Hybrid Multi-Lane Spatial Logic

Sven Linker

May 26, 2024

Abstract

We present a semantic embedding of a spatio-temporal multi-modal logic, specifically defined to reason about motorway traffic, into Isabelle/HOL. The semantic model is an abstraction of a motorway, emphasising local spatial properties, and parameterised by the types of sensors deployed in the vehicles. We use the logic to define controller constraints to ensure safety, i.e., the absence of collisions on the motorway. After proving safety with a restrictive definition of sensors, we relax these assumptions and show how to amend the controller constraints to still guarantee safety.

Published in iFM 2017 [4].

Formal verification of autonomous vehicles on motorways is a challenging problem, due to the complex interactions between dynamical behaviours and controller choices of the vehicles. To overcome the complexities of proving safety properties, we proposed to separate the dynamical behaviour from the concrete changes in space [2]. To that end, we defined *Multi-Lane Spatial Logic* (MLSL), which was used to express guards and invariants of controller automata defining a protocol for safe lane-change manoeuvres. Under the assumption that all vehicles adhere to this protocol, we proved that collisions were avoided. Subsequently, we presented an extension of MLSL to reason about changes in space over time, a system of natural deduction, and formally proved a safety theorem [5, 3]. This proof was carried out manually and dependent on strong assumptions about the vehicles' sensors.

We define a semantic embedding of a further extension of MLSL, inspired by Hybrid Logic [1]. Subsequently, we show how the safety theorem can be proved within this embedding. Finally, we alter this formal embedding by relaxing the assumptions on the sensors. We show that the previously proven safety theorem does *not* ensure safety in this case, and how the controller constraints can be strengthened to guarantee safety.

Contents

1 Discrete Intervals based on Natural Numbers

We define a type of intervals based on the natural numbers. To that end, we employ standard operators of Isabelle, but in addition prove some structural properties of the intervals. In particular, we show that this type constitutes a meet-semilattice with a bottom element and equality.

Furthermore, we show that this semilattice allows for a constrained join, i.e., the union of two intervals is defined, if either one of them is empty, or they are consecutive. Finally, we define the notion of *chopping* an interval into two consecutive subintervals.

theory *NatInt*
imports *Main*
begin

A discrete interval is a set of consecutive natural numbers, or the empty set.

typedef *nat-int* = $\{S\ .\ (\exists\ (m{::}nat)\ n\ .\ \{m..n\}{=}S)\ \}$
⟨*proof*⟩
setup-lifting *type-definition-nat-int*

1.1 Basic properties of discrete intervals.

locale *nat-int*
interpretation *nat-int-class?*: *nat-int* ⟨*proof*⟩

context *nat-int*
begin

lemma *un-consec-seq*: $(m{::}nat){\leq}\ n \wedge n{+}1 \leq l \longrightarrow \{m..n\} \cup \{n{+}1..l\} = \{m..l\}$
⟨*proof*⟩

lemma *int-conseq-seq*: $\{(m{::}nat)..n\} \cap \{n{+}1..l\} = \{\}$
⟨*proof*⟩

lemma *empty-type*: $\{\} \in \{\ S\ .\ \exists\ (m{::}\ nat)\ n\ .\ \{m..n\}{=}S\}$
⟨*proof*⟩

lemma *inter-result*: $\forall\, x \in \{S\ .\ (\exists\ (m{::}nat)\ n\ .\ \{m..n\}{=}S)\ \}.$
$\forall\, y \in \{S\ .\ (\exists\ (m{::}nat)\ n\ .\ \{m..n\}{=}S)\ \}.$
$x \cap y \in \{S\ .\ (\exists\ (m{::}nat)\ n\ .\ \{m..n\}{=}S)\}$
⟨*proof*⟩

lemma *union-result*: $\forall\, x \in \{S\ .\ (\exists\ (m{::}nat)\ n\ .\ \{m..n\}{=}S)\ \}.$
$\forall\, y \in \{S\ .\ (\exists\ (m{::}nat)\ n\ .\ \{m..n\}{=}S)\ \}.$
$x \neq \{\} \wedge y \neq \{\} \wedge Max\ x + 1 = Min\ y$
$\longrightarrow x \cup y \in \{S\ .\ (\exists\ (m{::}nat)\ n\ .\ \{m..n\}{=}S)\ \}$
⟨*proof*⟩

lemma *union-empty-result1*: $\forall\, i \in \{S\ .\ (\exists\ (m{::}nat)\ n\ .\ \{m..n\}{=}S)\ \}.$
$i \cup \{\} \in \{S\ .\ (\exists\ (m{::}nat)\ n\ .\ \{m..n\}{=}S)\ \}$
⟨*proof*⟩

lemma *union-empty-result2*: $\forall\, i \in \{S\ .\ (\exists\ (m{::}nat)\ n\ .\ \{m..n\}{=}S)\ \}.$
$\{\} \cup i \in \{S\ .\ (\exists\ (m{::}nat)\ n\ .\ \{m..n\}{=}S)\ \}$
⟨*proof*⟩

lemma *finite*: $\forall\, i \in \{S\ .\ (\exists\ (m{::}nat)\ n\ .\ \{m..n\}{=}S)\ \}\ .\ (finite\ i)$
⟨*proof*⟩

lemma *not-empty-means-seq*:$\forall\, i \in \{S\ .\ (\exists\ (m{::}nat)\ n\ .\ \{m..n\}{=}S)\ \}\ .\ i \neq \{\}$
$\longrightarrow (\ \exists\, m\ n\ .\ m \leq n \wedge \{m..n\} = i)$
$\langle proof \rangle$
end

The empty set is the bottom element of the type. The infimum/meet of the semilattice is set intersection. The order is given by the subset relation.

instantiation *nat-int* :: *bot*
begin
lift-definition *bot-nat-int* :: *nat-int* **is** *Set.empty* $\langle proof \rangle$
instance $\langle proof \rangle$
end

instantiation *nat-int* :: *inf*
begin
lift-definition *inf-nat-int* ::*nat-int* $\Rightarrow$ *nat-int* $\Rightarrow$ *nat-int* **is** *Set.inter* $\langle proof \rangle$
instance
$\langle proof \rangle$
end

instantiation *nat-int* :: *order-bot*
begin
lift-definition *less-eq-nat-int* :: *nat-int* $\Rightarrow$ *nat-int* $\Rightarrow$ *bool* **is** *Set.subset-eq* $\langle proof \rangle$
lift-definition *less-nat-int* :: *nat-int* $\Rightarrow$ *nat-int* $\Rightarrow$ *bool* **is** *Set.subset* $\langle proof \rangle$
instance
$\langle proof \rangle$
end

instantiation *nat-int* :: *semilattice-inf*
begin
instance
$\langle proof \rangle$
end

instantiation *nat-int*:: *equal*
begin
definition *equal-nat-int* :: *nat-int* $\Rightarrow$ *nat-int* $\Rightarrow$ *bool*
where *equal-nat-int i j* $\equiv i \leq j \wedge j \leq i$
instance
$\langle proof \rangle$
end

context *nat-int*
begin
abbreviation *subseteq* :: *nat-int* $\Rightarrow$ *nat-int*$\Rightarrow$ *bool* (**infix** $\sqsubseteq$ *30*)
where $i \sqsubseteq j == i \leq j$
abbreviation *empty* :: *nat-int* ($\emptyset$)

where $\emptyset \equiv bot$

notation *inf* (**infix** $\sqcap$ *70*)

The union of two intervals is only defined, if it is also a discrete interval.

definition *union* :: *nat-int* $\Rightarrow$ *nat-int* $\Rightarrow$ *nat-int* (**infix** $\sqcup$ *65*)
where $i \sqcup j =$ *Abs-nat-int* (*Rep-nat-int* $i \cup$ *Rep-nat-int* j)

Non-empty intervals contain a minimal and maximal element. Two non-empty intervals i and j are consecutive, if the minimum of j is the successor of the maximum of i. Furthermore, the interval i can be chopped into the intervals j and k, if the union of j and k equals i, and if j and k are not-empty, they must be consecutive. Finally, we define the cardinality of discrete intervals by lifting the cardinality of sets.

definition *maximum* :: *nat-int* $\Rightarrow$ *nat*
where *maximum-def*: $i \neq \emptyset \Longrightarrow$ *maximum* $(i) =$ *Max* (*Rep-nat-int* i)

definition *minimum* :: *nat-int* $\Rightarrow$ *nat*
where *minimum-def*: $i \neq \emptyset \Longrightarrow$ *minimum*$(i) =$ *Min* (*Rep-nat-int* i)

definition *consec*:: *nat-int*$\Rightarrow$*nat-int* $\Rightarrow$ *bool*
where *consec* $i\ j \equiv (i \neq \emptyset \wedge j \neq \emptyset \wedge$ (*maximum*(i)+*1* $=$ *minimum* $j))$

definition *N-Chop* :: *nat-int* $\Rightarrow$ *nat-int* $\Rightarrow$ *nat-int* $\Rightarrow$ *bool* (*N'-Chop'*(-,-,-') *51*)
where *nchop-def* :
N-Chop$(i,j,k) \equiv (i =\ j \sqcup k\ \ \wedge (j = \emptyset \vee\ \ k = \emptyset \vee$ *consec* $j\ k))$

lift-definition *card'* ::*nat-int* $\Rightarrow$ *nat* (|-| *70*) **is** *card* $\langle$*proof*$\rangle$

For convenience, we also lift the membership relation and its negation to discrete intervals.

lift-definition *el*::*nat* $\Rightarrow$ *nat-int* $\Rightarrow$ *bool* (**infix** $\in$ *50*) **is** *Set.member* $\langle$*proof*$\rangle$

lift-definition *not-in* ::*nat* $\Rightarrow$ *nat-int* $\Rightarrow$ *bool* (**infix** $\notin$ *40*) **is** *Set.not-member* $\langle$*proof*$\rangle$
end

lemmas[*simp*] = *nat-int.el.rep-eq nat-int.not-in.rep-eq nat-int.card'.rep-eq*

context *nat-int*
begin

lemma *in-not-in-iff1* :$n \in i \longleftrightarrow \neg\ n \notin i$ $\langle$*proof*$\rangle$
lemma *in-not-in-iff2*: $n \notin i \longleftrightarrow \neg\ n \in i$ $\langle$*proof*$\rangle$

lemma *rep-non-empty-means-seq*:$i \neq \emptyset$
$\longrightarrow (\exists\, m\ n.\ m \leq n \wedge (\{m..n\} =($ *Rep-nat-int* $i)))$

⟨*proof*⟩

lemma *non-empty-max*: $i \neq \emptyset \longrightarrow (\exists\, m\ .\ maximum(i) = m)$
⟨*proof*⟩

lemma *non-empty-min*: $i \neq \emptyset \longrightarrow (\exists\, m\ .\ minimum(i) = m)$
⟨*proof*⟩

lemma *minimum-in*: $i \neq \emptyset \longrightarrow minimum\ i \in i$
⟨*proof*⟩

lemma *maximum-in*: $i \neq \emptyset \longrightarrow maximum\ i \in i$
⟨*proof*⟩

lemma *non-empty-elem-in*:$i \neq \emptyset \longleftrightarrow (\exists\, n.\ n \in i)$
⟨*proof*⟩

lemma *leq-nat-non-empty*:$(m::nat) \leq n \longrightarrow Abs\text{-}nat\text{-}int\{m..n\} \neq \emptyset$
⟨*proof*⟩

lemma *leq-max-sup*:$(m::nat) \leq n \longrightarrow Max\ \{m..n\} = n$
⟨*proof*⟩

lemma *leq-min-inf*: $(m::nat) \leq n \longrightarrow Min\ \{m..n\} = m$
⟨*proof*⟩

lemma *leq-max-sup'*:$(m::nat) \leq n \longrightarrow maximum(Abs\text{-}nat\text{-}int\{m..n\}) = n$
⟨*proof*⟩

lemma *leq-min-inf'*:$(m::nat) \leq n \longrightarrow minimum(Abs\text{-}nat\text{-}int\{m..n\}) = m$
⟨*proof*⟩

lemma *in-refl*:$(n::nat)\ \in Abs\text{-}nat\text{-}int\ \{n\}$
⟨*proof*⟩

lemma *in-singleton*: $m \in Abs\text{-}nat\text{-}int\{n\} \longrightarrow m = n$
⟨*proof*⟩

1.2 Algebraic properties of intersection and union.

lemma *inter-empty1*:$(i::nat\text{-}int) \sqcap \emptyset = \emptyset$
⟨*proof*⟩

lemma *inter-empty2*:$\emptyset \sqcap (i::nat\text{-}int) = \emptyset$
⟨*proof*⟩

lemma *un-empty-absorb1*:$i \sqcup \emptyset = i$
⟨*proof*⟩

lemma *un-empty-absorb2*:$\emptyset \sqcup i = i$
⟨*proof*⟩

Most properties of the union of two intervals depends on them being consectuive, to ensure that their union exists.

lemma *consec-un*:*consec i j* ∧ $n \notin$ *Rep-nat-int*(*i*) ∪ *Rep-nat-int j*
$\longrightarrow n \notin (i \sqcup j)$
⟨*proof*⟩

lemma *un-subset1*: *consec i j* $\longrightarrow i \sqsubseteq i \sqcup j$
⟨*proof*⟩

lemma *un-subset2*: *consec i j* $\longrightarrow j \sqsubseteq i \sqcup j$
⟨*proof*⟩

lemma *inter-distr1*:*consec j k* $\longrightarrow i \sqcap (j \sqcup k) = (i \sqcap j) \sqcup (i \sqcap k)$
⟨*proof*⟩

lemma *inter-distr2*:*consec j k* $\longrightarrow (j \sqcup k) \sqcap i = (j \sqcap i) \sqcup (k \sqcap i)$
⟨*proof*⟩

lemma *consec-un-not-elem1*:*consec i j* ∧ $n \notin i \sqcup j \longrightarrow n \notin i$
⟨*proof*⟩

lemma *consec-un-not-elem2*:*consec i j* ∧ $n \notin i \sqcup j \longrightarrow n \notin j$
⟨*proof*⟩

lemma *consec-un-element1*:*consec i j* ∧ $n \in i \longrightarrow n \in i \sqcup j$
⟨*proof*⟩

lemma *consec-un-element2*:*consec i j* ∧ $n \in j \longrightarrow n \in i \sqcup j$
⟨*proof*⟩

lemma *consec-lesser*: *consec i j* $\longrightarrow (\forall n\ m.\ (n \in i \wedge m \in j \longrightarrow n < m))$
⟨*proof*⟩

lemma *consec-in-exclusive1*:*consec i j* ∧ $n \in i \longrightarrow n \notin j$
⟨*proof*⟩

lemma *consec-in-exclusive2*:*consec i j* ∧ $n \in j \longrightarrow n \notin i$
⟨*proof*⟩

lemma *consec-un-max*:*consec i j* $\longrightarrow$ *maximum j* = *maximum* $(i \sqcup j)$
⟨*proof*⟩

lemma *consec-un-min*:*consec i j* $\longrightarrow$ *minimum i* = *minimum* $(i \sqcup j)$
⟨*proof*⟩

lemma *consec-un-defined*:

consec i j $\longrightarrow$ (*Rep-nat-int* ($i \sqcup j$) $\in$ {S . ($\exists$ (m::*nat*) n . {m..n }=S) })
⟨*proof*⟩

