nrec\ VacantUpto\ c\ t \longrightarrow alw\ (holds\ (\lambda c.\ Nrec\ VacantUpto\ c\ t))\ s)$

definition *InvGlobVacantUptoImpliesNrec* :: ('p :: finite, 't :: order) configuration
 \Rightarrow bool **where**

InvGlobVacantUptoImpliesNrec c =
 $(\forall t q. \text{vacant-upto } (c\text{-glob } c \ q) \ t \longrightarrow \text{vacant-upto } (c\text{-records } c) \ t)$

definition *InvTempUpright* **where**

InvTempUpright c = $(\forall p. \text{upright } (c\text{-temp } c \ p))$

lemma *init-InvTempUpright*: *init-config* c \Longrightarrow *InvTempUpright* c
 \langle proof \rangle

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

\langle proof \rangle

lemma *upright-vec-add*:

assumes *upright* v1

and *upright* v2

shows *upright* (v1 + v2)

\langle proof \rangle

lemma *next-InvTempUpright*: *holds* *InvTempUpright* s \Longrightarrow *next* s \Longrightarrow *nxt* (*holds* *InvTempUpright*) s

\langle proof \rangle

lemma *alw-InvTempUpright*: *spec* s \Longrightarrow *alw* (*holds* *InvTempUpright*) s

\langle proof \rangle

definition *IncomingInfo* **where**

IncomingInfo c k p q = $(\text{sum-list } (\text{drop } k \ (c\text{-msg } c \ p \ q)) + c\text{-temp } c \ p)$

definition *InvIncomingInfoUpright* **where**

InvIncomingInfoUpright c = $(\forall k \ p \ q. \text{upright } (\text{IncomingInfo } c \ k \ p \ q))$

lemma *upright-0*: *upright* 0

\langle proof \rangle

lemma *init-InvIncomingInfoUpright*: *init-config* c \Longrightarrow *InvIncomingInfoUpright* c

\langle proof \rangle

lemma *next-InvIncomingInfoUpright*: *holds* *InvIncomingInfoUpright* s \Longrightarrow *next* s
 \Longrightarrow *nxt* (*holds* *InvIncomingInfoUpright*) s

\langle proof \rangle

lemma *alw-InvIncomingInfoUpright*: *spec* s \Longrightarrow *alw* (*holds* *InvIncomingInfoUpright*)

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

definition $\text{GlobalIncomingInfo} :: ('p :: \text{finite}, 't) \text{ configuration} \Rightarrow \text{nat} \Rightarrow 'p \Rightarrow 'p \Rightarrow 't \text{ delta-vec}$ **where**

$\text{GlobalIncomingInfo } c \ k \ p \ q = (\sum p' \in \text{UNIV}. \text{IncomingInfo } c \ (\text{if } p' = p \text{ then } k \text{ else } 0) \ p' \ q)$

abbreviation $\text{GlobalIncomingInfoAt}$ **where**

$\text{GlobalIncomingInfoAt } c \ q \equiv \text{GlobalIncomingInfo } c \ 0 \ q \ q$

definition $\text{InvGlobalRecordCount}$ **where**

$\text{InvGlobalRecordCount } c = (\forall q. \text{c-records } c = \text{GlobalIncomingInfoAt } c \ q + \text{c-glob } c \ q)$

lemma $\text{init-InvGlobalRecordCount}$: $\text{holds init-config } s \Longrightarrow \text{holds InvGlobalRecordCount } s$

$\langle \text{proof} \rangle$

lemma if-eq-same : $(\text{if } a = b \text{ then } f \ b \ \text{else } f \ a) = f \ a$

$\langle \text{proof} \rangle$

lemma $\text{next-InvGlobalRecordCount}$: $\text{holds InvGlobalRecordCount } s \Longrightarrow \text{next } s \Longrightarrow \text{next } (\text{holds InvGlobalRecordCount}) \ s$

$\langle \text{proof} \rangle$

lemma $\text{alw-InvGlobalRecordCount}$: $\text{spec } s \Longrightarrow \text{alw } (\text{holds InvGlobalRecordCount}) \ s$

$\langle \text{proof} \rangle$

definition $\text{InvGlobalIncomingInfoUpright}$ **where**

$\text{InvGlobalIncomingInfoUpright } c = (\forall k \ p \ q. \text{upright } (\text{GlobalIncomingInfo } c \ k \ p \ q))$

lemma $\text{upright-sum-upright}$: $\text{finite } X \Longrightarrow \forall x. \text{upright } (A \ x) \Longrightarrow \text{upright } (\sum x \in X. A \ x)$

$\langle \text{proof} \rangle$

lemma $\text{InvIncomingInfoUpright-imp-InvGlobalIncomingInfoUpright}$: $\text{holds InvIncomingInfoUpright } s \Longrightarrow \text{holds InvGlobalIncomingInfoUpright } s$

$\langle \text{proof} \rangle$

lemma $\text{alw-InvGlobalIncomingInfoUpright}$: $\text{spec } s \Longrightarrow \text{alw } (\text{holds InvGlobalIncomingInfoUpright}) \ s$

$\langle \text{proof} \rangle$

abbreviation *nrec-pos* **where**

$nrec\text{-}pos\ c \equiv \forall t. zcount\ (c\text{-}records\ c)\ t \geq 0$

lemma *init-nrec-pos*: $holds\ init\text{-}config\ s \implies holds\ nrec\text{-}pos\ s$

<proof>

lemma *next-nrec-pos*: $holds\ nrec\text{-}pos\ s \implies next\ s \implies nxt\ (holds\ nrec\text{-}pos)\ s$

<proof>

lemma *alw-nrec-pos*: $spec\ s \implies alw\ (holds\ nrec\text{-}pos)\ s$

<proof>

lemma *next-performop-vacant*:

$vacant\text{-}upto\ (c\text{-}records\ (shd\ s))\ t \implies next\text{-}performop\ s \implies vacant\text{-}upto\ (c\text{-}records\ (shd\ (stl\ s)))\ t$

<proof>

lemma *next-sendupd-vacant*:

$vacant\text{-}upto\ (c\text{-}records\ (shd\ s))\ t \implies next\text{-}sendupd\ s \implies vacant\text{-}upto\ (c\text{-}records\ (shd\ (stl\ s)))\ t$

<proof>

lemma *next-recvupd-vacant*:

$vacant\text{-}upto\ (c\text{-}records\ (shd\ s))\ t \implies next\text{-}recvupd\ s \implies vacant\text{-}upto\ (c\text{-}records\ (shd\ (stl\ s)))\ t$

<proof>

lemma *spec-imp-SafeStickyNrecVacantUpto-aux*: $alw\ next\ s \implies alw\ SafeStickyNrecVacantUpto\ s$

<proof>

lemma *spec-imp-SafeStickyNrecVacantUpto*: $spec\ s \implies alw\ SafeStickyNrecVacantUpto\ s$

<proof>

lemma *invs-imp-InvGlobVacantUptoImpliesNrec*:

assumes $holds\ InvGlobalIncomingInfoUpright\ s$

assumes $holds\ InvGlobalRecordCount\ s$

assumes $holds\ nrec\text{-}pos\ s$

shows $holds\ InvGlobVacantUptoImpliesNrec\ s$

<proof>

lemma *spec-imp-inv1*: $spec\ s \implies alw\ (holds\ InvGlobVacantUptoImpliesNrec)\ s$

<proof>

lemma *safe2-inv1-imp-safe*: $SafeStickyNrecVacantUpto\ s \implies holds\ InvGlobVacantUptoImpliesNrec\ s \implies SafeGlobVacantUptoImpliesStickyNrec\ s$

<proof>

lemma *spec-imp-safe*: $\text{spec } s \implies \text{alw SafeGlob VacantUptoImpliesStickyNrec } s$
 ⟨proof⟩

lemma *beta-upright-0*: $\text{beta-upright } 0 \text{ } vb$
 ⟨proof⟩

definition *PositiveImplies where*
 $\text{PositiveImplies } v \text{ } w = (\forall t. \text{zcount } v \text{ } t > 0 \implies \text{zcount } w \text{ } t > 0)$

lemma *betaupright-PositiveImplies*: $\text{upright } (va + vb) \implies \text{PositiveImplies } va \text{ } (va + vb) \implies \text{beta-upright } va \text{ } vb$
 ⟨proof⟩

lemma *betaupright-obtain-support*:
assumes *beta-upright* $va \text{ } vb$
 $\text{zcount } va \text{ } t > 0$
obtains *s where* $s < t \text{ zcount } va \text{ } s < 0 \vee \text{zcount } vb \text{ } s < 0 \text{ nonpos-upto } va \text{ } s$
 ⟨proof⟩

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

lemma *beta-upright-add*:
assumes *upright* vb
and *upright* vc
and *beta-upright* $va \text{ } vb$
shows *beta-upright* $va \text{ } (vb + vc)$
 ⟨proof⟩

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

definition *InvInfoAtBetaUpright where*
 $\text{InvInfoAtBetaUpright } c = (\forall k \text{ } p \text{ } q. \text{beta-upright } (\text{InfoAt } c \text{ } k \text{ } p \text{ } q) (\text{IncomingInfo } c \text{ } (k+1) \text{ } p \text{ } q))$

lemma *init-InvInfoAtBetaUpright*: $\text{init-config } c \implies \text{InvInfoAtBetaUpright } c$
 ⟨proof⟩

lemma *next-inv*[*consumes 1, case-names next-performop next-sendupd next-recvupd*]

stutter]:

assumes *next s*
and *next-performop s* $\implies P$
and *next-sendupd s* $\implies P$
and *next-recvupd s* $\implies P$
and *shd (stl s) = shd s* $\implies P$
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* \implies *alw (holds InvIncomingInfoUpright) s* \implies *holds InvInfoAtBetaUpright s* \implies *alw next s* \implies *alw (holds InvInfoAtBetaUpright) s*
<proof>

lemma *alw-InvInfoAtBetaUpright*: *spec s* \implies *alw (holds InvInfoAtBetaUpright) s*
<proof>

definition *InvGlobalInfoAtBetaUpright* **where**

InvGlobalInfoAtBetaUpright c = ($\forall k p q$. 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 $\implies T \subset S \implies P T \implies \exists s \in S - T$. P (insert s T)*
shows *P S*
<proof>

lemma *predicate-sum-decompose*:

fixes *f :: 'a \Rightarrow ('b :: ab-group-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) $\implies B$ (sum f Z) $\implies A$ (f x + sum f Z)*
and $\bigwedge x Z$. *B (f x) $\implies A$ (sum f Z) $\implies A$ (f x + sum f Z)*
shows *A ($\sum x \in X$. 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 \implies alw (holds InvGlobalInfoAtBetaUpright) s*
 ⟨*proof*⟩

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

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

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

lemma *gvut3:*
assumes
gvu: GlobVacantUpto (shd s) q t and
igvui: InvGlobVacantUptoImpliesNrec (shd s) and
igr: InvGlobalRecordCount (shd s) and
igiii: 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 ($\lambda c. \text{InvGlobVacantUptoImpliesNrec c}$) s) and
alw (holds ($\lambda c. \text{InvGlobalRecordCount c}$) s) and
alw (holds ($\lambda c. \text{InvGlobalIncomingInfoUpright c}$) s) and
alw (holds ($\lambda c. \text{InvGlobalInfoAtBetaUpright c}$) s) and
alw next s
shows *alw SafeStickyGlobVacantUpto s*
 ⟨*proof*⟩

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

definition *SafeGlobMono* **where**

$\text{SafeGlobMono } c0 \ c1 = (\forall p \ t. \text{GlobVacantUpto } c0 \ p \ t \longrightarrow \text{GlobVacantUpto } c1 \ p \ t)$

lemma *alw-SafeGlobMono*: $\text{spec } s \implies \text{alw}$ (relates *SafeGlobMono*) *s*
 ⟨proof⟩

4 Exchange Protocol

4.1 Specification

record $(\ 'p, \ 't)$ *configuration* =

$c\text{-temp} :: \ 'p \Rightarrow \ 't \ \text{zmultiset}$
 $c\text{-msg} :: \ 'p \Rightarrow \ 'p \Rightarrow \ 't \ \text{zmultiset list}$
 $c\text{-glob} :: \ 'p \Rightarrow \ 't \ \text{zmultiset}$
 $c\text{-caps} :: \ 'p \Rightarrow \ 't \ \text{zmultiset}$
 $c\text{-data-msg} :: (\ 'p \times \ 't) \ \text{multiset}$

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

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

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

context *order begin*

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

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

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

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

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

definition *justified* **where**

$\text{justified } C \ M = (\forall t. \ 0 < \text{zcount } M \ t \longrightarrow \text{supported } M \ t \vee (\exists t' < t. \ 0 < \text{zcount } C \ t') \vee \text{zcount } M \ t < \text{zcount } C \ t)$

lemma *justified-alt*:

justified $C M = (\forall t. 0 < \text{zcount } M t \longrightarrow \text{supported-strong } M t \vee (\exists t' < t. 0 < \text{zcount } C t') \vee \text{zcount } M t < \text{zcount } C t)$
 ⟨proof⟩

definition *justified-with where*

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

lemma *justified-with-alt: justified-with* $C M N =$

$(\forall t. 0 < \text{zcount } M t \longrightarrow$
 $(\exists s < t. (\text{zcount } M s < 0 \vee \text{zcount } N s < 0) \wedge (\forall s' < s. \text{zcount } M s' \leq 0)) \vee$
 $(\exists s < t. 0 < \text{zcount } C s) \vee$
 $\text{zcount } (M+N) t < \text{zcount } C t)$
 ⟨proof⟩

definition *PositiveImplies where*

PositiveImplies $v w \equiv \forall t. \text{zcount } v t > 0 \longrightarrow \text{zcount } w t > 0$

— A worker can mint capabilities greater or equal to any owned capability

definition *minting-self where*

minting-self $C M = (\forall t \in \#M. \exists t' \leq t. 0 < \text{zcount } C t')$

— Sending messages mints a capability at a strictly greater pointstamp

definition *minting-msg where*

minting-msg $C M = (\forall (p,t) \in \#M. \exists t' < t. 0 < \text{zcount } C t')$

definition *records where*

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

definition *InfoAt where*

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

definition *IncomingInfo* $:: ('p, 'a) \text{ configuration} \Rightarrow \text{nat} \Rightarrow 'p \Rightarrow 'p \Rightarrow 'a \text{ zmultiset}$
where

IncomingInfo $c k p q \equiv \text{sum-list } (\text{drop } k (c\text{-msg } c p q)) + c\text{-temp } c p$

definition *GlobalIncomingInfo* $:: ('p :: \text{finite}, 'a) \text{ configuration} \Rightarrow \text{nat} \Rightarrow 'p \Rightarrow 'p$
 $\Rightarrow 'a \text{ zmultiset}$ **where**

GlobalIncomingInfo $c k p q \equiv \sum p' \in UNIV. \text{IncomingInfo } c (\text{if } p' = p \text{ then } k \text{ else } 0) p' q$

abbreviation *GlobalIncomingInfoAt where*

$GlobalIncomingInfoAt\ c\ q \equiv GlobalIncomingInfo\ c\ 0\ q\ q$

definition $init-config :: ('p :: finite, 'a) configuration \Rightarrow bool$ **where**

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

definition $next-recvcap' :: ('p :: finite, 'a) configuration \Rightarrow ('p, 'a) configuration$
 $\Rightarrow 'p \Rightarrow 'a \Rightarrow bool$ **where**

$next-recvcap'\ c0\ c1\ p\ t =$
 $(p, t) \in\# c-data-msg\ c0$
 $\wedge c1 = c0((c-caps := (c-caps\ c0)(p := c-caps\ c0\ p + \{\#t\#}_z),$
 $c-data-msg := c-data-msg\ c0 - \{\#(p, t)\#}))$

abbreviation $next-recvcap$ **where**

$next-recvcap\ c0\ c1 \equiv \exists p\ t. next-recvcap'\ c0\ c1\ p\ t$

Can minting of capabilities be described as a refinement of the Abadi model?
 Short answer: No, not in general. Long answer: Could slightly modify Abadi model, such that a capability always comes with a multiplicity 2^{64} (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 $c\text{-caps } c$. In place of $nrec$ we define $records\ c$, which is the sum of all capabilities, as well as $c\text{-data-msg } c$, which contains the capabilities carried by data messages. Since $\forall p\ t. zcount\ (c\text{-caps } c\ p)\ t \leq zcount\ (records\ c)\ t$, our condition $\forall t. int\ (count\ \Delta\ neg\ t) \leq zcount\ (c\text{-caps } c0\ p)\ t$ implies the one on $nrec$ in Abadi's model.