lemma *consec-un-min-max*:
consec i j $\longrightarrow$ *Rep-nat-int*($i \sqcup j$) = {*minimum i* .. *maximum j*}
⟨*proof*⟩

lemma *consec-un-equality*:
(*consec i j* $\wedge$ $k \neq \emptyset$)
$\longrightarrow$(*minimum* ($i \sqcup j$) = *minimum* (k) $\wedge$ *maximum* ($i \sqcup j$) = *maximum* (k))
$\longrightarrow$ $i \sqcup j = k$
⟨*proof*⟩

lemma *consec-trans-lesser*:
consec i j $\wedge$ *consec j k* $\longrightarrow$ ($\forall$ n m. ($n \in i \wedge m \in k \longrightarrow n < m$))
⟨*proof*⟩

lemma *consec-inter-empty*:*consec i j* $\Longrightarrow$ $i \sqcap j = \emptyset$
⟨*proof*⟩

lemma *consec-intermediate1*:*consec j k* $\wedge$ *consec i* ($j \sqcup k$) $\longrightarrow$ *consec i j*
⟨*proof*⟩

lemma *consec-intermediate2*:*consec i j* $\wedge$ *consec* ($i \sqcup j$) k $\longrightarrow$ *consec j k*
⟨*proof*⟩

lemma *un-assoc*: *consec i j* $\wedge$ *consec j k* $\longrightarrow$ ($i \sqcup j$) $\sqcup$ k = $i \sqcup$ ($j \sqcup k$)
⟨*proof*⟩

lemma *consec-assoc1*:*consec j k* $\wedge$ *consec i* ($j \sqcup k$) $\longrightarrow$ *consec* ($i \sqcup j$) k
⟨*proof*⟩

lemma *consec-assoc2*:*consec i j* $\wedge$ *consec* ($i \sqcup j$) k $\longrightarrow$ *consec i* ($j \sqcup k$)
⟨*proof*⟩

lemma *consec-assoc-mult*:
($i2$=$\emptyset$$\vee$ *consec i1 i2*) $\wedge$ ($i3$ =$\emptyset$ $\vee$ *consec i3 i4*) $\wedge$ (*consec* ($i1 \sqcup i2$) ($i3 \sqcup i4$))
$\longrightarrow$ ($i1 \sqcup i2$) $\sqcup$ ($i3 \sqcup i4$) = ($i1 \sqcup$ ($i2 \sqcup i3$)) $\sqcup$ $i4$
⟨*proof*⟩

lemma *card-subset-le*: $i \sqsubseteq i' \longrightarrow |i| \leq |i'|$
⟨*proof*⟩

lemma *card-subset-less*:(i::*nat-int*) < $i' \longrightarrow |i|<|i'|$
⟨*proof*⟩

lemma *card-empty-zero*:$|\emptyset|$ = *0*
⟨*proof*⟩

lemma *card-non-empty-geq-one*:$i \neq \emptyset \longleftrightarrow |i| \geq 1$
⟨*proof*⟩

lemma *card-min-max*:$i \neq \emptyset \longrightarrow |i| = (maximum\ i - minimum\ i) + 1$
⟨*proof*⟩

lemma *card-un-add*: $consec\ i\ j \longrightarrow |i \sqcup j| = |i| + |j|$
⟨*proof*⟩

lemma *singleton*:$|i| = 1 \longrightarrow (\exists\, n.\ Rep\text{-}nat\text{-}int\ i = \{n\})$
⟨*proof*⟩

lemma *singleton2*: $(\exists\, n.\ Rep\text{-}nat\text{-}int\ i = \{n\}) \longrightarrow |i| = 1$
⟨*proof*⟩

lemma *card-seq*:
$\forall\, i\ .|i| = x \longrightarrow (Rep\text{-}nat\text{-}int\ i = \{\} \vee (\exists\, n.\ Rep\text{-}nat\text{-}int\ i = \{n..n+(x-1)\}))$
⟨*proof*⟩

lemma *rep-single*: $Rep\text{-}nat\text{-}int\ (Abs\text{-}nat\text{-}int\ \{m..m\}) = \{m\}$
⟨*proof*⟩

lemma *chop-empty-right*: $\forall\, i.\ N\text{-}Chop(i,i,\emptyset)$
⟨*proof*⟩

lemma *chop-empty-left*: $\forall\, i.\ N\text{-}Chop(i,\ \emptyset,\ i)$
⟨*proof*⟩

lemma *chop-empty* : $N\text{-}Chop(\emptyset,\emptyset,\emptyset)$
⟨*proof*⟩

lemma *chop-always-possible*:$\forall\, i.\exists\ j\ k.\ N\text{-}Chop(i,j,k)$
⟨*proof*⟩

lemma *chop-add1*: $N\text{-}Chop(i,j,k) \longrightarrow |i| = |j| + |k|$
⟨*proof*⟩

lemma *chop-add2*:$|i| = x+y \longrightarrow (\exists\ j\ k.\ N\text{-}Chop(i,j,k) \wedge |j|=x \wedge |k|=y)$
⟨*proof*⟩

lemma *chop-single*:$(N\text{-}Chop(i,j,k) \wedge |i| = 1) \longrightarrow (\ |j| =0 \vee |k|=0)$
⟨*proof*⟩

lemma *chop-leq-max*:$N\text{-}Chop(i,j,k) \wedge consec\ j\ k \longrightarrow$
$(\forall\, n\ .\ n \in Rep\text{-}nat\text{-}int\ i \wedge n \leq maximum\ j \longrightarrow n \in Rep\text{-}nat\text{-}int\ j)$
⟨*proof*⟩

lemma *chop-geq-min*:*N-Chop*(*i*,*j*,*k*) ∧ *consec j k* ⟶
(∀ *n* . *n* ∈ *Rep-nat-int i* ∧ *minimum k* ≤ *n* ⟶ *n* ∈ *Rep-nat-int k*)
⟨*proof*⟩

lemma *chop-min*:*N-Chop*(*i*,*j*,*k*) ∧ *consec j k* ⟶ *minimum i* = *minimum j*
⟨*proof*⟩

lemma *chop-max*:*N-Chop*(*i*,*j*,*k*) ∧ *consec j k* ⟶ *maximum i* = *maximum k*
⟨*proof*⟩

lemma *chop-assoc1*:
N-Chop(*i*,*i1*,*i2*) ∧ *N-Chop*(*i2*,*i3*,*i4*)
⟶ (*N-Chop*(*i*, *i1* ⊔ *i3*, *i4*) ∧ *N-Chop*(*i1* ⊔ *i3*, *i1*, *i3*))
⟨*proof*⟩

lemma *chop-assoc2*:
N-Chop(*i*,*i1*,*i2*) ∧ *N-Chop*(*i1*,*i3*,*i4*)
⟶ *N-Chop*(*i*, *i3*, *i4* ⊔ *i2*) ∧ *N-Chop*(*i4* ⊔ *i2*, *i4*,*i2*)
⟨*proof*⟩

lemma *chop-subset1*:*N-Chop*(*i*,*j*,*k*) ⟶ *j* ⊑ *i*
⟨*proof*⟩

lemma *chop-subset2*:*N-Chop*(*i*,*j*,*k*) ⟶ *k* ⊑ *i*
⟨*proof*⟩

end
end

2 Closed Real-valued Intervals

We define a type for real-valued intervals. It consists of pairs of real numbers, where the first is lesser or equal to the second. Both endpoints are understood to be part of the interval, i.e., the intervals are closed. This also implies that we do not consider empty intervals.

We define a measure on these intervals as the difference between the left and right endpoint. In addition, we introduce a notion of shifting an interval by a real value x. Finally, an interval r can be chopped into s and t, if the left endpoint of r and s as well as the right endpoint of r and t coincides, and if the right endpoint of s is the left endpoint of t.

theory *RealInt*
imports *HOL.Real*
begin

typedef *real-int* = {*r*::(*real*∗*real*) . *fst r* ≤ *snd r*}
⟨*proof*⟩
setup-lifting *type-definition-real-int*

lift-definition *left*::*real-int* ⇒ *real* **is** *fst* ⟨*proof*⟩
lift-definition *right*::*real-int* ⇒ *real* **is** *snd* ⟨*proof*⟩

lemmas[*simp*] = *left.rep-eq right.rep-eq*

locale *real-int*
interpretation *real-int-class?*: *real-int* ⟨*proof*⟩

context *real-int*
begin

definition *length* :: *real-int* ⇒ *real* (‖-‖ *70*)
  where ‖*r*‖ ≡ *right r* − *left r*

definition *shift*::*real-int* ⇒ *real* ⇒ *real-int* (*shift* - -)
  where (*shift r x*) = *Abs-real-int*(*left r* +*x*, *right r* +*x*)

definition *R-Chop* :: *real-int* ⇒ *real-int* ⇒ *real-int* ⇒ *bool* (*R'-Chop'*(-,-,-') *51*)
  where *rchop-def* :
    R-Chop(*r*,*s*,*t*) == *left r* = *left s* ∧ *right s* = *left t* ∧ *right r* = *right t*

end

The intervals defined in this way allow for the definition of an order: the subinterval relation.

instantiation *real-int* :: *order*
begin
definition *less-eq-real-int r s* ≡ (*left r* ≥ *left s*) ∧ (*right r* ≤ *right s*)
definition *less-real-int r s* ≡ (*left r* ≥ *left s*) ∧ (*right r* ≤ *right s*)
                                ∧ ¬((*left s* ≥ *left r*) ∧ (*right s* ≤ *right r*))
instance
⟨*proof*⟩
end

context *real-int*
begin

lemma *left-leq-right*: *left r* ≤ *right r*
  ⟨*proof*⟩

lemma *length-ge-zero* : ‖*r*‖ ≥ *0*
  ⟨*proof*⟩

lemma *consec-add*:
  left r = *left s* ∧ *right r* = *right t* ∧ *right s* = *left t* ⟹ ‖*r*‖ = ‖*s*‖ + ‖*t*‖
  ⟨*proof*⟩

lemma *length-zero-iff-borders-eq*:$\|r\| = 0 \longleftrightarrow \mathit{left}\ r = \mathit{right}\ r$
⟨*proof*⟩

lemma *shift-left-eq-right*:$\mathit{left}\ (\mathit{shift}\ r\ x) \leq \mathit{right}\ (\mathit{shift}\ r\ x)$
⟨*proof*⟩

lemma *shift-keeps-length*:$\|r\| = \|\ \mathit{shift}\ r\ x\|$
⟨*proof*⟩

lemma *shift-zero*:$(\mathit{shift}\ r\ 0) = r$
⟨*proof*⟩

lemma *shift-additivity*:$(\mathit{shift}\ r\ (x+y)) = \mathit{shift}\ (\mathit{shift}\ r\ x)\ y$
⟨*proof*⟩

lemma *chop-always-possible*: $\forall r\ .\exists\ s\ t.\ \mathit{R\text{-}Chop}(r,s,t)$
⟨*proof*⟩

lemma *chop-singleton-right*: $\forall r.\exists\ s.\ \mathit{R\text{-}Chop}(r,r,s)$
⟨*proof*⟩

lemma *chop-singleton-left*: $\forall r.\exists\ s.\ \mathit{R\text{-}Chop}(r,s,r)$
⟨*proof*⟩

lemma *chop-add-length*:$\mathit{R\text{-}Chop}(r,s,t) \Longrightarrow \|r\| = \|s\| + \|t\|$
⟨*proof*⟩

lemma *chop-add-length-ge-0*:$\mathit{R\text{-}Chop}(r,s,t) \wedge \|s\| > 0 \wedge \|t\|>0 \longrightarrow \|r\|>0$
⟨*proof*⟩

lemma *chop-dense* : $\|r\| > 0 \longrightarrow (\exists\ s\ t.\ \mathit{R\text{-}Chop}(r,s,t) \wedge \|s\|>0 \wedge \|t\|>0)$
⟨*proof*⟩

lemma *chop-assoc1*:
$\mathit{R\text{-}Chop}(r,r1,r2) \wedge \mathit{R\text{-}Chop}(r2,r3,r4)$
$\longrightarrow \mathit{R\text{-}Chop}(r,\ \mathit{Abs\text{-}real\text{-}int}(\mathit{left}\ r1,\ \mathit{right}\ r3),\ r4)$
$\wedge\ \mathit{R\text{-}Chop}(\mathit{Abs\text{-}real\text{-}int}(\mathit{left}\ r1,\ \mathit{right}\ r3),\ r1,r3)$
⟨*proof*⟩

lemma *chop-assoc2*:
$\mathit{R\text{-}Chop}(r,r1,r2) \wedge \mathit{R\text{-}Chop}(r1,r3,r4)$
$\longrightarrow \mathit{R\text{-}Chop}(r,r3,\ \mathit{Abs\text{-}real\text{-}int}(\mathit{left}\ r4,\ \mathit{right}\ r2))$
$\wedge\ \mathit{R\text{-}Chop}(\mathit{Abs\text{-}real\text{-}int}(\mathit{left}\ r4,\ \mathit{right}\ r2),\ r4,r2)$
⟨*proof*⟩

lemma *chop-leq1*:$\mathit{R\text{-}Chop}(r,s,t) \longrightarrow s \leq r$
⟨*proof*⟩

lemma *chop-leq2*:$\mathit{R\text{-}Chop}(r,s,t) \longrightarrow t \leq r$

⟨*proof*⟩

lemma *chop-empty1*:*R-Chop*(*r*,*s*,*t*) ∧ ‖*s*‖ = *0* ⟶ *r* = *t*
⟨*proof*⟩

lemma *chop-empty2*:*R-Chop*(*r*,*s*,*t*) ∧ ‖*t*‖ = *0* ⟶ *r* = *s*
⟨*proof*⟩

end

end

3 Cars

We define a type to refer to cars. For simplicity, we assume that (countably) infinite cars exist.

theory *Cars*
imports *Main*
begin

The type of cars consists of the natural numbers. However, we do not define or prove any additional structure about it.

typedef *cars* = {*n*::*nat*. *True*} ⟨*proof*⟩

locale *cars*
begin

For the construction of possible counterexamples, it is beneficial to prove that at least two cars exist. Furthermore, we show that there indeed exist infinitely many cars.

lemma *at-least-two-cars-exists*:∃ *c d* ::*cars* . *c*≠*d*
⟨*proof*⟩

lemma *infinite-cars*:*infinite* {*c*::*cars* . *True*}
⟨*proof*⟩

end
end

4 Traffic Snapshots

Traffic snapshots define the spatial and dynamical arrangement of cars on the whole of the motorway at a single point in time. A traffic snapshot consists of several functions assigning spatial properties and dynamical behaviour to each car. The functions are named as follows.

- pos: positions of cars
- res: reservations of cars
- clm: claims of cars
- dyn: current dynamic behaviour of cars
- physical_size: the real sizes of cars
- braking_distance: braking distance each car needs in emergency

theory *Traffic*
imports *NatInt RealInt Cars*
begin

type-synonym *lanes = nat-int*
type-synonym *extension = real-int*

Definition of the type of traffic snapshots. The constraints on the different functions are the *sanity conditions* of traffic snapshots.

typedef *traffic =*
$\{ts :: (cars \Rightarrow real) * (cars \Rightarrow lanes) * (cars \Rightarrow lanes) * (cars \Rightarrow real \Rightarrow real) * (cars \Rightarrow real) * (cars \Rightarrow real).$
$(\forall c.\ ((fst\ (snd\ ts)))\ c \sqcap ((fst\ (snd\ (snd\ ts))))\ c = \emptyset\ ) \ \wedge$
$(\forall c.\ |(fst\ (snd\ ts))\ c| \geq 1) \wedge$
$(\forall c.\ |(fst\ (snd\ ts))\ c| \leq 2) \wedge$
$(\forall c.\ |(fst\ (snd\ (snd\ ts)))\ c)| \leq 1) \wedge$
$(\forall c.\ |(fst\ (snd\ ts))\ c| + |(fst\ (snd\ (snd\ ts)))\ c| \leq 2) \wedge$
$(\forall c.\ (fst(snd(snd\ (ts))))\ c \neq \emptyset \longrightarrow$
$(\exists\ n.\ Rep\text{-}nat\text{-}int(fst\ (snd\ ts)\ c) \cup Rep\text{-}nat\text{-}int(fst\ (snd\ (snd\ ts))\ c)$
$= \{n, n+1\})) \wedge$
$(\forall c\ .\ fst\ (snd\ (snd\ (snd\ (snd\ (ts)))))\ c > 0) \wedge$
$(\forall c.\ \ snd\ (snd\ (snd\ (snd\ (snd\ (ts)))))\ c > 0)$
$\}$
⟨*proof*⟩

locale *traffic*
begin

notation *nat-int.consec* (*consec*)

For brevity, we define names for the different functions within a traffic snapshot.

definition *pos*::*traffic* ⇒ (*cars* ⇒ *real*)
where *pos ts* ≡ *fst* (*Rep-traffic ts*)

definition *res*::*traffic* ⇒ (*cars* ⇒ *lanes*)
where *res ts* ≡ *fst* (*snd* (*Rep-traffic ts*))

definition *clm* ::*traffic* ⇒ (*cars* ⇒ *lanes*)
where *clm ts* ≡ *fst* (*snd* (*snd* (*Rep-traffic ts*)))

definition *dyn*::*traffic* ⇒ (*cars* ⇒ (*real*⇒ *real*))
where *dyn ts* ≡ *fst* (*snd* (*snd* (*snd* (*Rep-traffic ts*))))

definition *physical-size*::*traffic* ⇒ (*cars* ⇒ *real*)
where *physical-size ts* ≡ *fst* (*snd* (*snd* (*snd* (*snd* (*Rep-traffic ts*)))))

definition *braking-distance*::*traffic* ⇒ (*cars* ⇒ *real*)
where *braking-distance ts* ≡ *snd* (*snd* (*snd* (*snd* (*snd* (*Rep-traffic ts*)))))

It is helpful to be able to refer to the sanity conditions of a traffic snapshot via lemmas, hence we prove that the sanity conditions hold for each traffic snapshot.

lemma *disjoint*: (*res ts c*) ⊓ (*clm ts c*) = ∅
⟨*proof*⟩

lemma *atLeastOneRes*: *1* ≤ |*res ts c*|
⟨*proof*⟩

lemma *atMostTwoRes*: |*res ts c*| ≤ *2*
⟨*proof*⟩

lemma *atMostOneClm*: |*clm ts c*| ≤ *1*
⟨*proof*⟩

lemma *atMostTwoLanes*: |*res ts c*| +|*clm ts c*| ≤ *2*
⟨*proof*⟩

lemma *consecutiveRes*: |*res ts c*| =*2* ⟶ (∃ *n* . *Rep-nat-int* (*res ts c*) = {*n*,*n+1*})
⟨*proof*⟩

lemma *clmNextRes* :
(*clm ts c*) ≠ ∅ ⟶ (∃ *n*. *Rep-nat-int*(*res ts c*) ∪ *Rep-nat-int*(*clm ts c*) = {*n*, *n+1*})
⟨*proof*⟩

lemma *psGeZero*:∀ *c*. (*physical-size ts c* > *0*)
⟨*proof*⟩

lemma *sdGeZero*:∀ *c*. (*braking-distance ts c* > *0*)
⟨*proof*⟩

While not a sanity condition directly, the following lemma helps to establish general properties of HMLSL later on. It is a consequence of clmNextRes.

lemma *clm-consec-res*:
(*clm ts*) *c* ≠ ∅ ⟶ *consec* (*clm ts c*) (*res ts c*) ∨ *consec* (*res ts c*) (*clm ts c*)
⟨*proof*⟩

We define several possible transitions between traffic snapshots. Cars may create or withdraw claims and reservations, as long as the sanity conditions of the traffic snapshots are fullfilled.

In particular, a car can only create a claim, if it possesses only a reservation on a single lane, and does not already possess a claim. Withdrawing a claim can be done in any situation. It only has an effect, if the car possesses a claim. Similarly, the transition for a car to create a reservation is always possible, but only changes the spatial situation on the road, if the car already has a claim. Finally, a car may withdraw its reservation to a single lane, if its current reservation consists of two lanes.

All of these transitions concern the spatial properties of a single car at a time, i.e., for several cars to change their properties, several transitions have to be taken.

definition *create-claim* ::
  traffic⇒cars⇒nat⇒traffic⇒bool (- $-c'($ -, - $')\rightarrow$ - *27*)
where $(ts\ -c(c,n)\rightarrow\ ts') == (pos\ ts') = (pos\ ts)$
$\wedge\ (res\ ts') = (res\ ts)$
$\wedge\ (dyn\ ts') = (dyn\ ts)$
$\wedge\ (physical\text{-}size\ ts') = (physical\text{-}size\ ts)$
$\wedge\ (braking\text{-}distance\ ts') = (braking\text{-}distance\ ts)$
$\wedge\ |clm\ ts\ c| = 0$
$\wedge\ |res\ ts\ c| = 1$
$\wedge\ ((n{+}1) \in res\ ts\ c \vee (n{-}1 \in res\ ts\ c))$
$\wedge\ (clm\ ts') = (clm\ ts)(c{:=}Abs\text{-}nat\text{-}int\ \{n\})$

definition *withdraw-claim* ::
  traffic⇒cars ⇒traffic⇒bool (- $-wdc'($ - $')\rightarrow$ - *27*)
where $(ts\ -wdc(c)\rightarrow\ ts') == (pos\ ts') = (pos\ ts)$
$\wedge\ (res\ ts') = (res\ ts)$
$\wedge\ (dyn\ ts') = (dyn\ ts)$
$\wedge\ (physical\text{-}size\ ts') = (physical\text{-}size\ ts)$
$\wedge\ (braking\text{-}distance\ ts') = (braking\text{-}distance\ ts)$
$\wedge\ (clm\ ts') = (clm\ ts)(c{:=}\emptyset)$

definition *create-reservation* ::
  traffic⇒cars⇒traffic⇒bool (- $-r'($ - $')\rightarrow$ - *27*)
where $(ts\ -r(c)\rightarrow\ ts') == (pos\ ts') = (pos\ ts)$
$\wedge\ (res\ ts') = (res\ ts)(c{:=}(\ (res\ ts\ c) \sqcup (clm\ ts\ c)\ ))$
$\wedge\ (dyn\ ts') = (dyn\ ts)$
$\wedge\ (clm\ ts') = (clm\ ts)(c{:=}\emptyset)$
$\wedge\ (physical\text{-}size\ ts') = (physical\text{-}size\ ts)$
$\wedge\ (braking\text{-}distance\ ts') = (braking\text{-}distance\ ts)$

definition *withdraw-reservation* ::
  traffic⇒cars⇒nat⇒traffic⇒ bool (- $-wdr'($ -, - $')\rightarrow$ - *27*)
where $(ts\ -wdr(c,n)\rightarrow\ ts') == (pos\ ts') = (pos\ ts)$

$\wedge$ (*res ts'*) = (*res ts*)(*c*:= *Abs-nat-int*{*n*})
$\wedge$ (*dyn ts'*) = (*dyn ts*)
$\wedge$ (*clm ts'*) = (*clm ts*)
$\wedge$ (*physical-size ts'*) = (*physical-size ts*)
$\wedge$ (*braking-distance ts'*) = (*braking-distance ts*)
$\wedge$ *n* $\in$ (*res ts c*)
$\wedge$ |*res ts c*| = *2*

The following two transitions concern the dynamical behaviour of the cars. Similar to the spatial properties, a car may change its dynamics, by setting it to a new function f from real to real. Observe that this function is indeed arbitrary and does not constrain the possible behaviour in any way. However, this transition allows a car to change the function determining their braking distance (in fact, all cars are allowed to change this function, if a car changes sets a new dynamical function). That is, our model describes an over-approximation of a concrete situation, where the braking distance is determined by the dynamics.

The final transition describes the passing of x time units. That is, all cars update their position according to their current dynamical behaviour. Observe that this transition requires that the dynamics of each car is at least 0, for each time point between 0 and x. Hence, this condition denotes that all cars drive into the same direction. If the current dynamics of a car violated this constraint, it would have to reset its dynamics, until time may pass again.

definition *change-dyn*::
traffic⇒*cars*⇒(*real*⇒*real*)⇒*traffic*⇒ *bool* (- − *dyn'*(-,-') → - *27*)
where (*ts* −*dyn*(*c*, *f*)→ *ts'*) == (*pos ts'* = *pos ts*)
$\wedge$ (*res ts'* = *res ts*)
$\wedge$ (*clm ts'* = *clm ts*)
$\wedge$ (*dyn ts'* = (*dyn ts*)(*c*:= *f*))
$\wedge$ (*physical-size ts'*) = (*physical-size ts*)

definition *drive*::
traffic⇒*real*⇒*traffic*⇒*bool* (- − - → - *27*)
where (*ts* − *x* → *ts'*) == ($\forall$ *c*. (*pos ts' c* = (*pos ts c*) + (*dyn ts c x*)))
$\wedge$ ($\forall$ *c y*. *0* $\leq$ *y* $\wedge$ *y* $\leq$ *x* $\longrightarrow$ *dyn ts c y* $\geq$ *0*)
$\wedge$ (*res ts'* = *res ts*)
$\wedge$ (*clm ts'* = *clm ts*)
$\wedge$ (*dyn ts'* = *dyn ts*)
$\wedge$ (*physical-size ts'*) = (*physical-size ts*)
$\wedge$ (*braking-distance ts'*) = (*braking-distance ts*)

We bundle the dynamical transitions into *evolutions*, since we will only reason about combinations of the dynamical behaviour. This fits to the level of abstraction by hiding the dynamics completely inside of the model.

inductive *evolve*::*traffic* ⇒ *traffic* ⇒ *bool* (- ⇝ -)

where *refl* : $ts \rightsquigarrow ts$ |
change: $\exists\, c.\ \exists f.\ (ts\ -dyn(c,f)\rightarrow ts') \Longrightarrow ts' \rightsquigarrow ts'' \Longrightarrow ts \rightsquigarrow ts''$ |
drive: $\exists\, x.\ x \geq 0 \wedge\ (\ ts\ -x\rightarrow\ ts') \Longrightarrow ts' \rightsquigarrow ts'' \quad \Longrightarrow ts \rightsquigarrow ts''$

lemma *evolve-trans*:$(ts0 \rightsquigarrow ts1) \Longrightarrow (ts1 \rightsquigarrow ts2) \Longrightarrow (ts0 \rightsquigarrow ts2)$
⟨*proof*⟩

For general transition sequences, we introduce *abstract transitions*. A traffic snapshot ts' is reachable from ts via an abstract transition, if there is an arbitrary sequence of transitions from ts to ts'.

inductive *abstract*::*traffic* ⇒ *traffic* ⇒ *bool* (- ⇒ -) **for** *ts*
where *refl*: $(ts \Rightarrow ts)$ |
evolve: $ts \Rightarrow ts' \Longrightarrow ts' \rightsquigarrow ts'' \quad \Longrightarrow ts \Rightarrow ts''$ |
cr-clm: $ts \Rightarrow ts' \Longrightarrow \exists\, c.\ \exists\ n.\ (ts'\ -c(c,n)\rightarrow\ ts'') \quad \Longrightarrow ts \Rightarrow ts''$ |
wd-clm:$ts \Rightarrow ts' \Longrightarrow \exists\, c.\ (ts'\ -wdc(c)\rightarrow\ ts'') \Longrightarrow\ ts \Rightarrow ts''$ |
cr-res:$ts \Rightarrow ts' \Longrightarrow \exists\, c.\ (ts'\ -r(c)\rightarrow\ ts'') \Longrightarrow\ ts \Rightarrow ts''$ |
wd-res:$ts \Rightarrow ts' \Longrightarrow \exists\, c.\ \exists\ n.\ (ts'\ -wdr(c,n)\rightarrow\ ts'') \Longrightarrow\ ts \Rightarrow ts''$

lemma *abs-trans*: $(ts1 \Rightarrow ts2) \Longrightarrow (ts0 \Rightarrow ts1) \Longrightarrow (ts0 \Rightarrow ts2)$
⟨*proof*⟩

Most properties of the transitions are straightforward. However, to show that the transition to create a reservation is always possible, we need to explicitly construct the resulting traffic snapshot. Due to the size of such a snapshot, the proof is lengthy.

lemma *create-res-subseteq1*:$(ts\ -r(c)\rightarrow\ ts') \longrightarrow res\ ts\ c \sqsubseteq res\ ts'\ c$
⟨*proof*⟩

lemma *create-res-subseteq2*:$(ts\ -r(c)\rightarrow\ ts') \longrightarrow clm\ ts\ c \sqsubseteq res\ ts'\ c$
⟨*proof*⟩

lemma *create-res-subseteq1-neq*:$(ts\ -r(d)\rightarrow\ ts') \wedge d \neq c \longrightarrow res\ ts\ c = res\ ts'\ c$
⟨*proof*⟩

lemma *create-res-subseteq2-neq*:$(ts\ -r(d)\rightarrow\ ts') \wedge d \neq c \longrightarrow clm\ ts\ c = clm\ ts'\ c$
⟨*proof*⟩

lemma *always-create-res*:$\forall\, ts.\ \exists\, ts'.\ (ts\ -r(c)\rightarrow\ ts')$
⟨*proof*⟩

lemma *create-clm-eq-res*:$(ts\ -c(d,n)\rightarrow\ ts') \longrightarrow res\ ts\ c = res\ ts'\ c$
⟨*proof*⟩

lemma *withdraw-clm-eq-res*:$(ts\ -wdc(d)\rightarrow\ ts') \longrightarrow res\ ts\ c = res\ ts'\ c$
⟨*proof*⟩

lemma *withdraw-res-subseteq*:$(ts\ -wdr(d,n)\rightarrow\ ts') \longrightarrow res\ ts'\ c \sqsubseteq res\ ts\ c$
⟨*proof*⟩

end
end

5 Views on Traffic

In this section, we define a notion of locality for each car. These local parts of a road are called *views* and define the part of the model currently under consideration by a car. In particular, a view consists of

- the *extension*, a real-valued interval denoting the distance perceived,
- the *lanes*, a discrete interval, denoting which lanes are perceived,
- the *owner*, the car associated with this view.

theory *Views*
imports *NatInt RealInt Cars*
begin

type-synonym *lanes* = *nat-int*
type-synonym *extension* = *real-int*

record *view* =
ext::*extension*
lan ::*lanes*
own ::*cars*

The orders on discrete and continuous intervals induce an order on views. For two views v and v' with $v \leq v'$, we call v a *subview* of v'.