Conditions in `performop`:

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

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

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

definition $next\text{-performop}' :: ('p::finite, 'a)\ configuration \Rightarrow ('p, 'a)\ configuration \Rightarrow 'p \Rightarrow 'a\ multiset \Rightarrow ('p \times 'a)\ multiset \Rightarrow 'a\ multiset \Rightarrow bool$ **where**

$next\text{-performop}'\ c0\ c1\ p\ \Delta\ neg\ \Delta\ mint\text{-msg}\ \Delta\ mint\text{-self} =$
— $\Delta\ pos$ contains all positive changes, Δ the combined positive and negative changes

(let $\Delta\ pos = timestamps\ (z\text{mset-of } \Delta\ mint\text{-msg}) + z\text{mset-of } \Delta\ mint\text{-self};$
 $\Delta = \Delta\ pos - z\text{mset-of } \Delta\ neg$

in

$(\Delta\ mint\text{-msg} \neq \{\#\} \vee z\text{mset-of } \Delta\ mint\text{-self} - z\text{mset-of } \Delta\ neg \neq \{\#\}_z)$

$\wedge (\forall t. int\ (count\ \Delta\ neg\ t) \leq zcount\ (c\text{-caps } c0\ p)\ t)$

— Pointstamps added in $\Delta\ mint\text{-self}$ are minted at p

$\wedge minting\text{-self}\ (c\text{-caps } c0\ p)\ \Delta\ mint\text{-self}$

— Pointstamps added in $\Delta\ mint\text{-msg}$ correspond to sent data messages

$\wedge minting\text{-msg}\ (c\text{-caps } c0\ p)\ \Delta\ mint\text{-msg}$

— Worker immediately knows about dropped and minted capabilities

$\wedge c1 = c0\ (c\text{-caps} := (c\text{-caps } c0)\ (p := c\text{-caps } c0\ p + z\text{mset-of } \Delta\ mint\text{-self} - z\text{mset-of } \Delta\ neg),$

— 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.

$c\text{-data-msg} := c\text{-data-msg } c0 + \Delta\ mint\text{-msg},$

$c\text{-temp} := (c\text{-temp } c0)\ (p := c\text{-temp } c0\ p + \Delta))$

abbreviation $next\text{-performop}$ **where**

$next\text{-performop } c0 \ c1 \equiv (\exists p \ \Delta neg \ \Delta mint\text{-msg} \ \Delta mint\text{-self}. \ next\text{-performop}' \ c0 \ c1 \ p \ \Delta neg \ \Delta mint\text{-msg} \ \Delta mint\text{-self})$

definition $next\text{-sendupd}' :: ('p :: finite, 'a) \text{ configuration} \Rightarrow ('p, 'a) \text{ configuration} \Rightarrow 'p \Rightarrow 'a \text{ set} \Rightarrow bool$ **where**

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

abbreviation $next\text{-sendupd}$ **where**

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

definition $next\text{-recvupd}' :: ('p :: finite, 'a) \text{ configuration} \Rightarrow ('p, 'a) \text{ configuration} \Rightarrow 'p \Rightarrow 'p \Rightarrow bool$ **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 $next\text{-recvupd}$ **where**

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

definition $next'$ **where**

$next' \ c0 \ c1 = (next\text{-performop } c0 \ c1 \ \vee \ next\text{-sendupd } c0 \ c1 \ \vee \ next\text{-recvupd } c0 \ c1$
 $\ \ \vee \ next\text{-recvup } c0 \ c1 \ \vee \ c1 = c0)$

abbreviation $next$ **where**

$next \ s \equiv next' \ (shd \ s) \ (shd \ (stl \ s))$

definition $spec :: ('p :: finite, 'a) \text{ computation} \Rightarrow bool$ **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 $SafeGlobVacantUptoImpliesStickyNrec :: ('p :: finite, 'a) \text{ computation} \Rightarrow bool$ **where**

$SafeGlobVacantUptoImpliesStickyNrec \ s =$
 $(let \ c = shd \ s \text{ in } \forall t \ q. \ GlobVacantUpto \ c \ q \ t \longrightarrow alw \ (holds \ (\lambda c. \ RecordsVacantUpto \ c \ t)) \ s)$

4.2 Auxiliary Lemmas

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

assumes *finite S*
and *empty: P {}*
and *select: $\bigwedge T. \text{finite } T \implies T \subset S \implies P T \implies \exists s \in S - T. P (\text{insert } s T)$*
shows *P S*
<proof>

lemma *finite-induct-decompose-sum*:

fixes *f :: 'c \Rightarrow ('b :: comm-monoid-add)*
assumes *finite X*
and *x \in X*
and *A (f x)*
and *$\forall Z. B (\text{sum } f Z)$*
and *$\bigwedge x Z. A (f x) \implies B (\text{sum } f Z) \implies A (f x + \text{sum } f Z)$*
and *$\bigwedge x Z. B (f x) \implies A (\text{sum } f Z) \implies A (f x + \text{sum } f Z)$*
shows *A ($\sum x \in X. f x$)*
<proof>

lemma *minting-msg-add-records*: *minting-msg C1 M $\implies \forall t. 0 \leq \text{zcount } C2 t \implies \text{minting-msg } (C1 + C2) M$*
<proof>

lemma *add-less*: *(a :: int) < c $\implies b \leq 0 \implies a + b < c$*
<proof>

lemma *disj3-split*: *P \vee Q \vee R $\implies (P \implies \text{thesis}) \implies (\neg P \wedge Q \implies \text{thesis}) \implies (\neg P \implies \neg Q \implies R \implies \text{thesis}) \implies \text{thesis}$*
<proof>

lemma *filter-zmset-conclude-predicate*: *0 < zcount {# x \in #_z M. P x #} x $\implies 0 < \text{zcount } M x \implies P x$*
<proof>

lemma *alw-holds2*: *alw (holds P) ss = (P (shd ss) \wedge alw (holds P) (stl ss))*
<proof>

lemma *zmset-of-remove1-mset*: *x \in # M $\implies \text{zmset-of } (\text{remove1-mset } x M) = \text{zmset-of } M - \{\#x\#_z$*
<proof>

lemma *timestamps-zmset-of-pair-image[simp]*: *timestamps (zmset-of {# (c,t). t \in # M #}) = \text{zmset-of } M*
<proof>

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

lemma *lift-invariant-to-spec*:

assumes $(\bigwedge c. \text{init-config } c \implies P \ c)$

and $(\bigwedge s. \text{holds } P \ s \implies \text{next } s \implies \text{next } (\text{holds } P) \ s)$

shows $\text{spec } s \implies \text{alw } (\text{holds } P) \ s$

<proof>

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

<proof>

lemma *timestamps-zmset-of[simp]*: $\text{timestamps } (\text{zmset-of } M) = \text{zmset-of } \{\# \ t.$

$(p,t) \in \# \ M \ \#\}$

<proof>

lemma *vacant-upto-add*: $\text{vacant-upto } a \ t \implies \text{vacant-upto } b \ t \implies \text{vacant-upto } (a+b) \ t$

<proof>

lemma *nonpos-upto-add*: $\text{nonpos-upto } a \ t \implies \text{nonpos-upto } b \ t \implies \text{nonpos-upto } (a+b) \ t$

<proof>

lemma *nonzero-lt-gtD*: $(n:::\text{linorder}) \neq 0 \implies 0 < n \vee n < 0$

<proof>

lemma *zero-lt-diff*: $(0::\text{int}) < a - b \implies b \geq 0 \implies 0 < a$

<proof>

lemma *zero-lt-add-disj*: $0 < (a::\text{int}) + b \implies 0 \leq a \implies 0 \leq b \implies 0 < a \vee 0 < b$

<proof>

4.2.1 Transition lemmas

lemma *next-performopD*:

assumes $\text{next-performop}' \ c0 \ c1 \ p \ \Delta \text{neg} \ \Delta \text{mint-msg} \ \Delta \text{mint-self}$

shows

$\Delta \text{mint-msg} \neq \{\#\} \vee \text{zmset-of } \Delta \text{mint-self} - \text{zmset-of } \Delta \text{neg} \neq \{\#\}_z$

$\forall t. \text{int } (\text{count } \Delta \text{neg } t) \leq \text{zcount } (c\text{-caps } c0 \ p) \ t$

$\text{minting-self } (c\text{-caps } c0 \ p) \ \Delta \text{mint-self}$

$\text{minting-msg } (c\text{-caps } c0 \ p) \ \Delta \text{mint-msg}$

$c\text{-temp } c1 = (c\text{-temp } c0)(p := c\text{-temp } c0 \ p + (\text{timestamps } (\text{zmset-of } \Delta \text{mint-msg})$

$+ \text{zmset-of } \Delta \text{mint-self} - \text{zmset-of } \Delta \text{neg}))$

$c\text{-msg } c1 = c\text{-msg } c0$

$c\text{-glob } c1 = c\text{-glob } c0$

$c\text{-data-msg } c1 = c\text{-data-msg } c0 + \Delta \text{mint-msg}$

$c\text{-caps } c1 = (c\text{-caps } c0)(p := c\text{-caps } c0 \ p + (\text{zmset-of } \Delta \text{mint-self} - \text{zmset-of } \Delta \text{neg}))$

<proof>

lemma *next-performop-complexD*:

assumes *next-performop'* $c0\ c1\ p\ \Delta neg\ \Delta mint\text{-}msg\ \Delta mint\text{-}self$

shows

$records\ c1 = records\ c0 + (timestamps\ (zmsset\text{-}of\ \Delta mint\text{-}msg) + zmsset\text{-}of\ \Delta mint\text{-}self - zmsset\text{-}of\ \Delta neg)$

$GlobalIncomingInfoAt\ c1\ q = GlobalIncomingInfoAt\ c0\ q + (timestamps\ (zmsset\text{-}of\ \Delta mint\text{-}msg) + zmsset\text{-}of\ \Delta mint\text{-}self - zmsset\text{-}of\ \Delta neg)$

$IncomingInfo\ c1\ k\ p'\ q = (if\ p' = p$

$then\ IncomingInfo\ c0\ k\ p'\ q + (timestamps\ (zmsset\text{-}of\ \Delta mint\text{-}msg) + zmsset\text{-}of\ \Delta mint\text{-}self - zmsset\text{-}of\ \Delta neg)$

$else\ IncomingInfo\ c0\ k\ p'\ q)$

$\forall t' < t. zcount\ (c\text{-}caps\ c0\ p)\ t' = 0 \implies zcount\ (timestamps\ (zmsset\text{-}of\ \Delta mint\text{-}msg))\ t = 0$

$InfoAt\ c1\ k\ p'\ q = InfoAt\ c0\ k\ p'\ q$

<proof>

lemma *next-sendupdD*:

assumes *next-sendupd'* $c0\ c1\ p\ tt$

shows

$\{\#t \in \#_z\ c\text{-}temp\ c0\ p. t \in tt\#\} \neq \{\#\}_z$

$justified\ (c\text{-}caps\ c0\ p)\ (c\text{-}temp\ c0\ p - \{\#t \in \#_z\ c\text{-}temp\ c0\ p. t \in tt\#\})$

$c\text{-}temp\ c1\ p' = (if\ p' = p\ then\ c\text{-}temp\ c0\ p - \{\#t \in \#_z\ c\text{-}temp\ c0\ p. t \in tt\#\}$

$else\ c\text{-}temp\ c0\ p')$

$c\text{-}msg\ c1 = (\lambda p'\ q. if\ p' = p\ then\ c\text{-}msg\ c0\ p\ q @ [\{\#t \in \#_z\ c\text{-}temp\ c0\ p. t \in tt\#\}] else\ c\text{-}msg\ c0\ p'\ q)$

$c\text{-}glob\ c1 = c\text{-}glob\ c0$

$c\text{-}caps\ c1 = c\text{-}caps\ c0$

$c\text{-}data\text{-}msg\ c1 = c\text{-}data\text{-}msg\ c0$

<proof>

lemma *next-sendupd-complexD*:

assumes *next-sendupd'* $c0\ c1\ p\ tt$

shows

$records\ c1 = records\ c0$

$IncomingInfo\ c1\ 0 = IncomingInfo\ c0\ 0$

$IncomingInfo\ c1\ k\ p'\ q = (if\ p' = p \wedge length\ (c\text{-}msg\ c0\ p\ q) < k$

$then\ IncomingInfo\ c0\ k\ p'\ q - \{\#t \in \#_z\ c\text{-}temp\ c0\ p'. t \in tt\#\}$

$else\ IncomingInfo\ c0\ k\ p'\ q)$

$k \leq length\ (c\text{-}msg\ c0\ p\ q) \implies IncomingInfo\ c1\ k\ p'\ q = IncomingInfo\ c0\ k\ p'\ q$

$length\ (c\text{-}msg\ c0\ p\ q) < k \implies$

$IncomingInfo\ c1\ k\ p'\ q = (if\ p' = p$

$then\ IncomingInfo\ c0\ k\ p'\ q - \{\#t \in \#_z\ c\text{-}temp\ c0\ p'. t \in tt\#\}$

$else\ IncomingInfo\ c0\ k\ p'\ q)$

$GlobalIncomingInfoAt\ c1\ q = GlobalIncomingInfoAt\ c0\ q$

$InfoAt\ c1\ k\ p'\ q = (if\ p' = p \wedge k = length\ (c\text{-}msg\ c0\ p\ q)\ then\ \{\#t \in \#_z\ c\text{-}temp\ c0\ p'. t \in tt\#\} else\ InfoAt\ c0\ k\ p'\ q)$

$\langle proof \rangle$

lemma *next-recvupdD*:

assumes *next-recvupd'* $c0\ c1\ p\ q$

shows

$c\text{-msg}\ c0\ p\ q \neq []$

$c\text{-temp}\ c1 = c\text{-temp}\ c0$

$c\text{-msg}\ c1 = (\lambda p'\ q'. \text{if } p' = p \wedge q' = q \text{ then } tl\ (c\text{-msg}\ c0\ p\ q) \text{ else } c\text{-msg}\ c0\ p'\ q')$