instantiation *view-ext*:: (*order*) *order*
begin
definition *less-eq-view-ext* (V:: *'a view-ext*) (V':: *'a view-ext*) ≡
$(ext\ V \leq ext\ V') \wedge (lan\ V \sqsubseteq\ lan\ V') \wedge own\ V = own\ V'$
$\wedge\ more\ V \leq more\ V'$
definition *less-view-ext* (V :: *'a view-ext*) (V':: *'a view-ext*) ≡
$(ext\ V \leq ext\ V') \wedge (lan\ V \sqsubseteq\ lan\ V') \wedge own\ V' = own\ V$
$\wedge\ more\ V \leq more\ V' \wedge$
$\neg((ext\ V' \leq ext\ V) \wedge (lan\ V' \sqsubseteq\ lan\ V) \wedge own\ V' = own\ V$
$\wedge\ more\ V' \leq more\ V)$
instance
⟨*proof*⟩
end

locale *view*
begin

notation *nat-int.maximum* (*maximum*)
notation *nat-int.minimum* (*minimum*)
notation *nat-int.consec* (*consec*)

We lift the chopping relations from discrete and continuous intervals to views, and introduce new notation for these relations.

definition *hchop* :: *view* ⇒ *view* ⇒ *view* ⇒ *bool* (-=-‖-)
where (*v*=*u*‖*w*) == *real-int.R-Chop*(*ext v*)(*ext u*)(*ext w*) ∧
lan v=*lan u* ∧
lan v=*lan w* ∧
own v = *own u* ∧
own v = *own w* ∧
more v = *more w* ∧
more v = *more u*
definition *vchop* :: *view* ⇒ *view* ⇒ *view* ⇒ *bool* (-=-−−-)
where (*v*=*u*−−*w*) == *nat-int.N-Chop*(*lan v*)(*lan u*)(*lan w*) ∧
ext v = *ext u* ∧
ext v = *ext w* ∧
own v = *own u* ∧
own v = *own w* ∧
more v = *more w* ∧
more v = *more u*

We can also switch the perspective of a view to the car *c*. That is, we substitute *c* for the original owner of the view.

definition *switch* :: *view* ⇒ *cars* ⇒ *view* ⇒ *bool* (- = - > -)
where (*v*=*c*>*w*) == *ext v* = *ext w* ∧
lan v = *lan w* ∧
own w = *c* ∧
more v = *more w*

Most of the lemmas in this theory are direct transfers of the corresponding lemmas on discrete and continuous intervals, which implies rather simple proofs. The only exception is the connection between subviews and the chopping operations. This proof is rather lengthy, since we need to distinguish several cases, and within each case, we need to explicitly construct six different views for the chopping relations.

lemma *h-chop-middle1*:(*v*=*u*‖*w*) ⟶ *left* (*ext v*) ≤ *right* (*ext u*)
⟨*proof*⟩

lemma *h-chop-middle2*:(*v*=*u*‖*w*) ⟶ *right* (*ext v*) ≥ *left* (*ext w*)
⟨*proof*⟩

lemma *horizontal-chop1*: $\exists\ u\ w.\ (v{=}u\|w)$
⟨*proof*⟩

lemma *horizontal-chop-empty-right* :$\forall\, v.\ \exists\ u.\ (v{=}v\|u)$
⟨*proof*⟩

lemma *horizontal-chop-empty-left* :$\forall\, v.\ \exists\, u.\ (v{=}u\|v)$
⟨*proof*⟩

lemma *horizontal-chop-non-empty*:
$\|ext\ v\| > 0 \longrightarrow (\exists\, u\ w.\ (v{=}u\|w) \wedge \|ext\ u\| > 0 \wedge \|ext\ w\|{>}0)$
⟨*proof*⟩

lemma *horizontal-chop-split-add*:
$x \geq 0 \wedge y \geq 0 \longrightarrow \|ext\ v\| = x{+}y \longrightarrow (\exists\, u\ w.\ (v{=}u\|w) \wedge \|ext\ u\| = x \wedge \|ext\ w\| = y)$
⟨*proof*⟩

lemma *horizontal-chop-assoc1*:
$(v{=}v1\|v2) \wedge (v2{=}v3\|v4) \longrightarrow (\exists\, v'.\ (v{=}v'\|v4) \wedge (v'{=}v1\|v3))$
⟨*proof*⟩

lemma *horizontal-chop-assoc2*:
$(v{=}v1\|v2) \wedge (v1{=}v3\|v4) \longrightarrow (\exists\, v'.\ (v{=}v3\|v') \wedge (v'{=}v4\|v2))$
⟨*proof*⟩

lemma *horizontal-chop-width-stable*:$(v{=}u\|w){\longrightarrow}|lan\ v|{=}|lan\ u|{\wedge}|lan\ v|{=}|lan\ w|$
⟨*proof*⟩

lemma *horizontal-chop-own-trans*:$(v{=}u\|w) \longrightarrow own\ u = own\ w$
⟨*proof*⟩

lemma *vertical-chop1*:$\forall\, v.\ \exists\ u\ w.\ (v{=}u{-}{-}w)$
⟨*proof*⟩

lemma *vertical-chop-empty-down*:$\forall\, v.\exists\ u.(v{=}v{-}{-}u)$
⟨*proof*⟩

lemma *vertical-chop-empty-up*:$\forall\, v.\exists\, u.(v{=}u{-}{-}v)$
⟨*proof*⟩

lemma *vertical-chop-assoc1*:
$(v{=}v1{-}{-}v2) \wedge (v2{=}v3{-}{-}v4) \longrightarrow (\exists\, v'.\ (v{=}v'{-}{-}v4) \wedge (v'{=}v1{-}{-}v3))$

⟨*proof*⟩

lemma *vertical-chop-assoc2*:
$(v{=}v1{--}v2) \wedge (v1{=}v3{--}v4) \longrightarrow (\exists\, v'.\ (v{=}v3{--}v') \wedge (v'{=}v4{--}v2))$
⟨*proof*⟩

lemma *vertical-chop-singleton*:
$(v{=}u{--}w) \wedge |lan\ v| = 1 \longrightarrow (\ |lan\ u| = 0 \vee |lan\ w| = 0)$
⟨*proof*⟩

lemma *vertical-chop-add1*:$(v{=}u{--}w) \longrightarrow |lan\ v| = |lan\ u| + |lan\ w|$
⟨*proof*⟩

lemma *vertical-chop-add2*:
$|lan\ v| = x{+}y \longrightarrow (\exists\ u\ w.\ \ (v{=}u{--}w) \wedge |lan\ u| = x \wedge |lan\ w| = y)$
⟨*proof*⟩

lemma *vertical-chop-length-stable*:
$(v{=}u{--}w) \longrightarrow \|ext\ v\| = \|ext\ u\| \wedge \|ext\ v\| = \|ext\ w\|$
⟨*proof*⟩

lemma *vertical-chop-own-trans*:$(v{=}u{--}w) \longrightarrow own\ u = own\ w$
⟨*proof*⟩

lemma *vertical-chop-width-mon*:
$(v{=}v1{--}v2) \wedge (v2{=}v3{--}v4) \wedge |lan\ v3| = x \longrightarrow |lan\ v| \geq x$
⟨*proof*⟩

lemma *horizontal-chop-leq1*:$(v{=}u\|w) \longrightarrow u \leq v$
⟨*proof*⟩

lemma *horizontal-chop-leq2*:$(v{=}u\|w) \longrightarrow w \leq v$
⟨*proof*⟩

lemma *vertical-chop-leq1*:$(v{=}u{--}w) \longrightarrow u \leq v$
⟨*proof*⟩

lemma *vertical-chop-leq2*:$(v{=}u{--}w) \longrightarrow w \leq v$
⟨*proof*⟩

lemma *somewhere-leq*:
$v \leq v' \longleftrightarrow (\exists\, v1\ v2\ v3\ vl\ vr\ vu\ vd.$
$(v'{=}vl\|v1) \wedge (v1{=}v2\|vr) \wedge (v2{=}vd{--}v3) \wedge (v3{=}v{--}vu))$
⟨*proof*⟩

The switch relation is compatible with the chopping relations, in the following sense. If we can chop a view v into two subviews u and w, and we can

reach v' via the switch relation, then there also exist two subviews u', w' of v', such that u' is reachable from u (and respectively for w', w).

lemma *switch-unique*:$(v =c> u) \wedge (v =c> w) \longrightarrow u = w$
⟨*proof*⟩

lemma *switch-exists*:$\exists\, c\ u.(\ v{=}c{>}u)$
⟨*proof*⟩

lemma *switch-always-exists*:$\forall\, c.\ \exists\, u.\ (v{=}c{>}u)$
⟨*proof*⟩

lemma *switch-origin*: $\exists\, u.\ (u{=}(own\ v){>}v)$
⟨*proof*⟩

lemma *switch-refl*:$(v{=}(own\ v){>}v)$
⟨*proof*⟩

lemma *switch-symm*:$(v{=}c{>}u) \longrightarrow (u{=}(own\ v){>}v)$
⟨*proof*⟩

lemma *switch-trans*:$(v{=}c{>}u) \wedge (u{=}d{>}w) \longrightarrow (v{=}d{>}w)$
⟨*proof*⟩

lemma *switch-triangle*:$(v{=}c{>}u) \wedge (v{=}d{>}w) \longrightarrow (u{=}d{>}w)$
⟨*proof*⟩

lemma *switch-hchop1*:
$(v{=}v1 \| v2) \wedge (v{=}c{>}v') \longrightarrow$
$(\exists\ v1'\ v2'.\ (v1 =c> v1') \wedge (v2 =c> v2') \wedge (v'{=}v1' \| v2'))$
⟨*proof*⟩

lemma *switch-hchop2*:
$(v'{=}v1' \| v2') \wedge (v{=}c{>}v') \longrightarrow$
$(\exists\ v1\ v2.\ (v1 =c> v1') \wedge (v2 =c> v2') \wedge (v{=}v1 \| v2))$
⟨*proof*⟩

lemma *switch-vchop1*:
$(v{=}v1 -\!- v2) \wedge (v{=}c{>}v') \longrightarrow$
$(\exists\ v1'\ v2'.\ (v1 =c> v1') \wedge (v2 =c> v2') \wedge (v'{=}v1' -\!- v2'))$
⟨*proof*⟩

lemma *switch-vchop2*:
$(v'{=}v1' -\!- v2') \wedge (v{=}c{>}v') \longrightarrow$
$(\exists\ v1\ v2.\ (v1 =c> v1') \wedge (v2 =c> v2') \wedge (v{=}v1 -\!- v2))$
⟨*proof*⟩

lemma *switch-leq*:$u' \leq u \wedge (v{=}c{>}u) \longrightarrow (\exists\, v'.\ (v'{=}c{>}u') \wedge v' \leq v)$
⟨*proof*⟩
end

end

6 Restrict Claims and Reservations to a View

To model that a view restricts the number of lanes a car may perceive, we define a function *restrict* taking a view v, a function f from cars to lanes and a car f, and returning the intersection between $f(c)$ and the set of lanes of v. This function will in the following only be applied to the functions yielding reservations and claims from traffic snapshots.

The lemmas of this section describe the connection between *restrict* and the different operations on traffic snapshots and views (e.g., the transition relations or the fact that reservations and claims are consecutive).

theory *Restriction*
  imports *Traffic Views*
begin

locale *restriction* = *view*+*traffic*
begin

definition *restrict* :: *view* ⇒ (*cars* ⇒ *lanes*) ⇒ *cars* ⇒ *lanes*
  where *restrict v f c* == (*f c*) ⊓ *lan v*

lemma *restrict-def'*: *restrict v f c* = *lan v* ⊓ *f c*
  ⟨*proof*⟩

lemma *restrict-subseteq*:*restrict v f c* ⊑ *f c*
  ⟨*proof*⟩

lemma *restrict-clm* : *restrict v* (*clm ts*) *c* ⊑ *clm ts c*
  ⟨*proof*⟩

lemma *restrict-res*: *restrict v* (*res ts*) *c* ⊑ *res ts c*
  ⟨*proof*⟩

lemma *restrict-view*:*restrict v f c* ⊑ *lan v*
  ⟨*proof*⟩

lemma *restriction-stable*:(*v*=*u*‖*w*) ⟶ *restrict u f c* = *restrict w f c*
  ⟨*proof*⟩

lemma *restriction-stable1*:(*v*=*u*‖*w*) ⟶ *restrict v f c* = *restrict u f c*
  ⟨*proof*⟩

lemma *restriction-stable2*:(*v*=*u*‖*w*) ⟶ *restrict v f c* = *restrict w f c*
  ⟨*proof*⟩

lemma *restriction-un*:
$(v{=}u{--}w) \longrightarrow restrict\ v\ f\ c = (restrict\ u\ f\ c \sqcup restrict\ w\ f\ c)$
⟨*proof*⟩

lemma *restriction-mon1*:$(v{=}u{--}w) \longrightarrow restrict\ u\ f\ c \sqsubseteq restrict\ v\ f\ c$
⟨*proof*⟩

lemma *restriction-mon2*:$(v{=}u{--}w) \longrightarrow restrict\ w\ f\ c \sqsubseteq restrict\ v\ f\ c$
⟨*proof*⟩

lemma *restriction-disj*:$(v{=}u{--}w) \longrightarrow (restrict\ u\ f\ c) \sqcap (restrict\ w\ f\ c) = \emptyset$
⟨*proof*⟩

lemma *vertical-chop-restriction-res-consec-or-empty*:
$(v{=}v1{--}v2) \wedge restrict\ v1\ (res\ ts)\ c \neq \emptyset \wedge consec\ ((lan\ v1))\ ((lan\ v2)) \wedge$
$\neg consec\ (restrict\ v1\ (res\ ts)\ c)\ (restrict\ v2\ (res\ ts)\ c)$
$\longrightarrow restrict\ v2\ (res\ ts)\ c = \emptyset$
⟨*proof*⟩

lemma *restriction-consec-res*:$(v{=}u{--}w)$
$\longrightarrow restrict\ u\ (res\ ts)\ c = \emptyset \vee restrict\ w\ (res\ ts)\ c = \emptyset$
$\vee\ \ consec\ (restrict\ u\ (res\ ts)\ c)\ (\ restrict\ w\ (res\ ts)\ c)$
⟨*proof*⟩

lemma *restriction-clm-res-disjoint*:
$(restrict\ v\ (res\ ts)\ c) \sqcap (restrict\ v\ (clm\ ts)\ c) = \emptyset$
⟨*proof*⟩

lemma *el-in-restriction-clm-singleton*:
$n \in restrict\ v\ (clm\ ts)\ c \longrightarrow (clm\ ts)\ c = Abs\text{-}nat\text{-}int(\{n\})$
⟨*proof*⟩

lemma *restriction-clm-v2-non-empty-v1-empty*:
$(v{=}u{--}w) \wedge restrict\ w\ (clm\ ts)\ c \neq \emptyset \wedge$
$consec\ ((lan\ u))\ ((lan\ w)) \longrightarrow restrict\ u\ (clm\ ts)\ c = \emptyset$
⟨*proof*⟩

lemma *restriction-consec-clm*:
$(v{=}u{--}w) \wedge consec\ (lan\ u)\ (lan\ w)$
$\longrightarrow restrict\ u\ (clm\ ts)\ c = \emptyset \vee restrict\ w\ (clm\ ts)\ c = \emptyset$
⟨*proof*⟩