$c\text{-glob}\ c1 = (c\text{-glob}\ c0)(q := c\text{-glob}\ c0\ q + hd\ (c\text{-msg}\ c0\ p\ q))$

$c\text{-caps}\ c1 = c\text{-caps}\ c0$

$c\text{-data-msg}\ c1 = c\text{-data-msg}\ c0$

$\langle proof \rangle$

lemma *next-recvupd-complexD*:

assumes *next-recvupd'* $c0\ c1\ p\ q$

shows

$records\ c1 = records\ c0$

$IncomingInfo\ c1\ 0\ p'\ q' = (\text{if } p' = p \wedge q' = q \text{ then } IncomingInfo\ c0\ 0\ p'\ q' - hd\ (c\text{-msg}\ c0\ p\ q) \text{ else } IncomingInfo\ c0\ 0\ p'\ q')$

$IncomingInfo\ c1\ k\ p'\ q' = (\text{if } p' = p \wedge q' = q \text{ then } IncomingInfo\ c0\ (k+1)\ p'\ q' \text{ else } IncomingInfo\ c0\ k\ p'\ q')$

$GlobalIncomingInfoAt\ c1\ q' = (\text{if } q' = q \text{ then } GlobalIncomingInfoAt\ c0\ q' - hd\ (c\text{-msg}\ c0\ p\ q) \text{ else } GlobalIncomingInfoAt\ c0\ q')$

$InfoAt\ c1\ k\ p\ q = InfoAt\ c0\ (k+1)\ p\ q$

$InfoAt\ c1\ k\ p'\ q' = (\text{if } p' = p \wedge q' = q \text{ then } InfoAt\ c0\ (k+1)\ p\ q \text{ else } InfoAt\ c0\ k\ p'\ q')$

$\langle proof \rangle$

lemma *next-recvcapD*:

assumes *next-recvcap'* $c0\ c1\ p\ t$

shows

$(p,t) \in \# c\text{-data-msg}\ c0$

$c\text{-temp}\ c1 = c\text{-temp}\ c0$

$c\text{-msg}\ c1 = c\text{-msg}\ c0$

$c\text{-glob}\ c1 = c\text{-glob}\ c0$

$c\text{-caps}\ c1 = (c\text{-caps}\ c0)(p := c\text{-caps}\ c0\ p + \{\#t\# \}_z)$

$c\text{-data-msg}\ c1 = c\text{-data-msg}\ c0 - \{\#(p,t)\#\}$

$\langle proof \rangle$

lemma *next-recvcap-complexD*:

assumes *next-recvcap'* $c0\ c1\ p\ t$

shows

$records\ c1 = records\ c0$

$IncomingInfo\ c1 = IncomingInfo\ c0$

$GlobalIncomingInfo\ c1 = GlobalIncomingInfo\ c0$

$InfoAt\ c1\ k\ p'\ q = InfoAt\ c0\ k\ p'\ q$

$\langle proof \rangle$

lemma *ex-next-recvupd*:
assumes $c\text{-msg } c0 \ p \ q \neq []$
shows $\exists c1. \text{next-recvupd}' \ c0 \ c1 \ p \ q$
 $\langle \text{proof} \rangle$

4.2.2 Facts about *justified*'ness

lemma *justified-empty[simp]*: *justified* $\{\#\}_z \ \{\#\}_z$
 $\langle \text{proof} \rangle$

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

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

lemma *justified-add*:
assumes *justified* $C1 \ M1$
and *justified* $C2 \ M2$
and $\forall t. \ 0 \leq \text{zcount } C1 \ t$
and $\forall t. \ 0 \leq \text{zcount } C2 \ t$
shows *justified* $(C1 + C2) \ (M1 + M2)$
 $\langle \text{proof} \rangle$

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

lemma *justified-add-records*:
assumes *justified* $C \ M$
and $\forall t. \ 0 \leq \text{zcount } C' \ t$
shows *justified* $(C + C') \ M$
 $\langle \text{proof} \rangle$

lemma *justified-add-zmset-records*:
assumes *justified* $C \ M$
shows *justified* $(\text{add-zmset } t \ C) \ M$
 $\langle \text{proof} \rangle$

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

lemma *justified-add-msg-delta*:

assumes *justified* $C\ M$
and *minting-msg* $C\ \Delta$
and $\forall t. 0 \leq \text{zcount } C\ t$
shows *justified* $C\ (M + \text{timestamps } (\text{zmsset-of } \Delta))$
 $\langle \text{proof} \rangle$

lemma *justified-add-same*:

assumes *justified* $C\ M$
and *minting-self* $C\ \Delta$
and $\forall t. 0 \leq \text{zcount } C\ t$
shows *justified* $(C + \text{zmsset-of } \Delta)\ (M + \text{zmsset-of } \Delta)$
 $\langle \text{proof} \rangle$

4.2.3 Facts about *justified-with*'ness

lemma *justified-with-add-records*:

assumes *justified-with* $C1\ M\ N$
and $\forall t. 0 \leq \text{zcount } C2\ t$
shows *justified-with* $(C1+C2)\ M\ N$
 $\langle \text{proof} \rangle$

lemma *justified-with-leastI*:

assumes
 $(\forall t. 0 < \text{zcount } M\ t \longrightarrow (\forall t' < t. \text{zcount } M\ t' \leq 0) \longrightarrow$
 $(\exists s < t. (\text{zcount } M\ s < 0 \vee \text{zcount } N\ s < 0) \wedge (\forall s' < s. \text{zcount } M\ s' \leq 0)) \vee$
 $(\exists s < t. 0 < \text{zcount } C\ s) \vee$
 $\text{zcount } (M+N)\ t < \text{zcount } C\ t)$
shows *justified-with* $C\ M\ N$
 $\langle \text{proof} \rangle$

lemma *justified-with-add*:

assumes *justified-with* $C1\ M\ N1$
and *justified* $C1\ N1$
and *justified* $C2\ N2$
and $\forall t. 0 \leq \text{zcount } C1\ t$
and $\forall t. 0 \leq \text{zcount } C2\ t$
shows *justified-with* $(C1+C2)\ M\ (N1+N2)$
 $\langle \text{proof} \rangle$

lemma *justified-with-sum'*:

assumes *finite* $X\ X \neq \{\}$
and $\forall x \in X. \text{justified-with } (C\ x)\ M\ (N\ x)$
and $\forall x \in X. \text{justified } (C\ x)\ (N\ x)$
and $\forall x \in X. \forall t. 0 \leq \text{zcount } (C\ x)\ t$
shows *justified-with* $(\sum x \in X. C\ x)\ M\ (\sum x \in X. N\ x)$
 $\langle \text{proof} \rangle$

lemma *justified-with-sum*:

assumes *finite* X $X \neq \{\}$
and $x \in X$
and *justified-with* $(C\ x)$ M $(N\ x)$
and $\forall x \in X. \textit{justified} (C\ x) (N\ x)$
and $\forall x \in X. \forall t. 0 \leq \textit{zcount} (C\ x)\ t$
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 $\forall t. 0 \leq \textit{zcount} C\ t$
shows *justified-with* $(C + \textit{zmsset-of } \Delta)$ M $(N + \textit{zmsset-of } \Delta)$
<proof>

lemma *justified-with-add-msg-delta*:

assumes *justified-with* C M N
and *minting-msg* C Δ
and $\forall t. 0 \leq \textit{zcount} C\ t$
shows *justified-with* C M $(N + \textit{timestamps} (\textit{zmsset-of } \Delta))$
<proof>

lemma *justified-with-diff*:

assumes *justified-with* C M N
and $\forall t. 0 \leq \textit{zcount} C\ t$
and $\forall t. \textit{count } \Delta\ t \leq \textit{zcount} C\ t$
and *justified* C N
shows *justified-with* $(C - \textit{zmsset-of } \Delta)$ M $(N - \textit{zmsset-of } \Delta)$
<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-zmsset[simp]*:

assumes *justified-with* C M N
shows *justified-with* $(\textit{add-zmsset } c\ C)$ M N
<proof>

lemma *next-performop'-preserves-justified-with*:

assumes *justified-with* $(c\ \textit{caps } c0\ p)$ M N
and *next-performop'* $c0$ $c1$ p $\Delta_{\textit{neg}}$ $\Delta_{\textit{mint-msg}}$ $\Delta_{\textit{mint-self}}$
and $\forall t. 0 \leq \textit{zcount} (c\ \textit{caps } c0\ p)\ t$
and *justified* $(c\ \textit{caps } c0\ p)$ N
shows *justified-with* $(c\ \textit{caps } c0\ p + \textit{zmsset-of } \Delta_{\textit{mint-self}} - \textit{zmsset-of } \Delta_{\textit{neg}})$ M
 $(N + \textit{zmsset-of } \Delta_{\textit{mint-self}} + \textit{timestamps} (\textit{zmsset-of } \Delta_{\textit{mint-msg}}) - \textit{zmsset-of } \Delta_{\textit{neg}})$
<proof>

4.3 Invariants

4.3.1 InvRecordCount

InvRecordCount states that for every processor, its local approximation $c\text{-glob } c \ q$ 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**

$$InvRecordCount \ c \equiv \forall q. \ records \ c = GlobalIncomingInfoAt \ c \ q + c\text{-glob } c \ q$$

lemma *init-config-implies-InvRecordCount*: $init\text{-}config \ c \implies InvRecordCount \ c$
<proof>

lemma *performop-preserves-InvRecordCount*:

assumes $InvRecordCount \ c0$

and $next\text{-}performop' \ c0 \ c1 \ p \ \Delta neg \ \Delta mint\text{-}msg \ \Delta mint\text{-}self$

shows $InvRecordCount \ c1$

<proof>

lemma *sendupd-preserves-InvRecordCount*:

assumes $InvRecordCount \ c0$

and $next\text{-}sendupd' \ c0 \ c1 \ p \ tt$

shows $InvRecordCount \ c1$

<proof>

lemma *recvupd-preserves-InvRecordCount*:

assumes $InvRecordCount \ c0$

and $next\text{-}recvupd' \ c0 \ c1 \ p \ q$

shows $InvRecordCount \ c1$

<proof>

lemma *recvcap-preserves-InvRecordCount*:

assumes $InvRecordCount \ c0$

and $next\text{-}recvcap' \ c0 \ c1 \ p \ t$

shows $InvRecordCount \ c1$

<proof>

lemma *next-preserves-InvRecordCount*: $InvRecordCount \ c0 \implies next' \ c0 \ c1 \implies InvRecordCount \ c1$

<proof>

lemma *alw-InvRecordCount*: $spec \ s \implies alw \ (holds \ InvRecordCount) \ s$

<proof>

4.3.2 InvCapsNonneg and InvRecordsNonneg

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

definition *InvCapsNonneg* :: $('p :: finite, 'a) \ configuration \Rightarrow bool$ **where**

$InvCapsNonneg\ c = (\forall p\ t.\ 0 \leq zcount\ (c\text{-caps}\ c\ p)\ t)$

definition *InvRecordsNonneg* **where**

$InvRecordsNonneg\ c = (\forall t.\ 0 \leq zcount\ (records\ c)\ t)$

lemma *init-config-implies-InvCapsNonneg*: $init\text{-}config\ c \implies InvCapsNonneg\ c$
(*proof*)

lemma *performop-preserves-InvCapsNonneg*:

assumes $InvCapsNonneg\ c0$

and $next\text{-}performop'\ c0\ c1\ p\ \Delta_m\ \Delta_{p1}\ \Delta_{p2}$

shows $InvCapsNonneg\ c1$

(*proof*)

lemma *sendupd-performs-InvCapsNonneg*:

assumes $InvCapsNonneg\ c0$

and $next\text{-}sendupd'\ c0\ c1\ p\ tt$

shows $InvCapsNonneg\ c1$

(*proof*)

lemma *recvupd-preserves-InvCapsNonneg*:

assumes $InvCapsNonneg\ c0$

and $next\text{-}recvupd'\ c0\ c1\ p\ q$

shows $InvCapsNonneg\ c1$

(*proof*)

lemma *recvcap-preserves-InvCapsNonneg*:

assumes $InvCapsNonneg\ c0$

and $next\text{-}recvcap'\ c0\ c1\ p\ t$

shows $InvCapsNonneg\ c1$

(*proof*)

lemma *next-preserves-InvCapsNonneg*: $holds\ InvCapsNonneg\ s \implies next\ s \implies nxt$
(*holds InvCapsNonneg*) s

(*proof*)

lemma *alw-InvCapsNonneg*: $spec\ s \implies alw\ (holds\ InvCapsNonneg)\ s$

(*proof*)

lemma *alw-InvRecordsNonneg*: $spec\ s \implies alw\ (holds\ InvRecordsNonneg)\ s$

(*proof*)

4.3.3 Resulting lemmas

lemma *pos-caps-pos-records*:

assumes $InvCapsNonneg\ c$

shows $0 < zcount\ (c\text{-caps}\ c\ p)\ x \implies 0 < zcount\ (records\ c)\ x$

(*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 \Rightarrow bool **where**
SafeRecordsMono $s = (\forall t. \text{RecordsVacantUpto } (\text{shd } s) t \longrightarrow \text{alw } (\text{holds } (\lambda c. \text{RecordsVacantUpto } c t)) s)$

lemma *performop-preserves-RecordsVacantUpto*:

assumes *RecordsVacantUpto* $c0\ t$
and *next-performop'* $c0\ c1\ p\ \Delta_{\text{neg}}\ \Delta_{\text{mint-msg}}\ \Delta_{\text{mint-self}}$
and *InvRecordsNonneg* $c1$
and *InvCapsNonneg* $c0$
shows *RecordsVacantUpto* $c1\ t$
<proof>

lemma *next'-preserves-RecordsVacantUpto*:

fixes $c0 :: ('p::\text{finite}, 'a)\ \text{configuration}$
shows *InvCapsNonneg* $c0 \Longrightarrow \text{InvRecordsNonneg } c1 \Longrightarrow \text{RecordsVacantUpto } c0\ t \Longrightarrow \text{next}'\ c0\ c1 \Longrightarrow \text{RecordsVacantUpto } c1\ t$
<proof>

lemma *alw-next-implies-alw-SafeRecordsMono*:

$\text{alw next } s \Longrightarrow \text{alw } (\text{holds } \text{InvCapsNonneg})\ s \Longrightarrow \text{alw } (\text{holds } \text{InvRecordsNonneg})\ s \Longrightarrow \text{alw } \text{SafeRecordsMono } s$
<proof>

lemma *alw-SafeRecordsMono*: $\text{spec } s \Longrightarrow \text{alw } \text{SafeRecordsMono } s$

<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**

$\text{InvJustifiedII } c = (\forall k\ p\ q. \text{justified } (c\text{-caps } c\ p)\ (\text{IncomingInfo } c\ k\ p\ q))$

definition *InvJustifiedGII* **where**

$\text{InvJustifiedGII } c = (\forall k\ p\ q. \text{justified } (\text{records } c)\ (\text{GlobalIncomingInfo } c\ k\ p\ q))$

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* $(c\text{-caps } c0\ p)\ M$
and *next-performop'* $c0\ c1\ p\ \Delta_{\text{neg}}\ \Delta_{\text{mint-msg}}\ \Delta_{\text{mint-self}}$
and *InvCapsNonneg* $c0$

shows *justified* (c-caps c1 p) (M + (timestamps (zmsset-of Δ mint-msg) + zmsset-of Δ mint-self - zmsset-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 \implies *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 \implies *InvJustifiedII* c0 \implies *next'* c0 c1 \implies *InvJustifiedII* c1
 ⟨proof⟩

lemma *alw-InvJustifiedII*: *spec* s \implies *alw* (holds *InvJustifiedII*) s

⟨proof⟩

lemma *alw-InvJustifiedGII*: *spec* s \implies *alw* (holds *InvJustifiedGII*) s

⟨proof⟩

4.3.6 InvTempJustified

definition *InvTempJustified* **where**

$$\text{InvTempJustified } c = (\forall p. \text{justified } (c\text{-caps } c \ p) \ (c\text{-temp } c \ p))$$

lemma *init-config-implies-InvTempJustified*: $\text{init-config } c \implies \text{InvTempJustified } c$
 $\langle \text{proof} \rangle$

lemma *recvcap-preserves-InvTempJustified*:

assumes *InvTempJustified* $c0$
and *next-recvcap'* $c0 \ c1 \ p \ t$
shows *InvTempJustified* $c1$
 $\langle \text{proof} \rangle$

lemma *recvupd-preserves-InvTempJustified*:

assumes *InvTempJustified* $c0$
and *next-recvupd'* $c0 \ c1 \ p \ t$
shows *InvTempJustified* $c1$
 $\langle \text{proof} \rangle$

lemma *sendupd-preserves-InvTempJustified*:

assumes *InvTempJustified* $c0$
and *next-sendupd'* $c0 \ c1 \ p \ tt$
shows *InvTempJustified* $c1$
 $\langle \text{proof} \rangle$

lemma *performop-preserves-InvTempJustified*:

assumes *InvTempJustified* $c0$
and *next-performop'* $c0 \ c1 \ p \ \Delta \text{neg} \ \Delta \text{mint-msg} \ \Delta \text{mint-self}$
and *InvCapsNonneg* $c0$
shows *InvTempJustified* $c1$
 $\langle \text{proof} \rangle$

lemma *next'-preserves-InvTempJustified*:

$\text{InvCapsNonneg } c0 \implies \text{InvTempJustified } c0 \implies \text{next}' \ c0 \ c1 \implies \text{InvTempJustified } c1$
 $\langle \text{proof} \rangle$

lemma *alw-InvTempJustified*: $\text{spec } s \implies \text{alw } (\text{holds } \text{InvTempJustified}) \ s$

$\langle \text{proof} \rangle$

4.3.7 InvGlobNonposImpRecordsNonpos

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

definition *InvGlobNonposImpRecordsNonpos* :: $(p :: \text{finite}, 'a)$ configuration $\implies \text{bool}$ **where**

$\text{InvGlobNonposImpRecordsNonpos } c = (\forall t \ q. \text{nonpos-upto } (c\text{-glob } c \ q) \ t \longrightarrow \text{nonpos-upto } (\text{records } c) \ t)$

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

lemma $\text{invs-imp-InvGlobNonposImpRecordsNonpos}$:
assumes $\text{InvJustifiedGII } c$
and $\text{InvRecordCount } c$
and $\text{InvRecordsNonneg } c$
shows $\text{InvGlobNonposImpRecordsNonpos } c$
 $\langle \text{proof} \rangle$

$\text{InvGlobVacantImpRecordsVacant}$ is the one proved in the Abadi paper. We prove $\text{InvGlobNonposImpRecordsNonpos}$, which implies this.

lemma $\text{invs-imp-InvGlobVacantImpRecordsVacant}$:
assumes $\text{InvJustifiedGII } c$
and $\text{InvRecordCount } c$
and $\text{InvRecordsNonneg } c$
shows $\text{InvGlobVacantImpRecordsVacant } c$
 $\langle \text{proof} \rangle$

lemma $\text{alw-InvGlobNonposImpRecordsNonpos}$: $\text{spec } s \Longrightarrow \text{alw } (\text{holds } \text{InvGlobNonposImpRecordsNonpos}) \ s$
 $\langle \text{proof} \rangle$

lemma $\text{alw-InvGlobVacantImpRecordsVacant}$: $\text{spec } s \Longrightarrow \text{alw } (\text{holds } \text{InvGlobVacantImpRecordsVacant}) \ s$
 $\langle \text{proof} \rangle$

4.3.8 SafeGlobVacantUptoImpliesStickyNrec

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

lemma $\text{invs-imp-SafeGlobVacantUptoImpliesStickyNrec}$:
 $\text{SafeRecordsMono } s \Longrightarrow \text{holds } \text{InvGlobVacantImpRecordsVacant } s \Longrightarrow \text{SafeGlobVacantUptoImpliesStickyNrec } s$
 $\langle \text{proof} \rangle$

lemma $\text{alw-SafeGlobVacantUptoImpliesStickyNrec}$:
 $\text{spec } s \Longrightarrow \text{alw } \text{SafeGlobVacantUptoImpliesStickyNrec } s$
 $\langle \text{proof} \rangle$

4.3.9 InvGlobNonposEqVacant

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

definition $\text{InvGlobNonposEqVacant}$ **where**

$InvGlobNonposEqVacant\ c = (\forall\ q\ t.\ GlobVacantUpto\ c\ q\ t = GlobNonposUpto\ c\ q\ t)$

lemma *invs-imp-InvGlobNonposEqVacant*:

assumes *InvRecordCount* c
and *InvJustifiedGII* c
and *InvRecordsNonneg* c
shows *InvGlobNonposEqVacant* c
 \langle proof \rangle

lemma *alw-InvGlobNonposEqVacant*: $spec\ s \implies alw\ (holds\ InvGlobNonposEqVacant)\ s$

\langle proof \rangle

4.3.10 InvInfoJustifiedWithII and InvInfoJustifiedWithGII

definition *InvInfoJustifiedWithII* **where**

$InvInfoJustifiedWithII\ c = (\forall\ k\ p\ q.\ justified-with\ (c-caps\ c\ p)\ (InfoAt\ c\ k\ p\ q)\ (IncomingInfo\ c\ (k+1)\ p\ q))$

definition *InvInfoJustifiedWithGII* **where**

$InvInfoJustifiedWithGII\ c = (\forall\ k\ p\ q.\ justified-with\ (records\ c)\ (InfoAt\ c\ k\ p\ q)\ (GlobalIncomingInfo\ c\ (k+1)\ p\ q))$

lemma *init-config-implies-InvInfoJustifiedWithII*: $init-config\ c \implies InvInfoJustifiedWithII\ c$

\langle proof \rangle

This proof relies heavily on the addition properties summarized in the lemma $\llbracket justified-with\ (c-caps\ ?c0.0\ ?p)\ ?M\ ?N; next-performop'\ ?c0.0\ ?c1.0\ ?p\ ?\Delta neg\ ?\Delta mint-msg\ ?\Delta mint-self; \forall\ t.\ 0 \leq zcount\ (c-caps\ ?c0.0\ ?p)\ t; justified\ (c-caps\ ?c0.0\ ?p)\ ?N \rrbracket \implies justified-with\ (c-caps\ ?c0.0\ ?p + zmsset-of\ ?\Delta mint-self - zmsset-of\ ?\Delta neg)\ ?M\ (?N + zmsset-of\ ?\Delta mint-self + times-tamps\ (zmsset-of\ ?\Delta mint-msg) - zmsset-of\ ?\Delta neg)$

lemma *performop-preserves-InvInfoJustifiedWithII*:

assumes *InvInfoJustifiedWithII* $c0$
and *next-performop'* $c0\ c1\ p'\ \Delta neg\ \Delta mint-msg\ \Delta mint-self$
and *InvJustifiedII* $c0$
and *InvCapsNonneg* $c0$
shows *InvInfoJustifiedWithII* $c1$
 \langle proof \rangle

lemma *sendupd-preserves-InvInfoJustifiedWithII*:

assumes *InvInfoJustifiedWithII* $c0$
and *next-sendupd'* $c0\ c1\ p'\ tt$
and *InvTempJustified* $c0$
shows *InvInfoJustifiedWithII* $c1$
 \langle proof \rangle

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 \implies alw (holds InvInfoJustifiedWithII)*

s

<proof>

lemma *alw-InvInfoJustifiedWithGII: spec s \implies 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 = ($\forall p q t. c\text{-msg } c p q \neq [] \longrightarrow t \in \#_z \text{hd } (c\text{-msg } c p q) \longrightarrow (\exists t' \leq t. 0 < zcount (c\text{-glob } c q) t')$)

lemma *not-InvMsgInGlob-imp-not-SafeGlobMono:*

assumes $\neg \text{InvMsgInGlob } c0$

and $\text{InvGlobNonposEqVacant } c0$

shows $\exists c1. \text{next-recvupd } c0 \ c1 \wedge \neg \text{SafeGlobMono } c0 \ c1$

<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-preserved-GlobVacantUpto:*

assumes $\text{GlobVacantUpto } c0 \ q \ t$

and $\text{next-recvupd}' \ c0 \ c1 \ p \ q$

and $\text{InvInfoJustifiedWithGII } c0$

and $\text{InvGlobNonposEqVacant } c1$

and $\text{InvGlobVacantImpRecordsVacant } c0$

and $\text{InvRecordCount } c0$

shows $\text{GlobVacantUpto } c1 \ q \ t$

<proof>

lemma *recvupd-imp-SafeGlobMono:*

assumes $\text{next-recvupd}' \ c0 \ c1 \ p \ q$

and $\text{InvInfoJustifiedWithGII } c0$

and $\text{InvGlobNonposEqVacant } c1$

and $\text{InvGlobVacantImpRecordsVacant } c0$

and $\text{InvRecordCount } c0$

shows $\text{SafeGlobMono } c0 \ c1$

<proof>

lemma *next'-imp-SafeGlobMono:*

assumes $\text{next}' \ c0 \ c1$

and $\text{InvInfoJustifiedWithGII } c0$

and $\text{InvGlobNonposEqVacant } c1$

and $\text{InvGlobVacantImpRecordsVacant } c0$

and $\text{InvRecordCount } c0$

shows $\text{SafeGlobMono } c0 \ c1$

<proof>

lemma *invs-imp-InvMsgInGlob:*

fixes $c0 :: ('p::\text{finite}, 'a) \text{ configuration}$

assumes $\text{InvInfoJustifiedWithGII } c0$

and $\text{InvGlobNonposEqVacant } c0$

and $\text{InvGlobVacantImpRecordsVacant } c0$

and $\text{InvRecordCount } c0$

and $\text{InvJustifiedII } c0$

and $\text{InvCapsNonneg } c0$

and $\text{InvRecordsNonneg } c0$

shows $\text{InvMsgInGlob } c0$

<proof>

lemma *alw-SafeGlobMono*: $\text{spec } s \implies \text{alw } (\text{relates SafeGlobMono}) s$
 ⟨proof⟩

lemma *alw-InvMsgInGlob*: $\text{spec } s \implies \text{alw } (\text{holds InvMsgInGlob}) s$
 ⟨proof⟩

lemma *SafeGlobMono-preserves-vacant*:
assumes $\forall t' \leq t. \text{zcount } (c\text{-glob } c0 \ q) \ t' = 0$
and $(\lambda c0 \ c1. \text{SafeGlobMono } c0 \ c1)^{**} \ c0 \ c1$
shows $\forall t' \leq t. \text{zcount } (c\text{-glob } c1 \ q) \ t' = 0$
 ⟨proof⟩

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

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

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

lemma *InvCapsNonneg-imp-InvRecordsNonneg*: $\text{InvCapsNonneg } c \implies \text{InvRecordsNonneg } c$
 ⟨proof⟩

lemma *invs-imp-msg-in-glob*:
fixes $c :: ('p::\text{finite}, 'a) \text{ configuration}$
assumes $M \in \text{set } (c\text{-msg } c \ p \ q)$
and $t \in \#_z \ M$
and $\text{InvGlobNonposEqVacant } c$
and $\text{InvJustifiedII } c$
and $\text{InvInfoJustifiedWithII } c$
and $\text{InvGlobVacantImpRecordsVacant } c$
and $\text{InvRecordCount } c$
and $\text{InvCapsNonneg } c$
and $\text{InvMsgInGlob } c$
shows $\exists t' \leq t. 0 < \text{zcount } (c\text{-glob } c \ q) \ t'$
 ⟨proof⟩

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

end

5 Antichains

definition *incomparable* **where**

$$\text{incomparable } A = (\forall x \in A. \forall y \in A. x \neq y \longrightarrow \neg x < y \wedge \neg y < x)$$

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 (⟨(-/ ∈_A -)⟩
 [51, 51] 50) **is** Set.member ⟨proof⟩

abbreviation *not-member-antichain* :: 'a :: order ⇒ 'a antichain ⇒ bool (⟨(-/ ∉_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]: $s \in_A A \implies A \neq \{ }_A$
 ⟨proof⟩

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

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

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

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

lemma *finite-minimal-antichain*[simp]: $\text{finite } A \implies \text{finite } (\text{minimal-antichain } A)$
 ⟨proof⟩

lemma *finite-set-antichain*[simp, intro]: $\text{finite } (\text{set-antichain } A)$
 ⟨proof⟩

lemma *minimal-antichain-subset*: $\text{minimal-antichain } A \subseteq A$
 ⟨proof⟩

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

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

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

lemma *minimal-antichain-idempotent[simp]*: $\text{minimal-antichain } (\text{minimal-antichain } A) = \text{minimal-antichain } A$
 ⟨proof⟩

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

instantiation *antichain* :: (order) plus **begin**
lift-definition *plus-antichain* :: 'a antichain \Rightarrow 'a antichain \Rightarrow 'a antichain **is** λM
 $N. \text{minimal-antichain } (M \cup N)$
 ⟨proof⟩
instance ⟨proof⟩
end

lemma *antichain-add-commute*: $(M :: 'a :: order \text{antichain}) + N = N + M$
 ⟨proof⟩

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

syntax (ASCII)
 -ACCollect :: pptrn \Rightarrow 'a :: order antichain \Rightarrow bool \Rightarrow 'a antichain ($\langle(1\{- :_A \ -/ \ -\})\rangle$)
syntax
 -ACCollect :: pptrn \Rightarrow 'a :: order antichain \Rightarrow bool \Rightarrow 'a antichain ($\langle(1\{- \in_A \ -/ \ -\})\rangle$)
syntax-consts
 -ACCollect == *filter-antichain*

translations
 $\{x \in_A M. P\} == \text{CONST } \text{filter-antichain } (\lambda x. P) \ M$