lemma *restriction-add-res*:
$(v{=}u{--}w)$
$\longrightarrow |restrict\ v\ (res\ ts)\ c|{=}|restrict\ u\ (res\ ts)\ c|{+}|restrict\ w\ (res\ ts)\ c|$
⟨*proof*⟩

lemma *restriction-eq-view-card*:*restrict v f c = lan v* $\longrightarrow$ |*restrict v f c*| =|*lan v*|
⟨*proof*⟩

lemma *restriction-card-mon1*:(*v*=*u*−−*w*) $\longrightarrow$ |*restrict u f c*| $\leq$ |*restrict v f c*|
⟨*proof*⟩

lemma *restriction-card-mon2*:(*v*=*u*−−*w*) $\longrightarrow$ |*restrict w f c*| $\leq$ |*restrict v f c*|
⟨*proof*⟩

lemma *restriction-res-leq-two*:|*restrict v* (*res ts*) *c*| $\leq$ *2*
⟨*proof*⟩

lemma *restriction-clm-leq-one*:|*restrict v* (*clm ts*) *c*| $\leq$ *1*
⟨*proof*⟩

lemma *restriction-add-clm*:
(*v*=*u*−−*w*)
$\longrightarrow$ |*restrict v* (*clm ts*) *c*|=|*restrict u* (*clm ts*) *c*|+|*restrict w* (*clm ts*) *c*|
⟨*proof*⟩

lemma *restriction-card-mon-trans*:
(*v*=*v1*−−*v2*) $\wedge$ (*v2*=*v3*−−*v4*) $\wedge$ |*restrict v3 f c*| = *1* $\longrightarrow$ |*restrict v f c*|$\geq$*1*
⟨*proof*⟩

lemma *restriction-card-somewhere-mon*:
(*v*=*vl*‖*v1*)∧(*v1*=*v2*‖*vr*)∧(*v2*=*vu*−−*v3*)∧(*v3*=*v′*−−*vd*)∧|*restrict v′ f c*|=*1*
$\longrightarrow$ |*restrict v f c*| $\geq$ *1*
⟨*proof*⟩

lemma *restrict-eq-lan-subs*:
|*restrict v f c*| = |*lan v*| $\wedge$ (*restrict v f c* $\sqsubseteq$ *lan v*) $\longrightarrow$ *restrict v f c = lan v*
⟨*proof*⟩

lemma *create-reservation-restrict-union*:
(*ts*−*r*(*c*)→*ts′*)
$\longrightarrow$ *restrict v* (*res ts′*) *c = restrict v* (*res ts*) *c* $\sqcup$ *restrict v* (*clm ts*) *c*
⟨*proof*⟩

lemma *switch-restrict-stable*:(*v*=*c*>*u*) $\longrightarrow$ *restrict v f d = restrict u f d*
⟨*proof*⟩
end
end

7 Move a View according to Difference between Traffic Snapshots

In this section, we define a function to move a view according to the changes between two traffic snapshots. The intuition is that the view moves with the same speed as its owner. That is, if we move a view v from ts to ts', we shift the extension of the view by the difference in the position of the owner of v.

theory *Move*
imports *Traffic Views*
begin

context *traffic*
begin

definition *move*::*traffic* ⇒ *traffic* ⇒ *view* ⇒ *view*
where
move ts ts′ v = (| *ext* = *shift* (*ext v*) ((*pos ts′* (*own v*)) − *pos ts* (*own v*)),
lan = *lan v*,
own =*own v* |)

lemma *move-keeps-length*:‖*ext v*‖ = ‖*ext* (*move ts ts′ v*)‖
⟨*proof*⟩

lemma *move-keeps-lanes*:*lan v* = *lan* (*move ts ts′ v*) ⟨*proof*⟩

lemma *move-keeps-owner*:*own v* = *own* (*move ts ts′ v*) ⟨*proof*⟩

lemma *move-nothing* :*move ts ts v* = *v* ⟨*proof*⟩

lemma *move-trans*:
(*ts* ⇒ *ts′*) ∧ (*ts′* ⇒*ts″*) ⟶ *move ts′ ts″* (*move ts ts′ v*) = *move ts ts″ v*
⟨*proof*⟩

lemma *move-stability-res*:(*ts*−*r*(*c*)→*ts′*) ⟶ *move ts ts′ v* = *v*
and *move-stability-clm*: (*ts*−*c*(*c*,*n*)→*ts′*) ⟶ *move ts ts′ v* = *v*
and *move-stability-wdr*:(*ts*−*wdr*(*c*,*n*)→*ts′*) ⟶ *move ts ts′ v* = *v*
and *move-stability-wdc*:(*ts*−*wdc*(*c*)→*ts′*) ⟶ *move ts ts′ v* = *v*
⟨*proof*⟩

end
end

8 Sensors for Cars

This section presents the abstract definition of a function determining the sensor capabilities of cars. Such a function takes a car e, a traffic snapshot

ts and another car c, and returns the length of c as perceived by e at the situation determined by ts. The only restriction we impose is that this length is always greater than zero.

With such a function, we define a derived notion of the *space* the car c occupies as perceived by e. However, this does not define the lanes c occupies, but only a continuous interval. The lanes occupied by c are given by the reservation and claim functions of the traffic snapshot ts.

theory *Sensors*
  imports *Traffic Views*
begin

locale *sensors* = *traffic* + *view* +
  fixes *sensors*::(*cars*) ⇒ *traffic* ⇒ (*cars*) ⇒ *real*
  assumes *sensors-ge*:(*sensors e ts c*) > *0*
begin

definition *space* :: *traffic* ⇒ *view* ⇒ *cars* ⇒ *real-int*
  where *space ts v c* ≡ *Abs-real-int* (*pos ts c*, *pos ts c* + *sensors* (*own v*) *ts c*)

lemma *left-space*: *left* (*space ts v c*) = *pos ts c*
⟨*proof*⟩

lemma *right-space*: *right* (*space ts v c*) = *pos ts c* + *sensors* (*own v*) *ts c*
⟨*proof*⟩

lemma *space-nonempty*:*left* (*space ts v c*) < *right* (*space ts v c*)
  ⟨*proof*⟩

end
end

9 Visible Length of Cars with Perfect Sensors

Given a sensor function, we can define the length of a car c as perceived by the owner of a view v. This length is restricted by the size of the extension of the view v, but always given by a continuous interval, which may possibly be degenerate (i.e., a point-interval).

The lemmas connect the end-points of the perceived length with the end-points of the current view. Furthermore, they show how the chopping and subview relations affect the perceived length of a car.

theory *Length*
  imports *Sensors*
begin

context *sensors*
begin

definition *len*:: *view* ⇒ *traffic* ⇒ *cars* ⇒ *real-int*
where *len-def* :*len v* (*ts*) *c* ==
if (*left* (*space ts v c*) > *right* (*ext v*))
then *Abs-real-int* (*right* (*ext v*),*right* (*ext v*))
else
if (*right* (*space ts v c*) < *left* (*ext v*))
then *Abs-real-int* (*left* (*ext v*),*left* (*ext v*))
else
Abs-real-int (*max* (*left* (*ext v*)) (*left* (*space ts v c*)),
min (*right* (*ext v*)) (*right* (*space ts v c*)))

lemma *len-left*: *left* ((*len v ts*) *c*) ≥ *left* (*ext v*)
⟨*proof*⟩

lemma *len-right*: *right* ((*len v ts*) *c*) ≤ *right* (*ext v*)
⟨*proof*⟩

lemma *len-sub-int*:*len v ts c* ≤ *ext v*
⟨*proof*⟩

lemma *len-space-left*:
left (*space ts v c*) ≤ *right* (*ext v*) ⟶ *left* (*len v ts c*) ≥ *left* (*space ts v c*)
⟨*proof*⟩

lemma *len-space-right*:
right (*space ts v c*) ≥ *left* (*ext v*) ⟶ *right* (*len v ts c*) ≤ *right* (*space ts v c*)
⟨*proof*⟩

lemma *len-hchop-left-right-border*:
(*len v ts c* = *ext v*) ∧ (*v*=*v1*‖*v2*) ⟶ (*right* (*len v1 ts c*) = *right* (*ext v1*))
⟨*proof*⟩

lemma *len-hchop-left-left-border*:
((*len v ts*) *c* = *ext v*) ∧ (*v*=*v1*‖*v2*) ⟶ (*left* ((*len v1 ts*) *c*) = *left* (*ext v1*))
⟨*proof*⟩

lemma *len-view-hchop-left*:
((*len v ts*) *c* = *ext v*) ∧ (*v*=*v1*‖*v2*) ⟶ ((*len v1 ts*) *c* = *ext v1*)
⟨*proof*⟩

lemma *len-hchop-right-left-border*:
((*len v ts*) *c* = *ext v*) ∧ (*v*=*v1*‖*v2*) ⟶ (*left* ((*len v2 ts*) *c*) = *left* (*ext v2*))
⟨*proof*⟩

lemma *len-hchop-right-right-border*:
((*len v ts*) *c* = *ext v*) ∧ (*v*=*v1*‖*v2*) ⟶ (*right* ((*len v2 ts*) *c*) = *right* (*ext v2*))

⟨*proof*⟩

lemma *len-view-hchop-right*:
(($len\ v\ ts$) $c = ext\ v$) $\wedge$ ($v{=}v1 \| v2$) $\longrightarrow$ (($len\ v2\ ts$) $c = ext\ v2$)
⟨*proof*⟩

lemma *len-compose-hchop*:
($v{=}v1 \| v2$) $\wedge$ ($len\ v1\ (\ ts)\ c = ext\ v1$) $\wedge$ ($len\ v2\ (\ ts)\ c = ext\ v2$)
$\longrightarrow$ ($len\ v\ (\ ts)\ c = ext\ v$)
⟨*proof*⟩

lemma *len-stable*:($v{=}v1 -- v2$) $\longrightarrow$ $len\ v1\ ts\ c = len\ v2\ ts\ c$
⟨*proof*⟩

lemma *len-empty-on-subview1*:
$\|len\ v\ (\ ts)\ c\| = 0 \wedge (v{=}v1 \| v2) \longrightarrow \|len\ v1\ (\ ts)\ c\| = 0$
⟨*proof*⟩

lemma *len-empty-on-subview2*:
$\|len\ v\ ts\ c\| = 0 \wedge (v{=}v1 \| v2) \longrightarrow \|len\ v2\ ts\ c\| = 0$
⟨*proof*⟩

lemma *len-hchop-add*:
($v{=}v1 \| v2$) $\longrightarrow \|len\ v\ ts\ c\| = \|len\ v1\ ts\ c\| + \|len\ v2\ ts\ c\|$
⟨*proof*⟩

lemma *len-non-empty-inside*:
$\|len\ v\ (\ ts)\ c\| > 0$
$\longrightarrow left\ (space\ ts\ v\ c) < right\ (ext\ v) \wedge right\ (space\ ts\ v\ c) > left\ (ext\ v)$
⟨*proof*⟩

lemma *len-fills-subview*:
$\|len\ v\ ts\ c\| > 0$
$\longrightarrow (\exists\ v1\ v2\ v3\ v'.\ (v{=}v1 \| v2) \wedge (v2{=}v' \| v3) \wedge len\ v'\ ts\ c = ext\ v' \wedge$
$\|len\ v'\ ts\ c\| = \|len\ v\ ts\ c\|)$
⟨*proof*⟩

lemma *ext-eq-len-eq*:
$ext\ v = ext\ v' \wedge own\ v = own\ v' \longrightarrow len\ v\ ts\ c = len\ v'\ ts\ c$
⟨*proof*⟩

lemma *len-stable-down*:($v{=}v1 -- v2$) $\longrightarrow$ $len\ v\ ts\ c = len\ v1\ ts\ c$
⟨*proof*⟩

lemma *len-stable-up*:($v{=}v1 -- v2$) $\longrightarrow$ $len\ v\ ts\ c = len\ v2\ ts\ c$
⟨*proof*⟩

lemma *len-empty-subview*:$\|len\ v\ ts\ c\| = 0 \wedge (v' \leq v) \longrightarrow \|len\ v'\ ts\ c\| = 0$

⟨*proof*⟩

lemma *view-leq-len-leq*:$(ext\ v \leq ext\ v') \wedge (own\ v = own\ v') \wedge \|len\ v\ ts\ c\| > 0$
$\longrightarrow len\ v\ ts\ c \leq len\ v'\ ts\ c$
⟨*proof*⟩

end
end

10 Basic HMLSL

In this section, we define the basic formulas of HMLSL. All of these basic formulas and theorems are independent of the choice of sensor function. However, they show how the general operators (chop, changes in perspective, atomic formulas) work.

theory *HMLSL*
imports *Restriction Move Length*
begin

10.1 Syntax of Basic HMLSL

Formulas are functions associating a traffic snapshot and a view with a Boolean value.

type-synonym $\sigma = traffic \Rightarrow view \Rightarrow bool$

locale *hmlsl* = *restriction*+
fixes *sensors*::*cars* ⇒ *traffic* ⇒ *cars* ⇒ *real*
assumes *sensors-ge*:$(sensors\ e\ ts\ c) > 0$ **begin**
end

sublocale *hmlsl*<*sensors*
⟨*proof*⟩

context *hmlsl*
begin

All formulas are defined as abbreviations. As a consequence, proofs will directly refer to the semantics of HMLSL, i.e., traffic snapshots and views.