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 \implies (t::- ::order) \in A \implies *minimal-antichain* A \neq {}
(*proof*)

lemma *minimal-antichain-member*:
finite A \implies (t::- ::order) \in A \implies $\exists t'. t' \in$ *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*: $c \in_A A - B \iff c \in_A A \wedge c \notin_A B$
(*proof*)

lemma *ac-DiffD2*: $c \in_A A - B \implies c \in_A B \implies P$
(*proof*)

lemma *ac-notin-Diff*: $\neg x \in_A A - B \implies \neg x \in_A A \vee x \in_A B$
(*proof*)

lemma *ac-eq-iff*: $A = B \iff (\forall x. x \in_A A \iff x \in_A B)$
(*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 $\implies x \in_A X$
(*proof*)

lemma *set-antichain2*[simp]: $x \in_A X \implies x \in$ *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**

$LBL \equiv \lambda(s, l, t). l$

abbreviation (*input*) *TO* **where**

$TO \equiv \lambda(s, l, t). t$

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

locale *graph* =

fixes *weights* :: 'vtx :: finite \Rightarrow 'vtx \Rightarrow 'lbl :: {order, monoid-add} antichain

assumes *zero-le[simp]*: $0 \leq (s::'lbl)$

and *plus-mono*: $(s1::'lbl) \leq s2 \implies s3 \leq s4 \implies s1 + s3 \leq s2 + s4$

and *summary-self*: $weights\ loc\ loc = \{ \}_A$

begin

lemma *le-plus*: $(s::'lbl) \leq s + s' \ (s'::'lbl) \leq s + s'$

<proof>

6.1 Paths

inductive *path* :: 'vtx \Rightarrow 'vtx \Rightarrow ('vtx \times 'lbl \times 'vtx) list \Rightarrow bool **where**

path0: $l1 = l2 \implies path\ l1\ l2\ []$

| *path*: $path\ l1\ l2\ xs \implies lbl \in_A\ weights\ l2\ l3 \implies path\ l1\ l3\ (xs @ [(l2, lbl, l3)])$

inductive-cases *path0E*: $path\ l1\ l2\ []$

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

lemma *path-trans*: $path\ l1\ l2\ xs \implies path\ l2\ l3\ ys \implies path\ l1\ l3\ (xs @ ys)$

<proof>

lemma *path-take-from*: $path\ l1\ l2\ xs \implies m < length\ xs \implies FROM\ (xs ! m) = l2' \implies path\ l1\ l2'\ (take\ m\ xs)$

<proof>

lemma *path-take-to*: $path\ l1\ l2\ xs \implies m < length\ xs \implies TO\ (xs ! m) = l2' \implies path\ l1\ l2'\ (take\ (m+1)\ xs)$

<proof>

lemma *path-determines-loc*: $path\ l1\ l2\ xs \implies path\ l1\ l3\ xs \implies l2 = l3$

<proof>

lemma *path-first-loc*: $path\ loc\ loc'\ xs \implies xs \neq [] \implies FROM\ (xs ! 0) = loc$

<proof>

lemma *path-to-eq-from*: $path\ loc1\ loc2\ xs \implies i + 1 < length\ xs \implies FROM\ (xs ! (i+1)) = TO\ (xs ! i)$

<proof>

lemma *path-singleton*[*intro, simp*]: $s \in_A \text{weights } l1 \ l2 \implies \text{path } l1 \ l2 \ [(l1, s, l2)]$
 ⟨*proof*⟩

lemma *path-appendE*: $\text{path } l1 \ l3 \ (xs \ @ \ ys) \implies \exists l2. \text{path } l2 \ l3 \ ys \wedge \text{path } l1 \ l2 \ xs$
 ⟨*proof*⟩

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

lemma *drop-subseq*: $n \leq \text{length } xs \implies \text{drop } n \ xs \preceq xs$
 ⟨*proof*⟩

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

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

lemma *path-distinct*:
 $\text{path } l1 \ l2 \ xs \implies \exists xs'. \text{distinct } xs' \wedge \text{path } l1 \ l2 \ xs' \wedge \text{map } LBL \ xs' \preceq \text{map } LBL \ xs$
 ⟨*proof*⟩

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

6.2 Path Weights

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

abbreviation *sum-path-weights* $xs \equiv \text{sum-weights } (\text{map } LBL \ xs)$

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

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

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

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

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

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

<proof>

lemma *sum-summary-prepend-le*: $\text{sum-path-weights } ys \leq \text{sum-path-weights } xs \implies \text{sum-path-weights } (zs @ ys) \leq \text{sum-path-weights } (zs @ xs)$
<proof>

lemma *sum-summary-append-le*: $\text{sum-path-weights } ys \leq \text{sum-path-weights } xs \implies \text{sum-path-weights } (ys @ zs) \leq \text{sum-path-weights } (xs @ zs)$
<proof>

lemma *foldr-plus-zero-le*: $\text{foldr } (+) \text{ } xs \ (0::'lbl) \leq \text{foldr } (+) \text{ } xs \ a$
<proof>

lemma *subseq-sum-weights-le*:
assumes $xs \preceq ys$
shows $\text{sum-weights } xs \leq \text{sum-weights } ys$
<proof>

lemma *subseq-sum-path-weights-le*:
 $\text{map } LBL \ xs \preceq \text{map } LBL \ ys \implies \text{sum-path-weights } xs \leq \text{sum-path-weights } ys$
<proof>

lemma *sum-path-weights-take-le*[*simp, intro*]: $\text{sum-path-weights } (\text{take } i \ xs) \leq \text{sum-path-weights } xs$
<proof>

lemma *sum-weights-append-singleton*:
 $\text{sum-weights } (xs @ [x]) = \text{sum-weights } xs + x$
<proof>

lemma *sum-path-weights-append-singleton*:
 $\text{sum-path-weights } (xs @ [(l,x,l')]) = \text{sum-path-weights } xs + x$
<proof>

lemma *path-weightp-ex-path*:
 $\text{path-weightp } l1 \ l2 \ s \implies \exists \ xs.$
 $(\text{let } s' = \text{sum-path-weights } xs \text{ in } s' \leq s \wedge \text{path-weightp } l1 \ l2 \ s' \wedge \text{distinct } xs \wedge$
 $(\forall (l1,s,l2) \in \text{set } xs. s \in_A \text{weights } l1 \ l2))$
<proof>

lemma *finite-set-summaries*:
 $\text{finite } ((\lambda(l1,l2),s). (l1,s,l2)) \text{ ' } (\text{Sigma } UNIV \ (\lambda(l1,l2). \text{set-antichain } (\text{weights } l1 \ l2))))$
<proof>

lemma *finite-summaries*: $\text{finite } \{xs. \text{distinct } xs \wedge (\forall (l1, s, l2) \in \text{set } xs. s \in_A \text{weights } l1 \ l2)\}$
<proof>

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

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

definition *reachable* *l1* *l2* $\equiv \text{path-weight } l1\ l2 \neq \{\}_A$

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

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

lemma *zero-in-minimal-antichain[simp]*: $(0::'lbl) \in S \implies 0 \in \text{minimal-antichain } S$
 ⟨*proof*⟩

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

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*} $\neq \{\}$ \longleftrightarrow {*xs*. *path-weightp-distinct* *l1* *l2* *xs*} $\neq \{\}$
 ⟨*proof*⟩

lemma *path-weightp-distinct-member*:
 $s \in \{s. \text{path-weightp } l1\ l2\ s\} \implies \exists u. u \in \{s. \text{path-weightp-distinct } l1\ l2\ s\} \wedge u \leq s$
 ⟨*proof*⟩

lemma *minimal-antichain-path-weightp-member*:
 $s \in \{xs. \text{path-weightp } l1\ l2\ xs\} \implies \exists u. u \in \text{minimal-antichain } \{xs. \text{path-weightp } l1\ l2\ xs\} \wedge u \leq s$
 ⟨*proof*⟩

lemma *path-path-weight*: $\text{path } l1\ l2\ xs \implies \exists s. s \in_A \text{path-weight } l1\ l2 \wedge s \leq$

sum-path-weights xs
(proof)

lemma *path-weight-conv-path*:

$s \in_A \text{path-weight } l1 \ l2 \implies \exists xs. \text{path } l1 \ l2 \ xs \wedge s = \text{sum-path-weights } xs \wedge (\forall ys. \text{path } l1 \ l2 \ ys \longrightarrow \neg \text{sum-path-weights } ys < \text{sum-path-weights } xs)$
(proof)

abbreviation *optimal-path loc1 loc2 xs* $\equiv \text{path } loc1 \ loc2 \ xs \wedge$

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

lemma *path-weight-path*: $s \in_A \text{path-weight } loc1 \ loc2 \implies$

$(\bigwedge xs. \text{optimal-path } loc1 \ loc2 \ xs \implies \text{distinct } xs \implies \text{sum-path-weights } xs = s \implies P) \implies P$
(proof)

lemma *path-weight-elem-trans*:

$s \in_A \text{path-weight } l1 \ l2 \implies s' \in_A \text{path-weight } l2 \ l3 \implies \exists u. u \in_A \text{path-weight } l1 \ l3 \wedge u \leq s + s'$
(proof)

end

7 Local Progress Propagation

7.1 Specification

record (overloaded) ('loc, 't) *configuration* =

c-work :: 'loc \Rightarrow 't *zmultiset*

c-pts :: 'loc \Rightarrow 't *zmultiset*

c-imp :: 'loc \Rightarrow 't *zmultiset*

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

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

for *summary* :: 'loc \Rightarrow 'loc :: *finite* \Rightarrow 'sum :: {*order*, *monoid-add*} *antichain* +

fixes *results-in* :: 't :: *order* \Rightarrow 'sum \Rightarrow 't

assumes *results-in-zero*: *results-in* t 0 = t

and *results-in-mono-raw*: $t1 \leq t2 \implies s1 \leq s2 \implies \text{results-in } t1 \ s1 \leq \text{results-in } t2 \ s2$

and *followed-by-summary*: $\text{results-in } (\text{results-in } t \ s1) \ s2 = \text{results-in } t \ (s1 + s2)$

and *no-zero-cycle*: $\text{path } loc \ loc \ xs \implies xs \neq [] \implies s = \text{sum-path-weights } xs \implies t < \text{results-in } t \ s$

begin

lemma *results-in-mono*:

$t1 \leq t2 \implies \text{results-in } t1 \ s \leq \text{results-in } t2 \ s$

$s1 \leq s2 \implies \text{results-in } t \ s1 \leq \text{results-in } t \ s2$
 ⟨proof⟩

abbreviation *path-summary* \equiv *path-weight*

abbreviation *followed-by* $:: 'sum \Rightarrow 'sum \Rightarrow 'sum$ **where**
followed-by \equiv *plus*

definition *safe* $:: ('loc, 't)$ configuration \Rightarrow *bool* **where**

safe $c \equiv \forall loc1 \ loc2 \ t \ s. \text{zcount } (c\text{-pts } c \ loc1) \ t > 0 \wedge s \in_A \text{path-summary } loc1$
 $loc2$
 $\longrightarrow (\exists t' \leq \text{results-in } t \ s. t' \in_A \text{frontier } (c\text{-imp } c \ loc2))$

Implications are always non-negative.

definition *inv-implications-nonneg* **where**

inv-implications-nonneg $c = (\forall loc \ t. \text{zcount } (c\text{-imp } c \ loc) \ t \geq 0)$

abbreviation *unchanged* $f \ c0 \ c1 \equiv f \ c1 = f \ c0$

abbreviation *zmsset-frontier* **where**

zmsset-frontier $M \equiv \text{zmsset-of } (\text{mset-set } (\text{set-antichain } (\text{frontier } M)))$

definition *init-config* **where**

init-config $c \equiv \forall loc.$
 $c\text{-imp } c \ loc = \{\#\}_z \wedge$
 $c\text{-work } c \ loc = \text{zmsset-frontier } (c\text{-pts } c \ loc)$

definition *after-summary* $:: 't \ \text{zmultiset} \Rightarrow 'sum \ \text{antichain} \Rightarrow 't \ \text{zmultiset}$ **where**

after-summary $M \ S \equiv (\sum s \in \text{set-antichain } S. \text{image-zmsset } (\lambda t. \text{results-in } t \ s)$
 $M)$

abbreviation *frontier-changes* $:: 't \ \text{zmultiset} \Rightarrow 't \ \text{zmultiset} \Rightarrow 't \ \text{zmultiset}$ **where**

frontier-changes $M \ N \equiv \text{zmsset-frontier } M - \text{zmsset-frontier } N$

definition *next-change-multiplicity'* $:: ('loc, 't)$ configuration $\Rightarrow ('loc, 't)$ configuration $\Rightarrow 'loc \Rightarrow 't \Rightarrow \text{int} \Rightarrow \text{bool}$ **where**

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

abbreviation *next-change-multiplicity* $:: ('loc, 't)$ configuration $\Rightarrow ('loc, 't)$ configuration $\Rightarrow \text{bool}$ **where**

$next\text{-change}\text{-multiplicity}\ c0\ c1 \equiv \exists loc\ t\ n. next\text{-change}\text{-multiplicity}'\ c0\ c1\ loc\ t\ n$

lemma *cm-unchanged-worklist*:

assumes $next\text{-change}\text{-multiplicity}'\ c0\ c1\ loc\ t\ n$

and $loc' \neq loc$

shows $c\text{-work}\ c1\ loc' = c\text{-work}\ c0\ loc'$

<proof>

definition $next\text{-propagate}' :: ('loc, 't)\ configuration \Rightarrow ('loc, 't)\ configuration \Rightarrow$

$'loc \Rightarrow 't \Rightarrow bool$ **where**

$next\text{-propagate}'\ c0\ c1\ loc\ t \equiv$

— t is a least timestamp of all worklist entries

$(t \in \#_z\ c\text{-work}\ c0\ loc \wedge$

$(\forall t' loc'. t' \in \#_z\ c\text{-work}\ c0\ loc' \longrightarrow \neg t' < t) \wedge$

$c1 = c0 \langle c\text{-imp} := (c\text{-imp}\ c0)(loc := c\text{-imp}\ c0\ loc + \{\#t' \in \#_z\ c\text{-work}\ c0\ loc. t' = t\ \#}\rangle,$

$c\text{-work} := (\lambda loc'.$

— worklist entries for t are removed from loc 's worklist

$if\ loc' = loc\ then\ \{\#t' \in \#_z\ c\text{-work}\ c0\ loc'. t' \neq t\ \#}$

— worklists at other locations change by the loc 's frontier change

after adding summaries

$else\ c\text{-work}\ c0\ loc'$

$+ after\text{-summary}$

$(frontier\text{-changes}\ (c\text{-imp}\ c0\ loc + \{\#t' \in \#_z\ c\text{-work}\ c0$

$loc. t' = t\ \#})\ (c\text{-imp}\ c0\ loc))$

$(summary\ loc\ loc'))))$

abbreviation $next\text{-propagate} :: ('loc, 't :: order)\ configuration \Rightarrow ('loc, 't)\ confi-$

$guration \Rightarrow bool$ **where**

$next\text{-propagate}\ c0\ c1 \equiv \exists loc\ t. next\text{-propagate}'\ c0\ c1\ loc\ t$

definition $next'$ **where**

$next'\ c0\ c1 = (next\text{-propagate}\ c0\ c1 \vee next\text{-change}\text{-multiplicity}\ c0\ c1 \vee c1 = c0)$

abbreviation $next$ **where**

$next\ s \equiv next'\ (shd\ s)\ (shd\ (stl\ s))$

abbreviation $cm\text{-valid}$ **where**

$cm\text{-valid} \equiv nxt\ (\lambda s. next\text{-change}\text{-multiplicity}\ (shd\ s)\ (shd\ (stl\ s)))\ impl$

$(\lambda s. next\text{-change}\text{-multiplicity}\ (shd\ s)\ (shd\ (stl\ s)))\ or\ nxt\ (holds\ (\lambda c.$

$(\forall l. c\text{-work}\ c\ l = \{\#\}_z))$

definition $spec :: ('loc, 't :: order)\ computation \Rightarrow bool$ **where**

$spec \equiv holds\ init\text{-config}\ aand\ alw\ next$

lemma $next'\text{-inv}$ [*consumes 1, case-names next-change-multiplicity next-propagate next-finish-init*]:

assumes $next'\ c0\ c1\ P\ c0$

and $\bigwedge loc\ t\ n. P\ c0 \implies next\text{-change}\text{-multiplicity}'\ c0\ c1\ loc\ t\ n \implies P\ c1$

and $\bigwedge \text{loc } t. P \text{ c}0 \implies \text{next-propagate}' \text{ c}0 \text{ c}1 \text{ loc } t \implies P \text{ c}1$
shows $P \text{ c}1$
 $\langle \text{proof} \rangle$

7.2 Auxiliary

lemma *next-change-multiplicity'-unique*:
assumes $n \neq 0$
and $\exists t'. t' \in_A \text{frontier } (c\text{-imp } c \text{ loc}) \wedge t' \leq t$
shows $\exists !c'. \text{next-change-multiplicity}' c \text{ c}' \text{ loc } t \ n$
 $\langle \text{proof} \rangle$

lemma *frontier-change-zmset-frontier*:
assumes $t \in_A \text{frontier } M1 - \text{frontier } M0$
shows $\text{zcount } (\text{zmset-frontier } M1) \ t = 1 \wedge \text{zcount } (\text{zmset-frontier } M0) \ t = 0$
 $\langle \text{proof} \rangle$

lemma *frontier-empty[simp]*: $\text{frontier } \{\#\}_z = \{\}_A$
 $\langle \text{proof} \rangle$

lemma *zmset-frontier-empty[simp]*: $\text{zmset-frontier } \{\#\}_z = \{\#\}_z$
 $\langle \text{proof} \rangle$

lemma *after-summary-empty[simp]*: $\text{after-summary } \{\#\}_z \ S = \{\#\}_z$
 $\langle \text{proof} \rangle$

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

lemma *mem-frontier-diff*:
assumes $t \in_A \text{frontier } M - \text{frontier } N$
shows $\text{zcount } (\text{frontier-changes } M \ N) \ t = 1$
 $\langle \text{proof} \rangle$

lemma *mem-frontier-diff'*:
assumes $t \in_A \text{frontier } N - \text{frontier } M$
shows $\text{zcount } (\text{frontier-changes } M \ N) \ t = -1$
 $\langle \text{proof} \rangle$

lemma *not-mem-frontier-diff*:
assumes $t \notin_A \text{frontier } M - \text{frontier } N$
and $t \notin_A \text{frontier } N - \text{frontier } M$
shows $\text{zcount } (\text{frontier-changes } M \ N) \ t = 0$
 $\langle \text{proof} \rangle$

lemma *mset-neg-after-summary*: $\text{mset-neg } M = \{\#\} \implies \text{mset-neg } (\text{after-summary } M \ S) = \{\#\}$
 $\langle \text{proof} \rangle$

lemma *next-p-frontier-change*:

assumes $\text{next-propagate}' c0 c1 loc t$
and $\text{summary } loc loc' \neq \{\}_A$
shows $c\text{-work } c1 loc' =$
 $c\text{-work } c0 loc'$
 $+ \text{after-summary}$
 $(\text{frontier-changes } (c\text{-imp } c1 loc) (c\text{-imp } c0 loc))$
 $(\text{summary } loc loc')$
 $\langle \text{proof} \rangle$

lemma $\text{after-summary-union}$: $\text{after-summary } (M + N) S = \text{after-summary } M S$
 $+ \text{after-summary } N S$
 $\langle \text{proof} \rangle$

7.3 Invariants

7.3.1 Invariant: inv-imps-work-sum

abbreviation $\text{union-frontiers} :: ('loc, 't) \text{ configuration} \Rightarrow 'loc \Rightarrow 't \text{ zmultiset}$ **where**
 $\text{union-frontiers } c loc \equiv$
 $(\sum loc' \in UNIV. \text{after-summary } (\text{zmsset-frontier } (c\text{-imp } c loc')) (\text{summary } loc' loc))$

— Implications + worklist is equal to the frontiers of pointstamps and all preceding nodes (after accounting for summaries).

definition $\text{inv-imps-work-sum} :: ('loc, 't) \text{ configuration} \Rightarrow \text{bool}$ **where**
 $\text{inv-imps-work-sum } c \equiv$
 $\forall loc. c\text{-imp } c loc + c\text{-work } c loc$
 $= \text{zmsset-frontier } (c\text{-pts } c loc) + \text{union-frontiers } c loc$

— Version with zcount is easier to reason with

definition $\text{inv-imps-work-sum-zcount} :: ('loc, 't) \text{ configuration} \Rightarrow \text{bool}$ **where**
 $\text{inv-imps-work-sum-zcount } c \equiv$
 $\forall loc t. \text{zcount } (c\text{-imp } c loc + c\text{-work } c loc) t$
 $= \text{zcount } (\text{zmsset-frontier } (c\text{-pts } c loc) + \text{union-frontiers } c loc) t$

lemma $\text{inv-imps-work-sum-zcount}$: $\text{inv-imps-work-sum } c \longleftrightarrow \text{inv-imps-work-sum-zcount } c$
 $\langle \text{proof} \rangle$

lemma $\text{union-frontiers-nonneg}$: $0 \leq \text{zcount } (\text{union-frontiers } c loc) t$
 $\langle \text{proof} \rangle$

lemma $\text{next-p-union-frontier-change}$:

assumes $\text{next-propagate}' c0 c1 loc t$
and $\text{summary } loc loc' \neq \{\}_A$
shows $\text{union-frontiers } c1 loc' =$
 $\text{union-frontiers } c0 loc'$
 $+ \text{after-summary}$
 $(\text{frontier-changes } (c\text{-imp } c1 loc) (c\text{-imp } c0 loc))$

(summary loc loc')

⟨proof⟩

lemma *init-imp-inv-imps-work-sum*: *init-config c* \implies *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 *next (holds inv-imps-work-sum) s*
 ⟨proof⟩

lemma *spec-imp-iiws*: *spec s* \implies *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 $\equiv \forall \text{loc } t. 0 \leq \text{zcount } (c\text{-imp } c \text{ loc}) t + \text{zcount } (c\text{-work } c \text{ loc}) t$

lemma *iiws-imp-iiwn*:
assumes *inv-imps-work-sum c*
shows *inv-imp-plus-work-nonneg c*
 ⟨proof⟩

lemma *spec-imp-iiwn*: *spec s* \implies *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 *next (holds inv-implications-nonneg)* s
 ⟨*proof*⟩

lemma *alw-inv-implications-nonneg*: *spec* $s \implies$ *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* $M \longleftrightarrow x \in_A$ *frontier* M

⟨*proof*⟩

lemma *obtain-frontier-elem*:

assumes $0 < zcount$ M t
obtains u **where** $u \in_A$ *frontier* M $u \leq t$
 ⟨*proof*⟩

lemma *frontier-unionD*: $t \in_A$ *frontier* $(M+N) \implies 0 < zcount$ M $t \vee 0 < zcount$ N t

⟨*proof*⟩

lemma *ps-frontier-in-imps-wl*:

assumes *inv-imps-work-sum* c
and $0 < zcount$ (*zmset-frontier* $(c\text{-pts}$ 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** $s \leq t \wedge s \in_A$ *frontier* M
 ⟨*proof*⟩

lemma *obtain-elem-zmset-frontier*:

assumes $0 < \text{zcount } M \ t$

obtains s **where** $s \leq t \wedge 0 < \text{zcount } (\text{zmset-frontier } M) \ s$

<proof>

lemma *ps-in-imps-wl*:

assumes *inv-imps-work-sum* c

and $0 < \text{zcount } (c\text{-pts } c \ \text{loc}) \ t$

obtains s **where** $s \leq t \wedge 0 < \text{zcount } (c\text{-imp } c \ \text{loc} + c\text{-work } c \ \text{loc}) \ s$

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *pos-zcount-after-summary*:

$(\bigwedge t. 0 \leq \text{zcount } M \ t) \implies 0 < \text{zcount } M \ t \implies s \in_A \ S \implies 0 < \text{zcount } (\text{after-summary } M \ S) \ (\text{results-in } t \ s)$

<proof>

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

<proof>

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

<proof>

lemma *pos-zcount-union-frontiers*:

$\text{zcount } (\text{after-summary } (\text{zmset-frontier } (c\text{-imp } c \ l1)) \ (\text{summary } l1 \ l2)) \ (\text{results-in } t \ s)$

$\leq \text{zcount } (\text{union-frontiers } c \ l2) \ (\text{results-in } t \ s)$

<proof>

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

<proof>

lemma *after-summary-obtain-pre*:

assumes $\bigwedge t. 0 \leq \text{zcount } M \ t$

and $0 < \text{zcount } (\text{after-summary } M \ S) \ t$

obtains $t' s$ **where** $0 < \text{zcount } M t' \text{ results-in } t' s = t s \in_A S$
 ⟨proof⟩

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

definition *impWitnessPath* **where**

$\text{impWitnessPath } c \text{ loc1 } \text{loc2 } xs \ t = ($
 $\text{path } \text{loc1 } \text{loc2 } xs \wedge$
 $\text{distinct } xs \wedge$
 $(\exists t'. t' \in_A \text{frontier } (c\text{-imp } c \text{ loc1}) \wedge t = \text{results-in } t' (\text{sum-path-weights } xs) \wedge$
 $(\forall k < \text{length } xs. (\exists t. t \in_A \text{frontier } (c\text{-imp } c (TO (xs ! k))) \wedge t = \text{results-in } t'$
 $(\text{sum-path-weights } (\text{take } (k+1) xs))))))$

lemma *impWitnessPathEx*:

assumes $t \in_A \text{frontier } (c\text{-imp } c \text{ loc2})$
shows $(\exists \text{loc1 } xs. \text{impWitnessPath } c \text{ loc1 } \text{loc2 } xs \ t)$
 ⟨proof⟩

definition *longestImpWitnessPath* **where**

$\text{longestImpWitnessPath } c \text{ loc1 } \text{loc2 } xs \ t = ($
 $\text{impWitnessPath } c \text{ loc1 } \text{loc2 } xs \ t \wedge$
 $(\forall \text{loc}' \ xs'. \text{impWitnessPath } c \text{ loc}' \ \text{loc2 } \ xs' \ t \longrightarrow \text{length } (xs') \leq \text{length } (xs)))$

lemma *finite-edges*: $\text{finite } \{(\text{loc1}, s, \text{loc2}). s \in_A \text{summary } \text{loc1 } \text{loc2}\}$
 ⟨proof⟩

lemma *longestImpWitnessPathEx*:

assumes $t \in_A \text{frontier } (c\text{-imp } c \text{ loc2})$
shows $(\exists \text{loc1 } xs. \text{longestImpWitnessPath } c \text{ loc1 } \text{loc2 } xs \ t)$
 ⟨proof⟩

lemma *path-first-loc*: $\text{path } l1 \ l2 \ xs \implies xs \neq [] \implies xs ! 0 = (l1', s, l2') \implies l1 = l1'$
 ⟨proof⟩

lemma *find-witness-from-frontier*:

assumes $t \in_A \text{frontier } (c\text{-imp } c \text{ loc2})$
and $\text{inv-imps-work-sum } c$
shows $\exists t' \text{loc1 } xs. (\text{path } \text{loc1 } \text{loc2 } xs \wedge t = \text{results-in } t' (\text{sum-path-weights } xs) \wedge$
 $(t' \in_A \text{frontier } (c\text{-pts } c \text{ loc1}) \vee 0 > \text{zcount } (c\text{-work } c \text{ loc1}) \ t'))$
 ⟨proof⟩

lemma *implication-implies-pointstamp*:

assumes $t \in_A \text{frontier } (c\text{-imp } c \text{ loc})$
and $\text{inv-imps-work-sum } c$
shows $\exists t' \text{loc}' \ s. s \in_A \text{path-summary } \text{loc}' \ \text{loc} \wedge t \geq \text{results-in } t' \ s \wedge$
 $(t' \in_A \text{frontier } (c\text{-pts } c \ \text{loc}') \vee 0 > \text{zcount } (c\text{-work } c \ \text{loc}') \ t')$
 ⟨proof⟩

7.4 Proof of Safety

lemma *results-in-sum-path-weights-append*:

$results\text{-}in\ t\ (sum\text{-}path\text{-}weights\ (xs\ @\ [(loc2,\ s,\ loc3)])) = results\text{-}in\ (results\text{-}in\ t\ (sum\text{-}path\text{-}weights\ xs))\ s$
 ⟨proof⟩

context

fixes $c :: ('loc,\ 't)\ configuration$

begin

inductive *loc-imps-fw* **where**

$loc\text{-}imps\text{-}fw\ loc\ loc\ (c\text{-}imp\ c\ loc)\ []\ |$
 $loc\text{-}imps\text{-}fw\ loc1\ loc2\ M\ xs \implies s \in_A\ summary\ loc2\ loc3 \implies distinct\ (xs\ @\ [(loc2,\ s,\ loc3)]) \implies$
 $loc\text{-}imps\text{-}fw\ loc1\ loc3\ (\{\#\ results\text{-}in\ t\ s.\ t \in_{\#z}\ M\ \#\} + c\text{-}imp\ c\ loc3)\ (xs\ @\ [(loc2,\ s,\ loc3)])$

end

lemma *loc-imps-fw-conv-path*: $loc\text{-}imps\text{-}fw\ c\ loc1\ loc2\ M\ xs \implies path\ loc1\ loc2\ xs$
 ⟨proof⟩

lemma *path-conv-loc-imps-fw*: $path\ loc1\ loc2\ xs \implies distinct\ xs \implies \exists M.\ loc\text{-}imps\text{-}fw\ c\ loc1\ loc2\ M\ xs$
 ⟨proof⟩

lemma *path-summary-conv-loc-imps-fw*:

$s \in_A\ path\text{-}summary\ loc1\ loc2 \implies \exists M\ xs.\ loc\text{-}imps\text{-}fw\ c\ loc1\ loc2\ M\ xs \wedge sum\text{-}path\text{-}weights\ xs = s$
 ⟨proof⟩