The first-order operators are direct translations into HOL operators.

abbreviation *mtrue* :: σ ($\top$)
where $\top \equiv \lambda\ ts\ w.\ True$
abbreviation *mfalse* :: σ ($\bot$)
where $\bot \equiv \lambda\ ts\ w.\ False$
abbreviation *mnot* :: $\sigma \Rightarrow \sigma$ ($\neg$-[52]53)
where $\neg\varphi \equiv \lambda\ ts\ w.\ \neg\varphi(ts)(w)$
abbreviation *mnegpred* :: $(cars \Rightarrow \sigma) \Rightarrow (cars \Rightarrow \sigma)$ ($^\neg$-[52]53)

where $^{\neg}\Phi \equiv \lambda x.\lambda\ ts\ w.\ \neg\Phi(x)(ts)(w)$
abbreviation *mand* :: $\sigma\Rightarrow\sigma\Rightarrow\sigma$ (**infixr**$\wedge$*51*)
where $\varphi\wedge\psi \equiv \lambda\ ts\ w.\ \varphi(ts)(w)\wedge\psi(ts)(w)$
abbreviation *mor* :: $\sigma\Rightarrow\sigma\Rightarrow\sigma$ (**infix** $\vee$*50*)
where $\varphi\vee\psi \equiv \lambda\ ts\ w.\ \varphi(ts)(w)\vee\psi(ts)(w)$
abbreviation *mimp* :: $\sigma\Rightarrow\sigma\Rightarrow\sigma$ (**infixr**$\rightarrow$*49*)
where $\varphi\rightarrow\psi \equiv \lambda\ ts\ w.\ \varphi(ts)(w)\longrightarrow\psi(ts)(w)$
abbreviation *mequ* :: $\sigma\Rightarrow\sigma\Rightarrow\sigma$ (**infixr**$\leftrightarrow$*48*)
where $\varphi\leftrightarrow\psi \equiv \lambda\ ts\ w.\ \varphi(ts)(w)\longleftrightarrow\psi(ts)(w)$
abbreviation *mforall* :: $('a\Rightarrow\sigma)\Rightarrow\sigma$ ($\forall$)
where $\forall\,\Phi \equiv \lambda\ ts\ w.\forall\,x.\ \Phi(x)(ts)(w)$
abbreviation *mforallB* :: $('a\Rightarrow\sigma)\Rightarrow\sigma$ (**binder**$\forall$ [*8*]*9*)
where $\forall\,x.\ \varphi(x) \equiv \forall\,\varphi$
abbreviation *mexists* :: $('a\Rightarrow\sigma)\Rightarrow\sigma$ ($\exists$)
where $\exists\,\Phi \equiv \lambda\ ts\ w.\exists\,x.\ \Phi(x)(ts)(w)$
abbreviation *mexistsB* :: $(('a)\Rightarrow\sigma)\Rightarrow\sigma$ (**binder**$\exists$ [*8*]*9*)
where $\exists\,x.\ \varphi(x) \equiv \exists\,\varphi$
abbreviation *meq* :: $'a\Rightarrow'a\Rightarrow\sigma$ (**infixr**$=$*60*) — Equality
where $x{=}y \equiv \lambda\ ts\ w.\ x = y$
abbreviation *mgeq* :: ($'a$::*ord*) $\Rightarrow$ $'a$ $\Rightarrow$ σ (**infix** $\geq$ *60*)
where $x \geq y \equiv \lambda\ ts\ w.\ x \geq y$
abbreviation *mge* ::($'a$::*ord*) $\Rightarrow$ $'a$ $\Rightarrow$ σ (**infix** $>$ *60*)
where $x > y \equiv \lambda\ ts\ w.\ x > y$

For the spatial modalities, we use the chopping operations defined on views. Observe that our chop modalities are existential.

abbreviation *hchop* :: $\sigma\Rightarrow\sigma\Rightarrow\sigma$ (**infixr** $\frown$ *53*)
where $\varphi \frown \psi \equiv \lambda\ ts\ w.\exists\,v\ u.\ (w{=}v\|u) \wedge \varphi(ts)(v)\wedge\psi(ts)(u)$
abbreviation *vchop* :: $\sigma\Rightarrow\sigma\Rightarrow\sigma$ (**infixr** $\smile$ *53*)
where $\varphi \smile \psi \equiv \lambda\ ts\ w.\exists\,v\ u.\ (w{=}v{-}{-}u) \wedge \varphi(ts)(v) \wedge \psi(ts)(u)$
abbreviation *somewhere* ::$\sigma\Rightarrow\sigma$ ($\langle$-$\rangle$ *55*)
where $\langle\varphi\rangle \equiv \top \frown (\top\smile\varphi \smile\top)\frown\top$
abbreviation *everywhere*::$\sigma\Rightarrow\sigma$ ([-] *55*)
where $[\varphi] \equiv \neg\langle\neg\varphi\rangle$

To change the perspective of a view, we use an operator in the fashion of Hybrid Logic.

abbreviation *at* :: *cars* $\Rightarrow$ σ $\Rightarrow$ σ (**@** - - *56*)
where **@***c* $\varphi \equiv \lambda ts\ w\ .\ \forall\,v'.\ (w{=}c{>}v') \longrightarrow \varphi(ts)(v')$

The behavioural modalities are defined as usual modal box-like modalities, where the accessibility relations are given by the different types of transitions between traffic snapshots.

abbreviation *res-box*::*cars* $\Rightarrow$ σ $\Rightarrow$ σ ($\Box r'$(-') - *55*)
where $\Box r(c)\ \varphi \equiv \lambda\ ts\ w.\ \forall\,ts'.\ (ts{-}r(c){\rightarrow}ts') \longrightarrow \varphi(ts')(w)$
abbreviation *clm-box*::*cars* $\Rightarrow$ σ $\Rightarrow$ σ ($\Box c'$(-') - *55*)
where $\Box c(c)\ \varphi \equiv \lambda\ ts\ w.\ \forall\,ts'\ n.\ (ts{-}c(c,n){\rightarrow}ts') \longrightarrow \varphi(ts')(w)$
abbreviation *wdres-box*::*cars* $\Rightarrow$ σ $\Rightarrow$ σ ($\Box wdr'$(-') - *55*)

where $\Box wdr(c)\ \varphi \equiv \lambda\ ts\ w.\ \forall\, ts'\ n.\ (ts{-}wdr(c{,}n){\rightarrow}ts') \longrightarrow \varphi(ts')(w)$
abbreviation *wdclm-box*::*cars* $\Rightarrow \sigma \Rightarrow \sigma$ $(\Box wdc'(\text{-}')$ - *55*$)$
where $\Box wdc(c)\ \varphi \equiv \lambda\ ts\ w.\ \forall\, ts'.\ (ts{-}wdc(c){\rightarrow}ts') \longrightarrow \varphi(ts')(w)$
abbreviation *time-box*::$\sigma \Rightarrow \sigma$ $(\Box\boldsymbol{\tau}$ - *55*$)$
where $\Box\boldsymbol{\tau}\ \varphi \equiv \lambda ts\ w.\ \forall\, ts'.\ (ts{\rightsquigarrow}ts') \longrightarrow \varphi(ts')(move\ ts\ ts'\ w)$
abbreviation *globally*::$\sigma \Rightarrow \sigma$ $(\mathbf{G}$ - *55*$)$
where $\mathbf{G}\ \varphi \equiv \lambda ts\ w.\ \forall\, ts'.\ (ts \Rightarrow ts') \longrightarrow \varphi(ts')(move\ ts\ ts'\ w)$

The spatial atoms to refer to reservations, claims and free space are direct translations of the original definitions of MLSL [2] into the Isabelle implementation.

abbreviation *re*:: *cars* $\Rightarrow \sigma$ $(re'(\text{-}')\ 70)$
where
$re(c) \equiv \lambda\ ts\ v.\ \|ext\ v\| > 0 \wedge len\ v\ ts\ c = ext\ v \wedge$
$restrict\ v\ (res\ ts)\ c = lan\ v \wedge |lan\ v| = 1$

abbreviation *cl*:: *cars* $\Rightarrow \sigma$ $(cl'(\text{-}')\ 70)$
where
$cl(c) \equiv \lambda\ ts\ v.\ \|ext\ v\| > 0 \wedge len\ v\ ts\ c = ext\ v \wedge$
$restrict\ v\ (clm\ ts)\ c = lan\ v \wedge |lan\ v| = 1$

abbreviation *free*:: σ $(free)$
where
$free \equiv \lambda\ ts\ v.\ \|ext\ v\| > 0 \wedge |lan\ v| = 1 \wedge$
$(\forall\, c.\ \ \|len\ v\ ts\ c\| = 0 \vee$
$(restrict\ v\ (clm\ ts)\ c = \emptyset \wedge restrict\ v\ (res\ ts)\ c = \emptyset))$

Even though we do not need them for the subsequent proofs of safety, we define ways to measure the number of lanes (width) and the size of the extension (length) of a view. This allows us to connect the atomic formulas for reservations and claims with the atom denoting free space [5].

abbreviation *width-eq*::*nat* $\Rightarrow \sigma$ $(\boldsymbol{\omega} =$ - *60*$)$
where $\boldsymbol{\omega} = n \equiv \lambda\ \ ts\ v.\ |lan\ v| = n$

abbreviation *width-geq*::*nat* $\Rightarrow \sigma$ $(\boldsymbol{\omega} \geq$ - *60*$)$
where $\boldsymbol{\omega} \geq n \equiv \lambda\ \ ts\ v.\ |lan\ v| \geq n$

abbreviation *width-ge*::*nat* $\Rightarrow \sigma$ $(\boldsymbol{\omega} >$ - *60*$)$
where $\boldsymbol{\omega} > n \equiv (\boldsymbol{\omega} = n{+}1) \smile \top$

abbreviation *length-eq*::*real* $\Rightarrow \sigma$ $(\mathbf{l} =$ - *60*$)$
where $\mathbf{l} = r \equiv \lambda\ ts\ v.\ \|ext\ v\| = r$

abbreviation *length-ge*:: *real* $\Rightarrow \sigma$ $(\mathbf{l} >$ - *60*$)$
where $\mathbf{l} > r \equiv \lambda\ ts\ v.\ \|ext\ v\| > r$

abbreviation *length-geq*::*real* $\Rightarrow \sigma$ $(\mathbf{l} \geq$ - *60*$)$
where $\mathbf{l} \geq r \equiv (\mathbf{l} = r) \vee (\mathbf{l} > r)$

For convenience, we use abbreviations for the validity and satisfiability of formulas. While the former gives a nice way to express theorems, the latter is useful within proofs.

abbreviation *valid* :: $\sigma \Rightarrow$ *bool* ($\models$ - *10*)
  where $\models \varphi \equiv \ \forall\, ts.\ \forall\, v.\ \varphi(ts)(v)$

abbreviation *satisfies*:: *traffic* $\Rightarrow$ *view* $\Rightarrow \sigma \Rightarrow$ *bool* (- , - $\models$ - *10*)
  where $ts,v \models \varphi \equiv \varphi(ts)(v)$

10.2 Theorems about Basic HMLSL

lemma *hchop-weaken1*: $\models \varphi \rightarrow (\varphi \frown \top)$
  ⟨*proof*⟩

lemma *hchop-weaken2*: $\models \varphi \rightarrow (\top \frown \varphi)$
  ⟨*proof*⟩

lemma *hchop-weaken*: $\models \varphi \rightarrow (\top \frown \varphi \frown \top)$
  ⟨*proof*⟩

lemma *hchop-neg1*:$\models \neg\ (\varphi \frown \top) \rightarrow ((\neg\ \varphi) \frown \top)$
  ⟨*proof*⟩

lemma *hchop-neg2*:$\models \neg\ (\top \frown \varphi\ ) \rightarrow (\top \frown \neg\ \varphi)$
  ⟨*proof*⟩

lemma *hchop-disj-distr1*:$\models ((\varphi \frown (\psi \vee \chi)) \leftrightarrow ((\varphi \frown \psi) \vee (\varphi \frown \chi)))$
  ⟨*proof*⟩

lemma *hchop-disj-distr2*:$\models (((\psi \vee \chi) \frown \varphi) \leftrightarrow ((\psi \frown \varphi) \vee (\chi \frown \varphi)))$
  ⟨*proof*⟩

lemma *hchop-assoc*:$\models \varphi \frown (\psi \frown \chi) \leftrightarrow (\varphi \frown \psi) \frown \chi$
  ⟨*proof*⟩

lemma *v-chop-weaken1*:$\models (\varphi \rightarrow (\varphi \smile \top))$
  ⟨*proof*⟩

lemma *v-chop-weaken2*:$\models (\varphi \rightarrow (\top \smile \varphi))$
  ⟨*proof*⟩

lemma *v-chop-assoc*:$\models (\varphi \smile (\psi \smile \chi)) \leftrightarrow ((\varphi \smile \psi) \smile \chi)$
  ⟨*proof*⟩

lemma *vchop-disj-distr1*:$\models ((\varphi \smile (\psi \vee \chi)) \leftrightarrow ((\varphi \smile \psi) \vee (\varphi \smile \chi)))$
  ⟨*proof*⟩

lemma *vchop-disj-distr2*:$\models (((\psi \vee \chi) \smile \varphi\ ) \leftrightarrow ((\psi \smile \varphi) \vee (\chi \smile \varphi)))$
  ⟨*proof*⟩

lemma *at-exists* :$\models \varphi \rightarrow (\exists\ c.\ @c\ \varphi)$
⟨*proof*⟩

lemma *at-conj-distr*:$\models(@c\ (\ \varphi \wedge \psi)) \leftrightarrow ((@c\ \varphi) \wedge (@c\ \psi))$
⟨*proof*⟩

lemma *at-disj-dist*:$\models(@c\ (\varphi \vee \psi)) \leftrightarrow ((@c\ \varphi\ ) \ \vee (\ @c\ \psi\ ))$
⟨*proof*⟩

lemma *at-hchop-dist1*:$\models(@c\ (\varphi \frown \psi)) \rightarrow ((@c\ \varphi) \frown (@c\ \psi))$
⟨*proof*⟩

lemma *at-hchop-dist2*:$\models(\ (@c\ \varphi) \frown (@c\ \psi)) \rightarrow (@c\ (\varphi \frown \psi))$
⟨*proof*⟩

lemma *at-hchop-dist*:$\models(\ (@c\ \varphi) \frown\ (@c\ \psi)) \leftrightarrow (@c\ (\varphi \frown \psi))$
⟨*proof*⟩

lemma *at-vchop-dist1*:$\models(@c\ (\varphi \smile \psi)) \rightarrow (\ (@c\ \varphi) \smile (@c\ \psi))$
⟨*proof*⟩

lemma *at-vchop-dist2*:$\models(\ (@c\ \varphi) \smile (@c\ \psi)) \rightarrow (@c\ (\varphi \smile \psi))$
⟨*proof*⟩

lemma *at-vchop-dist*:$\models(\ (@c\ \varphi) \smile (@c\ \psi)) \leftrightarrow (@c\ (\varphi \smile \psi))$
⟨*proof*⟩

lemma *at-eq*:$\models(@e\ c = d) \leftrightarrow (c = d)$
⟨*proof*⟩

lemma *at-neg1*:$\models(@c \neg \varphi) \rightarrow \neg\ (@c\ \varphi)$
⟨*proof*⟩

lemma *at-neg2*:$\models\neg\ (@c\ \varphi\ ) \rightarrow (\ (@c \neg \varphi))$
⟨*proof*⟩

lemma *at-neg* :$\models(@c(\ \neg \varphi)) \leftrightarrow \neg\ (@c\ \varphi)$
⟨*proof*⟩

lemma *at-neg'*:$ts,v \models \neg\ (@c\ \varphi) \leftrightarrow (@c(\ \neg \varphi))$ ⟨*proof*⟩

lemma *at-neg-neg1*:$\models(@c\ \varphi) \rightarrow \neg(@c \neg \varphi)$
⟨*proof*⟩

lemma *at-neg-neg2*:$\models\neg(@c \neg \varphi) \rightarrow (@c\ \ \varphi)$
⟨*proof*⟩

lemma *at-neg-neg*:$\models\ (@c\ \varphi) \leftrightarrow \neg(@c \neg \varphi)$

⟨*proof*⟩

lemma *globally-all-iff*:⊨ (**G**(∀ *c*. φ)) ↔ (∀ *c*.(**G** φ)) ⟨*proof*⟩
lemma *globally-all-iff'*:*ts*,*v*⊨ (**G**(∀ *c*. φ)) ↔ (∀ *c*.(**G** φ)) ⟨*proof*⟩

lemma *globally-refl*: ⊨(**G** φ) → φ
⟨*proof*⟩

lemma *globally-4*: ⊨ (**G** φ) → **G G** φ
⟨*proof*⟩

lemma *spatial-weaken*: ⊨ (φ → ⟨φ⟩)
⟨*proof*⟩

lemma *spatial-weaken2*:⊨ (φ → ψ) → (φ → ⟨ψ⟩)
⟨*proof*⟩

lemma *somewhere-distr*: ⊨ ⟨φ∨ψ⟩ ↔ ⟨φ⟩ ∨ ⟨ψ⟩
⟨*proof*⟩

lemma *somewhere-and*:⊨ ⟨φ ∧ ψ⟩ → ⟨φ⟩ ∧ ⟨ψ⟩
⟨*proof*⟩

lemma *somewhere-and-or-distr* :⊨(⟨ χ ∧ (φ ∨ ψ) ⟩ ↔ ⟨ χ ∧ φ ⟩ ∨ ⟨ χ ∧ ψ ⟩)
⟨*proof*⟩

lemma *width-add1*:⊨((***ω*** = *x*) ⌣ (***ω*** = *y*) → ***ω*** = *x*+*y*)
⟨*proof*⟩

lemma *width-add2*:⊨((***ω*** = *x*+*y*) → (***ω*** = *x*) ⌣ ***ω*** = *y*)
⟨*proof*⟩

lemma *width-hchop-stable*: ⊨((***ω*** = *x*) ↔ ((***ω*** = *x*) ⌢ (***ω***=*x*)))
⟨*proof*⟩

lemma *length-geq-zero*:⊨ (**l** ≥ *0*)
⟨*proof*⟩

lemma *length-split*: ⊨((**l** > *0*) → (**l** > *0*) ⌢ (**l** > *0*))
⟨*proof*⟩

lemma *length-meld*: ⊨((**l** > *0*) ⌢ (**l** > *0*) → (**l** > *0*))
⟨*proof*⟩

lemma *length-dense*:⊨((**l** > *0*) ↔ (**l** > *0*) ⌢ (**l** > *0*))
⟨*proof*⟩

lemma *length-add1*:⊨((**l**=*x*) ⌢ (**l**= *y*)) → (**l**= *x*+*y*)

⟨*proof*⟩

lemma *length-add2*:⊨ $(x \geq 0 \wedge y \geq 0) \rightarrow (\ (\mathbf{l}{=}x{+}y) \rightarrow ((\mathbf{l}{=}x) \frown (\mathbf{l}{=}y)))$
⟨*proof*⟩

lemma *length-add*:⊨ $(x \geq 0 \wedge y \geq 0) \rightarrow (\ (\mathbf{l}{=}x{+}y) \leftrightarrow ((\mathbf{l}{=}x) \frown (\mathbf{l}{=}y)))$
⟨*proof*⟩

lemma *length-vchop-stable*:⊨$(\mathbf{l} = x) \leftrightarrow ((\mathbf{l} = x) \smile (\ \mathbf{l} = x))$
⟨*proof*⟩

lemma *res-ge-zero*:⊨$(re(c) \rightarrow \mathbf{l}{>}0)$
⟨*proof*⟩

lemma *clm-ge-zero*:⊨$(cl(c) \rightarrow \mathbf{l}{>}0)$
⟨*proof*⟩

lemma *free-ge-zero*:⊨$free \rightarrow \mathbf{l}{>}0$
⟨*proof*⟩

lemma *width-res*:⊨$(re(c) \rightarrow \boldsymbol{\omega} = 1)$
⟨*proof*⟩

lemma *width-clm*:⊨$(cl(c) \rightarrow \boldsymbol{\omega} = 1)$
⟨*proof*⟩

lemma *width-free*:⊨$(free \rightarrow \boldsymbol{\omega} = 1)$
⟨*proof*⟩

lemma *width-somewhere-res*:⊨ $\langle re(c)\rangle \rightarrow (\boldsymbol{\omega} \geq 1)$
⟨*proof*⟩

lemma *clm-disj-res*:⊨ $\neg\ \langle\ cl(c) \wedge re(c)\ \rangle$
⟨*proof*⟩

lemma *width-ge*:⊨ $(\boldsymbol{\omega}{>}\ 0) \rightarrow (\exists\ x.\ (\boldsymbol{\omega} = x) \wedge (x > 0))$
⟨*proof*⟩

lemma *two-res-width*: ⊨$((re(c) \smile re(c)) \rightarrow \boldsymbol{\omega} = 2)$
⟨*proof*⟩

lemma *res-at-most-two*:⊨$\neg\ (re(c) \smile\ re(c)\ \smile\ re(c)\ )$
⟨*proof*⟩

lemma *res-at-most-two2*:⊨$\neg\ \langle re(c) \smile\ re(c)\ \smile\ re(c)\rangle$
⟨*proof*⟩

lemma *res-at-most-somewhere*:⊨$\neg\ \langle re(c)\rangle \smile \langle re(c)\rangle \smile \langle re(c)\rangle$

⟨*proof*⟩

lemma *res-adj*:$\models\neg\ \ (re(c) \smile (\boldsymbol{\omega} > 0) \smile re(c))$
⟨*proof*⟩

lemma *clm-sing*:$\models\neg\ \ (cl(c) \smile cl(c))$
⟨*proof*⟩

lemma *clm-sing-somewhere*:$\models\neg\ \ \langle cl(c) \smile cl(c)\rangle$
⟨*proof*⟩

lemma *clm-sing-not-interrupted*:$\models\ \neg(cl(c) \smile \top \smile cl(c))$
⟨*proof*⟩

lemma *clm-sing-somewhere2*:$\models\neg\ \ (\top \smile cl(c) \smile \top \smile\ \ cl(c) \smile \top)$
⟨*proof*⟩

lemma *clm-sing-somewhere3*:$\models\neg\ \ \langle(\top \smile cl(c) \smile \top \smile\ \ cl(c) \smile \top)\rangle$
⟨*proof*⟩

lemma *clm-at-most-somewhere*:$\models\neg\ (\langle cl(c)\rangle \smile \langle cl(c)\rangle)$
⟨*proof*⟩

lemma *res-decompose*: $\models(re(c)\ \ \rightarrow re(c) \frown re(c))$
⟨*proof*⟩

lemma *res-compose*: $\models(re(c) \frown re(c)\ \ \rightarrow re(c))$
⟨*proof*⟩

lemma *res-dense*:$\models re(c) \leftrightarrow re(c) \frown re(c)$
⟨*proof*⟩

lemma *res-continuous* :$\models(re(c)) \rightarrow (\neg\ (\top \frown (\ \neg re(c) \wedge \mathbf{1} > 0) \frown \top))$
⟨*proof*⟩

lemma *no-clm-before-res*:$\models\neg(cl(c) \frown re(c))$
⟨*proof*⟩

lemma *no-clm-before-res2*:$\models\neg\ (cl(c) \frown \top \frown re(c))$
⟨*proof*⟩

lemma *clm-decompose*: $\models(cl(c)\ \ \rightarrow cl(c) \frown cl(c))$
⟨*proof*⟩

lemma *clm-compose*: $\models(cl(c) \frown cl(c)\ \ \rightarrow cl(c))$

⟨*proof*⟩

lemma *clm-dense*:$\models cl(c) \leftrightarrow cl(c) \frown cl(c)$
⟨*proof*⟩

lemma *clm-continuous* :$\models(cl(c)) \rightarrow (\neg\ (\top \frown\ (\ \neg cl(c) \wedge \mathbf{l} > 0) \frown \top))$
⟨*proof*⟩

lemma *res-not-free*: $\models(\exists\ c.\ re(c) \rightarrow \neg free)$
⟨*proof*⟩

lemma *clm-not-free*: $\models(\exists\ c.\ cl(c) \rightarrow \neg free)$
⟨*proof*⟩

lemma *free-no-res*:$\models(free \rightarrow\ \neg(\exists\ c.\ re(c)))$
⟨*proof*⟩

lemma *free-no-clm*:$\models(free \rightarrow\ \neg(\exists\ c.\ cl(c)))$
⟨*proof*⟩

lemma *free-decompose*:$\models free \rightarrow (\ free \frown free)$
⟨*proof*⟩

lemma *free-compose*:$\models(free \frown free) \rightarrow free$
⟨*proof*⟩

lemma *free-dense*:$\models free \leftrightarrow (free \frown free)$
⟨*proof*⟩

lemma *free-dense2*:$\models free \rightarrow \top \frown free \frown \top$
⟨*proof*⟩

The next lemmas show the connection between the spatial. In particular, if the view consists of one lane and a non-zero extension, where neither a reservation nor a car resides, the view satisfies free (and vice versa).

lemma *no-cars-means-free*:
$\models((\mathbf{l}>0) \wedge (\boldsymbol{\omega} = 1) \wedge (\forall\, c.\ \neg\ (\top \frown\ (\ cl(c) \vee re(c)\ ) \frown \top))) \rightarrow free$
⟨*proof*⟩

lemma *free-means-no-cars*:
$\models free \rightarrow ((\mathbf{l}>0) \wedge (\boldsymbol{\omega} = 1) \wedge (\forall\, c.\ \neg\ (\top \frown\ (\ cl(c) \vee re(c)\ ) \frown \top)))$
⟨*proof*⟩

lemma *free-eq-no-cars*:
$\models free \leftrightarrow ((\mathbf{l}>0) \wedge (\boldsymbol{\omega} = 1) \wedge (\forall\, c.\ \neg\ (\top \frown\ (\ cl(c) \vee re(c)\ ) \frown \top)))$
⟨*proof*⟩

lemma *free-nowhere-res*:$\models$*free* $\rightarrow$ $\neg(\top \frown (re(c)) \frown \top)$
⟨*proof*⟩

lemma *two-res-not-res*: $\models((re(c) \smile re(c)) \rightarrow \neg re(c))$
⟨*proof*⟩

lemma *two-clm-width*: $\models((cl(c) \smile cl(c)) \rightarrow \boldsymbol{\omega} = 2)$
⟨*proof*⟩

lemma *two-res-no-car*: $\models(re(c) \smile re(c)) \rightarrow \neg(\exists\ c.\ (\ cl(c) \vee re(c))\ )$
⟨*proof*⟩

lemma *two-lanes-no-car*:$\models(\neg\ \boldsymbol{\omega} = 1) \rightarrow \neg(\exists\ c.(cl(c) \vee re(c)))$
⟨*proof*⟩

lemma *empty-no-car*:$\models(\ \mathbf{l} = 0) \rightarrow \neg(\exists\ c.(cl(c) \vee re(c)))$
⟨*proof*⟩

lemma *car-one-lane-non-empty*: $\models(\exists\ c.(cl(c) \vee re(c))) \rightarrow ((\boldsymbol{\omega} = 1) \wedge (\mathbf{l} > 0))$
⟨*proof*⟩

lemma *one-lane-notfree*:
$\models(\boldsymbol{\omega} = 1) \wedge (\mathbf{l} > 0) \wedge (\neg\ free) \rightarrow (\ (\top \frown (\exists\ c.\ (re(c) \vee cl(c))) \frown \top\ ))$
⟨*proof*⟩

lemma *one-lane-empty-or-car*:
$\models(\boldsymbol{\omega} = 1) \wedge (\mathbf{l} > 0) \rightarrow (free \vee (\top \frown (\exists\ c.\ (re(c) \vee cl(c))) \frown \top\ ))$
⟨*proof*⟩
end
end

11 Perfect Sensors

This section contains an instantiations of the sensor function for "perfect sensors". That is, each car can perceive both the physical size as well as the braking distance of each other car.

theory *Perfect-Sensors*
imports *../Length*
begin

definition *perfect*::*cars* ⇒ *traffic* ⇒ *cars* ⇒ *real*
where *perfect e ts c* ≡ *traffic.physical-size ts c* + *traffic.braking-distance ts c*

locale *perfect-sensors* = *traffic*+*view*
begin

interpretation *perfect-sensors* : *sensors perfect* :: *cars* ⇒ *traffic* ⇒ *cars* ⇒ *real*
⟨*proof*⟩

notation *perfect-sensors.space* (*space*)
notation *perfect-sensors.len* (*len*)

With this sensor definition, we can show that the perceived length of a car is independent of the spatial transitions between traffic snapshots. The length may only change during evolutions, in particular if the car changes its dynamical behaviour.

lemma *create-reservation-length-stable*:
$(ts{-}r(d){\rightarrow}ts') \longrightarrow len\ v\ ts\ c = len\ v\ ts'\ c$
⟨*proof*⟩

lemma *create-claim-length-stable*:
$(ts{-}c(d,n){\rightarrow}ts') \longrightarrow len\ v\ ts\ c = len\ v\ ts'\ c$
⟨*proof*⟩

lemma *withdraw-reservation-length-stable*:
$(ts{-}wdr(d,n){\rightarrow}ts') \longrightarrow len\ v\ ts\ c = len\ v\ ts'\ c$
⟨*proof*⟩

lemma *withdraw-claim-length-stable*:
$(ts{-}wdc(d){\rightarrow}ts') \longrightarrow len\ v\ ts\ c = len\ v\ ts'\ c$
⟨*proof*⟩

The following lemma shows that the perceived length is independent from the owner of the view. That is, as long as two views consist of the same extension, the perceived length of each car is the same in both views.

lemma *all-own-ext-eq-len-eq*:
$ext\ v = ext\ v' \longrightarrow len\ v\ ts\ c = len\ v'\ ts\ c$
⟨*proof*⟩

Finally, switching the perspective of a view does not change the perceived length.

lemma *switch-length-stable*:$(v{=}d{>}v') \longrightarrow len\ v\ ts\ c = len\ v'\ ts\ c$
⟨*proof*⟩

end
end

12 HMLSL for Perfect Sensors

Within this section, we instantiate HMLSL for cars with perfect sensors.

theory *HMLSL-Perfect*
imports *../HMLSL Perfect-Sensors*
begin

locale *hmlsl-perfect* = *perfect-sensors* + *restriction*
begin

interpretation *hmlsl* : *hmlsl perfect* :: *cars* ⇒ *traffic* ⇒ *cars* ⇒ *real*
⟨*proof*⟩

notation *hmlsl.re* (*re*′(-′))
notation *hmlsl.cl*(*cl*′(-′))
notation *hmlsl.len* (*len*)

The spatial atoms are independent of the perspective of the view. Hence we can prove several lemmas on the relation between the hybrid modality and the spatial atoms.

lemma *at-res1*:⊨(*re*(*c*)) → (∀ *d*. @*d* *re*(*c*))
⟨*proof*⟩

lemma *at-res2*:⊨(∀ *d*. @*d* *re*(*c*)) → *re*(*c*)
⟨*proof*⟩

lemma *at-res*:⊨*re*(*c*) ↔ (∀ *d*. @*d* *re*(*c*))
⟨*proof*⟩

lemma *at-res-inst*:⊨ (@*d* *re*(*c*)) →*re*(*c*)
⟨*proof*⟩

lemma *at-clm1*:⊨*cl*(*c*) → (∀ *d*. @*d* *cl*(*c*))
⟨*proof*⟩

lemma *at-clm2*:⊨(∀ *d*. @*d* *cl*(*c*)) → *cl*(*c*)
⟨*proof*⟩

lemma *at-clm*:⊨*cl*(*c*) ↔ (∀ *d*. @*d* *cl*(*c*))
⟨*proof*⟩

lemma *at-clm-inst*:⊨ (@*d* *cl*(*c*)) →*cl*(*c*)
⟨*proof*⟩

With the definition of sensors, we can also express how the spatial situation changes after the different transitions. In particular, we can prove lemmas corresponding to the activity and stability rules of the proof system for MLSL [5].
Observe that we were not able to prove these rules for basic HMLSL, since its generic sensor function allows for instantiations where the perceived length changes during spatial transitions.