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

lemma *sum-pos*: $finite\ M \implies \forall x \in M.\ 0 \leq f\ x \implies y \in M \implies 0 < (f\ y :: ordered\text{-}comm\text{-}monoid\text{-}add) \implies 0 < (\sum x \in M.\ f\ x)$
 ⟨proof⟩

lemma *loc-imps-fw-M-in-implications*:

assumes $loc\text{-}imps\text{-}fw\ c\ loc1\ loc2\ M\ xs$

and $inv\text{-}imps\text{-}work\text{-}sum\ c$

and $inv\text{-}implications\text{-}nonneg\ c$

and $\bigwedge loc.\ c\text{-}work\ c\ loc = \{\#\}_z$

and $0 < zcount\ M\ t$

shows $\exists s.\ s \leq t \wedge s \in_A\ frontier\ (c\text{-}imp\ c\ loc2)$

⟨proof⟩

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

assumes $loc\text{-}imps\text{-}fw\ c\ loc1\ loc2\ M\ xs$

and *inv-implications-nonneg* c
shows $0 \leq \text{zcount } M \ t$
 $\langle \text{proof} \rangle$

lemma *loc-imps-fw-implication-in-M*:
assumes *inv-imps-work-sum* c
and *inv-implications-nonneg* c
and *loc-imps-fw* $c \ \text{loc1} \ \text{loc2} \ M \ xs$
and $0 < \text{zcount} (c\text{-imp } c \ \text{loc1}) \ t$
shows $0 < \text{zcount } M \ (\text{results-in } t \ (\text{sum-path-weights } xs))$
 $\langle \text{proof} \rangle$

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

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

lemma *cm-preserves-impl-safe*:
assumes *impl-safe* $c0$
and *next-change-multiplicity'* $c0 \ c1 \ \text{loc} \ t \ n$
shows *impl-safe* $c1$
 $\langle \text{proof} \rangle$

lemma *cm-preserves-safe*:
assumes *safe* $c0$
and *impl-safe* $c0$
and *next-change-multiplicity'* $c0 \ c1 \ \text{loc} \ t \ n$
shows *safe* $c1$
 $\langle \text{proof} \rangle$

7.5 A Better (More Invariant) Safety

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

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

Intuition: Unlike *safe*, *inv-safe* is an invariant because it only claims the safety property $t' \in_A \text{frontier}(c\text{-imp } c \text{ loc}2)$ 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*: $\forall y \in \#_z N. \neg y \leq x \implies x \in_A \text{frontier}(M - N) \longleftrightarrow x \in_A \text{frontier } M$
 ⟨proof⟩

lemma *worklists-vacant-to-trans*:
 $\text{worklists-vacant-to } c \ t \implies t' \leq t \implies \text{worklists-vacant-to } c \ t'$
 ⟨proof⟩

lemma *loc-imps-fw-M-in-implications'*:
assumes *loc-imps-fw* $c \ \text{loc}1 \ \text{loc}2 \ M \ xs$
and *inv-imps-work-sum* c
and *inv-implications-nonneg* c
and *worklists-vacant-to* $c \ t$
and $0 < \text{zcount } M \ t$
shows $\exists s \leq t. s \in_A \text{frontier}(c\text{-imp } c \ \text{loc}2)$
 ⟨proof⟩

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

lemma *alw-conjI*: $\text{alw } P \ s \implies \text{alw } Q \ s \implies \text{alw } (\lambda s. P \ s \wedge Q \ s) \ s$
 ⟨proof⟩

lemma *alw-inv-safe*: $\text{spec } s \implies \text{alw } (\text{holds } \text{inv-safe}) \ s$
 ⟨proof⟩

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

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

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

7.6 Implied Frontier

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

definition *implied-frontier* where

$implied\text{-}frontier\ P\ loc = frontier\ (\sum\ loc' \in UNIV.\ after\text{-}summary\ (zmset\text{-}pos\ (P\ loc'))\ (path\text{-}summary\ loc'\ loc))$

definition *implied-frontier-alt* where

$implied\text{-}frontier\text{-}alt\ c\ loc = frontier\ (\sum\ loc' \in UNIV.\ after\text{-}summary\ (zmset\text{-}frontier\ (c\text{-}pts\ c\ loc'))\ (path\text{-}summary\ loc'\ loc))$

lemma *in-frontier-least*: $x \in_A\ frontier\ M \implies \forall y.\ 0 < zcount\ M\ y \longrightarrow \neg y < x$
 ⟨proof⟩

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

lemma *implied-frontier-alt-least*:

assumes $b \in_A\ implied\text{-}frontier\text{-}alt\ c\ loc2$

shows $\forall loc\ a'\ s'.\ a' \in_A\ frontier\ (c\text{-}pts\ c\ loc) \longrightarrow s' \in_A\ path\text{-}summary\ loc\ loc2 \longrightarrow \neg\ results\text{-}in\ a'\ s' < b$

⟨proof⟩

lemma *implied-frontier-alt-in-pointstamps*:

assumes $b \in_A\ implied\text{-}frontier\text{-}alt\ c\ loc2$

obtains $a\ s\ loc1$ where

$a \in_A\ frontier\ (c\text{-}pts\ c\ loc1)\ s \in_A\ path\text{-}summary\ loc1\ loc2\ results\text{-}in\ a\ s = b$

⟨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 $b \in_A\ implied\text{-}frontier\text{-}alt\ c\ loc2$

shows $b \in_A\ frontier\ (c\text{-}imp\ c\ loc2)$

⟨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 $b \in_A\ frontier\ (c\text{-}imp\ c\ loc2)$

shows $b \in_A\ implied\text{-}frontier\text{-}alt\ c\ loc2$

⟨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 $b \in_A \text{frontier } (c\text{-imp } c\ \text{loc}) \longleftrightarrow b \in_A \text{implied-frontier-alt } c\ \text{loc}$
<proof>

lemma *next-propagate-implied-frontier-alt-def*:
 $\text{next-propagate } c\ c' \implies \text{implied-frontier-alt } c\ \text{loc} = \text{implied-frontier-alt } c'\ \text{loc}$
<proof>

lemma *implication-frontier-eq-implied-frontier-alt*:
assumes *inv-imps-work-sum* c
and *inv-implications-nonneg* c
and $\bigwedge \text{loc. } c\text{-work } c\ \text{loc} = \{\#\}_z$
shows $\text{frontier } (c\text{-imp } c\ \text{loc}) = \text{implied-frontier-alt } c\ \text{loc}$
<proof>

lemma *alw-implication-frontier-eq-implied-frontier-alt-empty*: $\text{spec } s \implies$
 $\text{alw } (\text{holds } (\lambda c. (\forall \text{loc. } c\text{-work } c\ \text{loc} = \{\#\}_z) \longrightarrow \text{frontier } (c\text{-imp } c\ \text{loc}) = \text{implied-frontier-alt } c\ \text{loc}))\ s$
<proof>

lemma *alw-implication-frontier-eq-implied-frontier-alt-vacant*: $\text{spec } s \implies$
 $\text{alw } (\text{holds } (\lambda c. \text{worklists-vacant-to } c\ b \longrightarrow b \in_A \text{frontier } (c\text{-imp } c\ \text{loc}) \longleftrightarrow b \in_A \text{implied-frontier-alt } c\ \text{loc}))\ s$
<proof>

lemma *antichain-eqI*: $(\bigwedge b. b \in_A A \longleftrightarrow b \in_A B) \implies A = B$
<proof>

lemma *zmset-frontier-zmset-pos*: $\text{zmset-frontier } A \subseteq_{\#_z} \text{zmset-pos } A$
<proof>

lemma *image-mset-mono-pos*:
 $\forall b. 0 \leq \text{zcount } A\ b \implies \forall b. 0 \leq \text{zcount } B\ b \implies A \subseteq_{\#_z} B \implies \text{image-zmset } f\ A \subseteq_{\#_z} \text{image-zmset } f\ B$
<proof>

lemma *sum-mono-subseteq*:
 $(\bigwedge i. i \in K \implies f\ i \subseteq_{\#_z} g\ i) \implies (\sum i \in K. f\ i) \subseteq_{\#_z} (\sum i \in K. g\ i)$
<proof>

lemma *after-summary-zmset-frontier*:
 $\text{after-summary } (\text{zmset-frontier } A)\ S \subseteq_{\#_z} \text{after-summary } (\text{zmset-pos } A)\ S$
<proof>

lemma *frontier-eqI*: $\forall b. 0 \leq \text{zcount } A\ b \implies \forall b. 0 \leq \text{zcount } B\ b \implies$
 $A \subseteq_{\#_z} B \implies (\bigwedge b. b \in_{\#_z} B \implies \exists a. a \in_{\#_z} A \wedge a \leq b) \implies \text{frontier } A = \text{frontier } B$
<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 $P (Finite-Set.fold f z M)$

<proof>

8.1 Could-result-in Relation

context *dataflow-topology* **begin**

definition *cri-less-eq* :: $('loc \times 't) \Rightarrow ('loc \times 't) \Rightarrow bool$ ($\hookrightarrow \leq_p \rightarrow [51,51]$ 50) **where**
cri-less-eq =

$(\lambda(loc1,t1) (loc2,t2). (\exists s. s \in_A path-summary loc1 loc2 \wedge results-in t1 s \leq t2))$

definition *cri-less* :: $('loc \times 't) \Rightarrow ('loc \times 't) \Rightarrow bool$ ($\hookrightarrow <_p \rightarrow [51,51]$ 50) **where**
cri-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 = (λloc'.
 (if loc' = loc then update-zmultiset (c-pts c loc') t n
 else c-pts c loc')) in
 c (| c-pts := new-pointstamps |
 (| c-work :=
 (λloc'. c-work c loc' + frontier-changes (new-pointstamps loc') (c-pts c
 loc')))))

definition cm-all' **where**

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

definition cm-all **where**

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

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

8.2.2 Initial state and state transitions

definition InitConfig :: ('p::finite, 't::order, 'loc) configuration ⇒ bool **where**

InitConfig c =
 ((∀ p. init c p = False)
 ∧ cri.init-config (exchange-config c)
 ∧ (∀ p loc t. zcount (c-pts (prop-config c p) loc) t
 = zcount (c-glob (exchange-config c) p) (loc, t))

$\wedge (\forall w. \text{init-config } (\text{prop-config } c \ w))$

definition $\text{NextPerformOp}' :: ('p::\text{finite}, 't::\text{order}, 'loc) \text{configuration} \Rightarrow ('p, 't, 'loc) \text{configuration}$
 $\Rightarrow 'p \Rightarrow ('loc \times 't) \text{multiset} \Rightarrow ('p \times ('loc \times 't)) \text{multiset}$
 $\Rightarrow ('loc \times 't) \text{multiset} \Rightarrow \text{bool}$ **where**
 $\text{NextPerformOp}' \ c0 \ c1 \ p \ \Delta\text{neg} \ \Delta\text{mint-msg} \ \Delta\text{mint-self} =$
 $\text{cri.next-performop}' (\text{exchange-config } c0) (\text{exchange-config } c1) \ p \ \Delta\text{neg} \ \Delta\text{mint-msg}$
 $\Delta\text{mint-self}$
 $\wedge \text{unchanged prop-config } c0 \ c1$
 $\wedge \text{unchanged init } c0 \ c1$

abbreviation NextPerformOp **where**
 $\text{NextPerformOp} \ c0 \ c1 \equiv \exists p \ \Delta\text{neg} \ \Delta\text{mint-msg} \ \Delta\text{mint-self}. \text{NextPerformOp}' \ c0$
 $c1 \ p \ \Delta\text{neg} \ \Delta\text{mint-msg} \ \Delta\text{mint-self}$

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

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

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

abbreviation NextSendUpd **where**
 $\text{NextSendUpd} \ c0 \ c1 \equiv \exists p \ tt. \text{NextSendUpd}' \ c0 \ c1 \ p \ tt$

definition $\text{NextRecvUpd}' :: ('p::\text{finite}, 't::\text{order}, 'loc) \text{configuration} \Rightarrow ('p, 't, 'loc)$
 configuration
 $\Rightarrow 'p \Rightarrow 'p \Rightarrow \text{bool}$ **where**
 $\text{NextRecvUpd}' \ c0 \ c1 \ p \ q =$
 $\text{init } c0 \ q$ — Once init is set we are guaranteed that the CM transitions' premises
are satisfied
 $\wedge \text{cri.next-recvupd}' (\text{exchange-config } c0) (\text{exchange-config } c1) \ p \ q$
 $\wedge \text{unchanged init } c0 \ c1$
 $\wedge (\forall p'. \text{prop-config } c1 \ p' =$
 $(\text{if } p' = q$
 $\text{then cm-all } (\text{prop-config } c0 \ q) (\text{hd } (c\text{-msg } (\text{exchange-config } c0) \ p \ q)))$

else prop-config c0 p'))

abbreviation *NextRecvUpd* **where**

NextRecvUpd c0 c1 $\equiv \exists p q. \text{NextRecvUpd}' c0 c1 p q$

definition *NextPropagate'* :: (*'p*::finite, *'t*::order, *'loc*) configuration \Rightarrow (*'p*, *'t*, *'loc*) configuration

\Rightarrow *'p* \Rightarrow bool **where**

NextPropagate' *c0 c1 p* = (
unchanged exchange-config c0 c1
 \wedge *init c1* = (*init c0*)(*p* := True)
 \wedge ($\forall p'. \text{Some}(\text{prop-config } c1 p') =$
 (*if* *p'* = *p*
 then *propagate-all* (*prop-config c0 p'*)
 else *Some* (*prop-config c0 p'*)))

abbreviation *NextPropagate* **where**

NextPropagate c0 c1 $\equiv \exists p. \text{NextPropagate}' c0 c1 p$

definition *Next'* **where**

Next' c0 c1 = (*NextPerformOp c0 c1* \vee *NextSendUpd c0 c1* \vee *NextRecvUpd c0 c1* \vee *NextPropagate c0 c1* \vee *NextRecvCap c0 c1* \vee *c1 = c0*)

abbreviation *Next* **where**

Next s $\equiv \text{Next}'(\text{shd } s)(\text{shd }(\text{stl } s))$

definition *FullSpec* :: (*'p* :: finite, *'t* :: order, *'loc*) computation \Rightarrow bool **where**

FullSpec s = (*holds InitConfig s* \wedge *alw Next s*)

lemma *NextPerformOpD*:

assumes *NextPerformOp'* *c0 c1 p* $\Delta_{\text{neg}} \Delta_{\text{mint-msg}} \Delta_{\text{mint-self}}$

shows

cri.next-performop' (*exchange-config c0*) (*exchange-config c1*) *p* $\Delta_{\text{neg}} \Delta_{\text{mint-msg}} \Delta_{\text{mint-self}}$

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
 $(\forall p'. \text{prop-config } c1 \ p' =$
 (if p' = q
 then cm-all (prop-config c0 q) (hd (c-msg (exchange-config c0) p q))
 else prop-config c0 p'))
\langle proof \rangle

lemma *NextPropagateD*:
assumes *NextPropagate' c0 c1 p*
shows
 unchanged exchange-config c0 c1
 init c1 = (init c0)(p := True)
 $(\forall p'. \text{Some (prop-config } c1 \ p') =$
 (if p' = p
 then propagate-all (prop-config c0 p')
 else Some (prop-config c0 p')))
\langle proof \rangle

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
\langle proof \rangle

8.3 Auxiliary Lemmas

8.3.1 Auxiliary Lemmas for CM Conversion

lemma *apply-cm-is-cm*:
 $\exists t'. t' \in_A \text{frontier } (c\text{-imp } c \text{ loc}) \wedge t' \leq t \implies n \neq 0 \implies \text{next-change-multiplicity}'$
 $c \text{ (apply-cm } c \text{ loc } t \ n) \text{ loc } t \ n$
\langle proof \rangle

lemma *update-zmultiset-commute*:
 $\text{update-zmultiset } (\text{update-zmultiset } M \ t' \ n') \ t \ n = \text{update-zmultiset } (\text{update-zmultiset}$
 $M \ t \ n) \ t' \ n'$
\langle proof \rangle

lemma *apply-cm-commute*: $\text{apply-cm } (\text{apply-cm } c \text{ loc } t \ n) \ \text{loc}' \ t' \ n' = \text{apply-cm}$
 $(\text{apply-cm } c \ \text{loc}' \ t' \ n') \ \text{loc} \ t \ n$
\langle proof \rangle

lemma *comp-fun-commute-apply-cm[simp]*: $\text{comp-fun-commute } (\lambda(\text{loc}, t) \ c. \ \text{apply-cm}$
 $\text{apply-cm } c \ \text{loc} \ t \ (f \ \text{loc} \ t))$
\langle proof \rangle

lemma *ex-cm-imp-conds*:

assumes $\exists c'. \text{next-change-multiplicity}' c c' \text{ loc } t n$
shows $\exists t'. t' \in_A \text{frontier } (c\text{-imp } c \text{ loc}) \wedge t' \leq t n \neq 0$
<proof>

lemma *the-cm-eq-apply-cm*:

assumes $\exists c'. \text{next-change-multiplicity}' c c' \text{ loc } t n$
shows $\text{the-cm } c \text{ loc } t n = \text{apply-cm } c \text{ loc } t n$
<proof>

lemma *apply-cm-preserves-cond*:

assumes $\forall (loc, t) \in \text{set-zmset } \Delta. \exists t'. t' \in_A \text{frontier } (c\text{-imp } c0 \text{ loc}) \wedge t' \leq t$
shows $\forall (loc, t) \in \text{set-zmset } \Delta. \exists t'. t' \in_A \text{frontier } (c\text{-imp } (\text{apply-cm } c0 \text{ loc}' t'' n) \text{ loc}) \wedge t' \leq t$
<proof>

lemma *cm-all-eq-cm-all'*:

assumes $\forall (loc, t) \in \text{set-zmset } \Delta. \exists t'. t' \in_A \text{frontier } (c\text{-imp } c0 \text{ loc}) \wedge t' \leq t$
shows $\text{cm-all } c0 \Delta = \text{cm-all}' c0 \Delta$
<proof>

lemma *cm-eq-the-cm*:

assumes $\text{next-change-multiplicity}' c c' \text{ loc } t n$
shows $\text{the-cm } c \text{ loc } t n = c'$
<proof>

lemma *zcount-ps-apply-cm*:

$\text{zcount } (c\text{-pts } (\text{apply-cm } c \text{ loc } t n) \text{ loc}') t' = \text{zcount } (c\text{-pts } c \text{ loc}') t' + (\text{if } \text{loc} = \text{loc}' \wedge t = t' \text{ then } n \text{ else } 0)$
<proof>

lemma *zcount-pointstamps-update*: $\text{zcount } (c\text{-pts } (c\{c\text{-pts}:=M\}) \text{ loc}) x = \text{zcount } (M \text{ loc}) x$
<proof>

lemma *nop*: $\text{loc1} \neq \text{loc2} \vee t1 \neq t2 \longrightarrow$

$\text{zcount } (c\text{-pts } (\text{apply-cm } c \text{ loc2 } t2 (\text{zcount } \Delta (\text{loc2}, t2))) \text{ loc1}) t1 = \text{zcount } (c\text{-pts } c \text{ loc1}) t1$
<proof>

lemma *fold-nop*:

$\text{zcount } (c\text{-pts } (\text{Finite-Set.fold } (\lambda(\text{loc}', t') c. \text{apply-cm } c \text{ loc}' t' (\text{zcount } \Delta' (\text{loc}', t')))) c$
 $\quad (\text{set-zmset } \Delta - \{(\text{loc}, t)\}) \text{ loc}) t$
 $= \text{zcount } (c\text{-pts } c \text{ loc}) t$
<proof>

lemma *zcount-pointstamps-cm-all'*:

$\text{zcount } (c\text{-pts } (\text{cm-all}' c \Delta) \text{ loc}) x$

= 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-imp (cm-all' c Δ) = c-imp c
 ⟨proof⟩

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

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

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

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

lemma *propagate-all-imp-InvGlobPointstampsEq*:
 Some c1 = propagate-all c0 \implies c-pts c0 = c-pts c1
 ⟨proof⟩

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

lemma *lift-propagate-inv-propagate-all*:

assumes $(\bigwedge c0\ c1\ loc\ t.\ P\ c0 \implies next-propagate'\ c0\ c1\ loc\ t \implies P\ c1)$
and $P\ c0$
and $propagate-all\ c0 = Some\ c1$
shows $P\ c1$
 $\langle proof \rangle$

8.4 Exchange is a Subsystem of Tracker

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

lemma *next-imp-exchange-next*:

$Next'\ c0\ c1 \implies cri.next'\ (exchange-config\ c0)\ (exchange-config\ c1)$
 $\langle proof \rangle$

lemma *alw-next-imp-exchange-next*: $alw\ Next\ s \implies alw\ cri.next\ (smap\ exchange-config\ s)$

$\langle proof \rangle$

Any Tracker trace is a valid Exchange trace

lemma *spec-imp-exchange-spec*: $FullSpec\ s \implies cri.spec\ (smap\ exchange-config\ s)$

$\langle proof \rangle$

lemma *lift-exchange-invariant*:

assumes $\bigwedge s.\ cri.spec\ s \implies alw\ (holds\ P)\ s$

shows $FullSpec\ s \implies alw\ (\lambda s.\ P\ (exchange-config\ (shd\ s)))\ s$

$\langle proof \rangle$

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**

$exch-alw-InvGlobNonposImpRecordsNonpos = lift-exchange-invariant[OF\ cri.alw-InvGlobNonposImpRecordsNon$

$simplified\ atomize-imp,\ simplified,\ folded\ atomize-imp]$ **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 \Rightarrow bool **where**
safe-combined c $\equiv \forall$ loc1 loc2 t s p.
 zcount (cri.records (exchange-config c)) (loc1, t) > 0 \wedge s \in_A path-summary
 loc1 loc2 \wedge init c p
 $\longrightarrow (\exists t'. t' \in_A \text{frontier } (c\text{-imp } (\text{prop-config } c \ p) \ \text{loc2}) \wedge t' \leq \text{results-in } t \ s)$

definition *safe-combined2* :: ('p::finite, 't::order, 'loc) configuration \Rightarrow bool **where**
safe-combined2 c $\equiv \forall$ loc1 loc2 t s p1 p2.
 zcount (c-caps (exchange-config c) p1) (loc1, t) > 0 \wedge s \in_A path-summary
 loc1 loc2 \wedge init c p2
 $\longrightarrow (\exists t'. t' \in_A \text{frontier } (c\text{-imp } (\text{prop-config } c \ p2) \ \text{loc2}) \wedge t' \leq \text{results-in } t \ s)$

definition *InvGlobPointstampsEq* :: ('p :: finite, 't :: order, 'loc) configuration \Rightarrow bool **where**
InvGlobPointstampsEq c = (
 $(\forall p \ \text{loc} \ t. \text{zcount } (c\text{-pts } (\text{prop-config } c \ p) \ \text{loc}) \ t$
 $= \text{zcount } (c\text{-glob } (\text{exchange-config } c) \ p) \ (\text{loc}, \ t))$)

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 = ($\forall p \ q.$
 init c q \longrightarrow c-msg (exchange-config c) p q $\neq \square \longrightarrow$
 $(\forall (\text{loc}, t) \in \#_z (\text{hd } (c\text{-msg } (\text{exchange-config } c) \ p \ q)). \exists t'. t' \in_A \text{frontier } (c\text{-imp } (\text{prop-config } c \ q) \ \text{loc}) \wedge t' \leq t)$)

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 $\neq \square$
shows $\forall (\text{loc}, t) \in \#_z (\text{hd } (c\text{-msg } (\text{exchange-config } c) \ p \ q)). \exists t'. t' \in_A \text{frontier } (c\text{-imp } (\text{prop-config } c \ q) \ \text{loc}) \wedge t' \leq t$
 <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 $c \equiv \text{inv-implications-nonneg } c \wedge \text{inv-imps-work-sum } c$

abbreviation *prop-safe* **where**

prop-safe $c \equiv \text{impl-safe } c \wedge \text{safe } c$

definition *inv-init-imp-prop-safe* **where**

inv-init-imp-prop-safe $c = (\forall p. \text{init } c \ p \longrightarrow \text{prop-safe } (\text{prop-config } c \ p))$

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* $c0 \ q$) \wedge *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 $\forall p.$ *prop-invs* (*prop-config c0 p*)
and *InvGlobPointstampsEq c0*
and *NextRecvUpd' c0 c1 p q*
shows $\forall p.$ *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 $\forall p.$ *prop-invs* (*prop-config c0 p*)
shows
inv-init-imp-prop-safe c1
 $\forall p.$ *prop-invs* (*prop-config c1 p*)
InvGlobPointstampsEq c1
 ⟨*proof*⟩

lemma *init-imp-InvGlobPointstampsEq: InitConfig c \implies InvGlobPointstampsEq*

c
 $\langle proof \rangle$

lemma *init-imp-inv-init-imp-prop-safe*: $InitConfig\ c \implies inv-init-imp-prop-safe\ c$
 $\langle proof \rangle$

lemma *init-imp-prop-invs*: $InitConfig\ c \implies \forall p. prop-invs\ (prop-config\ c\ p)$
 $\langle proof \rangle$

abbreviation *all-invs where*

$all-invs\ c \equiv InvGlobPointstampsEq\ c \wedge inv-init-imp-prop-safe\ c \wedge (\forall p. prop-invs\ (prop-config\ c\ p))$

lemma *alw-Next'-alw-invs*:
assumes *holds all-invs s*
and $alw\ (holds\ (\lambda c. cri.InvMsgInGlob\ (exchange-config\ c)))\ s$
and $alw\ Next\ s$
shows $alw\ (holds\ all-invs)\ s$
 $\langle proof \rangle$

lemma *alw-invs*: $FullSpec\ s \implies alw\ (holds\ all-invs)\ s$
 $\langle proof \rangle$

lemma *alw-InvGlobPointstampsEq*: $FullSpec\ s \implies alw\ (holds\ InvGlobPointstampsEq)\ s$
 $\langle proof \rangle$

lemma *alw-inv-init-imp-prop-safe*: $FullSpec\ s \implies alw\ (holds\ inv-init-imp-prop-safe)\ s$
 $\langle proof \rangle$

lemma *alw-holds-conv-shd*: $alw\ (holds\ \varphi)\ s = alw\ (\lambda s. \varphi\ (shd\ s))\ s$
 $\langle proof \rangle$

lemma *alw-prop-invs*: $FullSpec\ s \implies alw\ (holds\ (\lambda c. \forall p. prop-invs\ (prop-config\ c\ p)))\ s$
 $\langle proof \rangle$

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

lemma *help-lemma*:
assumes $0 < zcount\ (c-pts\ (prop-config\ c\ p)\ loc0)\ t0$
and $(loc0, t0) \leq_p\ (loc1, t1)$
and $s2 \in_A\ path-summary\ loc1\ loc2$
and $safe\ (prop-config\ c\ p)$

shows $\exists t2. (t2 \leq \text{results-in } t1 \ s2$
 $\quad \wedge t2 \in_A \text{frontier } (c\text{-imp } (\text{prop-config } c \ p) \ \text{loc2}))$
 $\langle \text{proof} \rangle$
lemma *lift-prop-inv-NextPropagate'*:
assumes $(\bigwedge c0 \ c1 \ \text{loc } t. P \ c0 \implies \text{next-propagate}' \ c0 \ c1 \ \text{loc } t \implies P \ c1)$
shows $P \ (\text{prop-config } c0 \ p') \implies \text{NextPropagate}' \ c0 \ c1 \ p \implies P \ (\text{prop-config } c1 \ p')$
 $\langle \text{proof} \rangle$

8.6.3 Propagate is a Subsystem

lemma *NextRecvUpd'-next'*:
assumes *safe* $(\text{prop-config } c0 \ q)$
and *InvGlobPointstampsEq* $c0$
and *cri.InvMsgInGlob* $(\text{exchange-config } c0)$
and *NextRecvUpd'* $c0 \ c1 \ p \ q$
shows $\text{next}^{+++} \ (\text{prop-config } c0 \ q') \ (\text{prop-config } c1 \ q')$
 $\langle \text{proof} \rangle$

lemma *NextPropagate'-next'*:
assumes *NextPropagate'* $c0 \ c1 \ p$
shows $\text{next}^{+++} \ (\text{prop-config } c0 \ q) \ (\text{prop-config } c1 \ q)$
 $\langle \text{proof} \rangle$

lemma *next-imp-propagate-next*:
assumes *inv-init-imp-prop-safe* $c0$
and *InvGlobPointstampsEq* $c0$
and *cri.InvMsgInGlob* $(\text{exchange-config } c0)$
shows $\text{Next}' \ c0 \ c1 \implies \text{next}^{+++} \ (\text{prop-config } c0 \ p) \ (\text{prop-config } c1 \ p)$
 $\langle \text{proof} \rangle$

lemma *alw-next-imp-propagate-next*:
assumes *alw* $(\text{holds } \text{inv-init-imp-prop-safe}) \ s$
and *alw* $(\text{holds } \text{InvGlobPointstampsEq}) \ s$
and *alw* $(\text{holds } \text{cri.InvMsgInGlob}) \ (\text{smap } \text{exchange-config } s)$
and *alw* $\text{Next} \ s$
shows *alw* $(\text{relates } (\text{next}^{+++})) \ (\text{smap } (\lambda s. \text{prop-config } s \ p) \ s)$
 $\langle \text{proof} \rangle$

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*: $\text{FullSpec } s \implies (\text{holds } \text{init-config } a \ \text{and } \text{alw } (\text{relates } (\text{next}^{+++}))) \ (\text{smap } (\lambda c. \text{prop-config } c \ p) \ s)$
 $\langle \text{proof} \rangle$

8.7 Safety Proofs

lemma *safe-satisfied*:
assumes *cri.InvGlobNonposImpRecordsNonpos* $(\text{exchange-config } c)$

and *inv-init-imp-prop-safe* c
and *InvGlobPointstampsEq* c
shows *safe-combined* c
 \langle *proof* \rangle

lemma *alw-safe-combined*: $FullSpec\ s \implies alw\ (holds\ safe-combined)\ s$
 \langle *proof* \rangle

lemma *alw-safe-combined2*: $FullSpec\ s \implies alw\ (holds\ safe-combined2)\ s$
 \langle *proof* \rangle

lemma *alw-implication-frontier-eq-IMPLIED-frontier*:
 $FullSpec\ s \implies$
 $alw\ (holds\ (\lambda c.\ worklists-vacant-to\ (prop-config\ c\ p)\ b \longrightarrow$
 $b \in_A\ frontier\ (c-imp\ (prop-config\ c\ p)\ loc) \longleftrightarrow b \in_A\ IMPLIED-frontier\ (c-pts$
 $(prop-config\ c\ p))\ loc))\ s$
 \langle *proof* \rangle

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.