lemma *backwards-res-act*:
(*ts* −*r*(*c*) → *ts*′) ∧ (*ts*′,*v* ⊨ *re*(*c*)) ⟶ (*ts*,*v* ⊨ *re*(*c*) ∨ *cl*(*c*))
⟨*proof*⟩

lemma *backwards-res-act-somewhere*:
$(ts\ -r(c)\rightarrow\ ts') \wedge (ts',v \models \langle re(c)\rangle) \longrightarrow (ts,v \models\langle\ re(c) \vee\ cl(c)\rangle\ )$
⟨*proof*⟩

lemma *backwards-res-stab*:
$(ts\ -r(d) \rightarrow\ ts') \wedge\ \ (d \neq c) \wedge (ts',v \models re(c)) \longrightarrow (ts,v \models re(c))$
⟨*proof*⟩

lemma *backwards-c-res-stab*:
$(ts\ -c(d,n) \rightarrow\ ts') \wedge (ts',v \models re(c)) \longrightarrow (ts,v \models re(c))$
⟨*proof*⟩

lemma *backwards-wdc-res-stab*:
$(ts\ -wdc(d) \rightarrow\ ts') \wedge (ts',v \models re(c)) \longrightarrow (ts,v \models re(c))$
⟨*proof*⟩

lemma *backwards-wdr-res-stab*:
$(ts\ -wdr(d,n) \rightarrow\ ts') \wedge (ts',v \models re(c)) \longrightarrow (ts,v \models re(c))$
⟨*proof*⟩

We now proceed to prove the *reservation lemma*, which was crucial in the manual safety proof [2].

lemma *reservation1*: $\models(re(c) \vee\ cl(c)) \rightarrow \Box r(c)\ re(c)$
⟨*proof*⟩

lemma *reservation2*: $\models(\Box r(c)\ re(c)) \rightarrow (re(c) \vee\ cl(c))$
⟨*proof*⟩

lemma *reservation*:$\models(\Box r(c)\ re(c)) \leftrightarrow (re(c) \vee\ cl(c))$
⟨*proof*⟩
end
end

13 Safety for Cars with Perfect Sensors

This section contains the definition of requirements for lane change and distance controllers for cars, with the assumption of perfect sensors. Using these definitions, we show that safety is an invariant along all possible behaviour of cars.

theory *Safety-Perfect*
imports *HMLSL-Perfect*
begin

context *hmlsl-perfect*
begin
interpretation *hmlsl* : *hmlsl perfect* :: *cars* ⇒ *traffic* ⇒ *cars* ⇒ *real*

⟨*proof*⟩

notation *hmlsl.re* (*re′*(-′))
notation *hmlsl.cl*(*cl′*(-′))
notation *hmlsl.len* (*len*)

Safety in the context of HMLSL means the absence of overlapping reservations. Using the somewhere modality, this is easy to formalise.

abbreviation *safe*::*cars*⇒*σ*
  where *safe e* ≡ ∀ *c*. ¬(*c* = *e*) → ¬ ⟨*re*(*c*) ∧ *re*(*e*) ⟩

The distance controller ensures, that as long as the cars do not try to change their lane, they keep their distance. More formally, if the reservations of two cars do not overlap, they will also not overlap after an arbitrary amount of time passed. Observe that the cars are allowed to change their dynamical behaviour, i.e., to accelerate and brake.

abbreviation *DC*::*σ*
  where *DC* ≡ **G**(∀ *c d*. ¬(*c* = *d*) →
    ¬⟨*re*(*c*) ∧ *re*(*d*)⟩ → □***τ*** ¬⟨*re*(*c*) ∧ *re*(*d*)⟩)

To identify possibly dangerous situations during a lane change manoeuvre, we use the *potential collision check*. It allows us to identify situations, where the claim of a car *d* overlaps with any part of the car *c*.

abbreviation *pcc*::*cars* ⇒ *cars* ⇒ *σ*
  where *pcc c d* ≡ ¬ (*c* = *d*) ∧ ⟨ *cl*(*d*) ∧ (*re*(*c*) ∨ *cl*(*c*)) ⟩

The only restriction the lane change controller imposes onto the cars is that in the case of a potential collision, they are not allowed to change the claim into a reservation.

abbreviation *LC*::*σ*
  where *LC* ≡ **G** (∀ *d*.(∃ *c*. *pcc c d*) → □*r*(*d*) ⊥)

The safety theorem is as follows. If the controllers of all cars adhere to the specifications given by *LC* and *DC*, and we start with an initially safe traffic snapshot, then all reachable traffic snapshots are also safe.

theorem *safety*:⊨(∀ *e*. *safe e*) ∧ *DC* ∧ *LC* → **G** (∀ *e*. *safe e*)
⟨*proof*⟩

While the safety theorem was only proven for a single car, we can show that the choice of this car is irrelevant. That is, if we have a safe situation, and switch the perspective to another car, the resulting situation is also safe.

lemma *safety-switch-invariant*:⊨(∀ *e*. *safe*(*e*)) → @*c* (∀ *e*. *safe*(*e*))
⟨*proof*⟩
end
end

14 Regular Sensors

This section contains an instantiations of the sensor function for "regular sensors". That is, each car can perceive its own physical size and braking distance. However, it can only perceive the physical size of other cars, and does not know about their braking distance.

theory *Regular-Sensors*
imports *../Length*
begin

definition *regular*::*cars* ⇒ *traffic* ⇒ *cars* ⇒ *real*
where *regular e ts c* ≡
if (*e* = *c*) *then traffic.physical-size ts c* + *traffic.braking-distance ts c*
else traffic.physical-size ts c

locale *regular-sensors* = *traffic* + *view*
begin

interpretation *regular-sensors*: *sensors regular* :: *cars* ⇒ *traffic* ⇒ *cars* ⇒ *real*
⟨*proof*⟩

notation *regular-sensors.space* (*space*)
notation *regular-sensors.len* (*len*)

Similar to the situation with perfect sensors, we can show that the perceived length of a car is independent of the spatial transitions between traffic snapshots. The length may only change during evolutions, in particular if the car changes its dynamical behaviour.

lemma *create-reservation-length-stable*:
(*ts*−*r*(*d*)→*ts*′) ⟶ *len v ts c* = *len v ts*′ *c*
⟨*proof*⟩

lemma *create-claim-length-stable*:
(*ts*−*c*(*d*,*n*)→*ts*′) ⟶ *len v ts c* = *len v ts*′ *c*
⟨*proof*⟩

lemma *withdraw-reservation-length-stable*:
(*ts*−*wdr*(*d*,*n*)→*ts*′) ⟶ *len v ts c* = *len v ts*′ *c*
⟨*proof*⟩

lemma *withdraw-claim-length-stable*:
(*ts*−*wdc*(*d*)→*ts*′) ⟶ *len v ts c* = *len v ts*′ *c*
⟨*proof*⟩

Since the perceived length of cars depends on the owner of the view, we can now prove how this perception changes if we change the perspective of a view.

lemma *sensors-le*:*e* ≠ *c* ⟶ *regular e ts c* < *regular c ts c*

⟨*proof*⟩

lemma *sensors-leq*: *regular e ts c* ≤ *regular c ts c*
⟨*proof*⟩

lemma *space-eq*: *own v* = *own v′* ⟶ *space ts v c* = *space ts v′ c*
⟨*proof*⟩

lemma *switch-space-le*:(*own v*) ≠ *c* ∧ (*v*=*c*>*v′*) ⟶ *space ts v c* < *space ts v′ c*
⟨*proof*⟩

lemma *switch-space-leq*:(*v*=*c*>*v′*) ⟶ *space ts v c* ≤ *space ts v′ c*
⟨*proof*⟩
end
end

15 HMLSL for Regular Sensors

Within this section, we instantiate HMLSL for cars with regular sensors.

theory *HMLSL-Regular*
imports *../HMLSL Regular-Sensors*
begin

locale *hmlsl-regular* = *regular-sensors* + *restriction*
begin
interpretation *hmlsl* : *hmlsl regular* :: *cars* ⇒ *traffic* ⇒ *cars* ⇒ *real*
⟨*proof*⟩

notation *hmlsl.re* (*re′*(-′))
notation *hmlsl.cl*(*cl′*(-′))
notation *hmlsl.len* (*len*)

The spatial atoms are dependent of the perspective of the view, hence we cannot prove similar lemmas as for perfect sensors.

However, we can still prove lemmas corresponding to the activity and stability rules of the proof system for MLSL [5].

Similar to the situation with perfect sensors, needed to instantiate the sensor function, to ensure that the perceived length does not change during spatial transitions.

lemma *backwards-res-act*:
(*ts* −*r*(*c*) → *ts′*) ∧ (*ts′*,*v* ⊨ *re*(*c*)) ⟶ (*ts*,*v* ⊨ *re*(*c*) ∨ *cl*(*c*))
⟨*proof*⟩

lemma *backwards-res-act-somewhere*:
(*ts* −*r*(*c*)→ *ts′*) ∧ (*ts′*,*v* ⊨ ⟨*re*(*c*)⟩) ⟶ (*ts*,*v* ⊨⟨ *re*(*c*) ∨ *cl*(*c*)⟩)
⟨*proof*⟩

lemma *backwards-res-stab*:
$(ts\ -r(d) \rightarrow ts') \wedge\ (d \neq c) \wedge (ts',v \models re(c)) \longrightarrow (ts,v \models re(c))$
⟨*proof*⟩

lemma *backwards-c-res-stab*:
$(ts\ -c(d,n) \rightarrow ts') \wedge (ts',v \models re(c)) \longrightarrow (ts,v \models re(c))$
⟨*proof*⟩

lemma *backwards-wdc-res-stab*:
$(ts\ -wdc(d) \rightarrow ts') \wedge (ts',v \models re(c)) \longrightarrow (ts,v \models re(c))$
⟨*proof*⟩

lemma *backwards-wdr-res-stab*:
$(ts\ -wdr(d,n) \rightarrow ts') \wedge (ts',v \models re(c)) \longrightarrow (ts,v \models re(c))$
⟨*proof*⟩

We now proceed to prove the *reservation lemma*, which was crucial in the manual safety proof [2].

lemma *reservation1*: $\models(re(c) \vee cl(c)) \rightarrow \Box r(c)\ re(c)$
⟨*proof*⟩

lemma *reservation2*: $\models(\Box r(c)\ re(c)) \rightarrow (re(c) \vee cl(c))$
⟨*proof*⟩

lemma *reservation*:$\models(\Box r(c)\ re(c)) \leftrightarrow (re(c) \vee cl(c))$
⟨*proof*⟩
end
end

16 Safety for Cars with Regular Sensors

This section contains the definition of requirements for lane change and distance controllers for cars, with the assumption of regular sensors. Using these definitions, we show that safety is an invariant along all possible behaviour of cars. However, we need to slightly amend our notion of safety, compared to the safety proof for perfect sensors.

theory *Safety-Regular*
imports *HMLSL-Regular*
begin
context *hmlsl-regular*
begin

interpretation *hmlsl* : *hmlsl regular* :: *cars* ⇒ *traffic* ⇒ *cars* ⇒ *real*
⟨*proof*⟩
notation *hmlsl.space* (*space*)
notation *hmlsl.re* (*re'*(-'))

notation *hmlsl.cl*(*cl′*(-′))
notation *hmlsl.len* (*len*)

First we show that the same "safety" theorem as for perfect senors can be proven. However, we will subsequently show that this theorem does not ensure safety from the perspective of each car.

The controller definitions for this "flawed" safety are the same as for perfect sensors.

abbreviation *safe*::*cars*⇒σ
  where *safe e* ≡ ∀ *c*. ¬(*c* = *e*) → ¬⟨*re*(*c*) ∧ *re*(*e*) ⟩

abbreviation *DC*::σ
  where *DC* ≡ **G**(∀ *c d*. ¬(*c* = *d*) →
    ¬⟨*re*(*c*) ∧ *re*(*d*)⟩ → □$_{\boldsymbol{\tau}}$ ¬⟨*re*(*c*) ∧ *re*(*d*)⟩)

abbreviation *pcc*::*cars* ⇒ *cars* ⇒ σ
  where *pcc c d* ≡ ¬ (*c* = *d*) ∧ ⟨ *cl*(*d*) ∧ (*re*(*c*) ∨ *cl*(*c*))⟩

abbreviation *LC*::σ
  where *LC* ≡ **G** (∀ *d*.(∃ *c*. *pcc c d*) → □*r*(*d*) ⊥)

The safety proof is exactly the same as for perfect sensors. Note in particular, that we fix a single car *e* for which we show safety.

theorem *safety-flawed*:⊨(∀ *e*. *safe e*) ∧ *DC* ∧ *LC* → **G** (∀ *e*. *safe e*)
⟨*proof*⟩

As stated above, the flawed safety theorem does not ensure safety for the perspective of each car. In particular, we can construct a traffic snapshot and a view, such that it satisfies our safety predicate for each car, but if we switch the perspective of the view to another car, the situation is unsafe. A visualisation of this situation can be found in the publication of this work at iFM 2017 [4].

lemma *safety-not-invariant-switch*:
  ∃ *ts v*. (*ts*,*v* ⊨ ∀ *e*. *safe*(*e*) ∧ (∃ *c*. @*c* ¬(∀ *e*. *safe*(*e*))))
⟨*proof*⟩

Now we show how to amend the controller specifications to gain safety as an invariant even with regular sensors.

The distance controller can be strengthened, by requiring that we switch to the perspective of one of the cars involved first, before checking for the collision. Since all variables are universally quantified, this ensures that no collision exists for the perspective of any car.

abbreviation *DC′*::σ
  where *DC′* ≡ **G** (∀ *c d*. ¬(*c* = *d*) →
    (@*d* ¬⟨*re*(*c*) ∧ *re*(*d*)⟩) → □$_{\boldsymbol{\tau}}$ @*d* ¬⟨*re*(*c*) ∧ *re*(*d*)⟩)

The amendment to the lane change controller is slightly different. Instead of checking the potential collision only from the perspective of the car d trying to change lanes, we require that also no other car may perceive a potential collision. Note that the restriction to d's behaviour can only be enforced within d, if the information from the other car is somehow passed to d. Hence, we require the cars to communicate in some way. However, we do not need to specifiy, *how* this communication is implemented.

abbreviation $LC'::\sigma$
  where $LC' \equiv \mathbf{G}\ (\ \forall\, d.\ (\exists\ \ c.\ \ (@c\ (pcc\ c\ d)) \vee (@d\ (pcc\ c\ d))) \rightarrow \Box r(d)\ \bot\ )$

With these new controllers, we can prove a stronger theorem than before. Instead of proving safety from the perspective of a single car as previously, we now only consider a traffic situation to be safe, if it satisfies the safety predicate from the perspective of *all* cars. Note that this immediately implies the safety invariance theorem proven for perfect sensors.

theorem *safety*:$\models (\forall\, e.\ @e\ (\ safe\ e\ )\ ) \wedge DC' \wedge LC' \rightarrow\ \mathbf{G}(\forall\ \ e.\ \ @\ e\ (safe\ e))$
$\langle proof \rangle$
end
end

References

[1] T. Braüner. *Hybrid logic and its proof-theory.* Springer, 2010.

[2] M. Hilscher, S. Linker, E. Olderog, and A. Ravn. An abstract model for proving safety of multi-lane traffic manoeuvres. In *ICFEM*, volume 6991 of *LNCS*, pages 404–419. Springer, 2011.

[3] S. Linker. *Proofs for Traffic Safety: Combining Diagrams and Logic.* PhD thesis, University of Oldenburg, 2015. http://oops.uni-oldenburg.de/2337/.

[4] S. Linker. *Spatial Reasoning About Motorway Traffic Safety with Isabelle/HOL*, pages 34–49. Springer International Publishing, 2017.

[5] S. Linker and M. Hilscher. Proof theory of a multi-lane spatial logic. *LMCS*, 11(3), 2015.