Formalization of Conflict Analysis of Programs with Procedures, Thread Creation, and Monitors in Isabelle/HOL

Peter Lammich
Markus Müller-Olm

Institut für Informatik, Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster

peter.lammich@uni-muenster.de and mmo@math.uni-muenster.de

February 23, 2021

Abstract

In this work we formally verify the soundness and precision of a static program analysis that detects conflicts (e.g. data races) in programs with procedures, thread creation and monitors with the Isabelle theorem prover. As common in static program analysis, our program model abstracts guarded branching by nondeterministic branching, but completely interprets the call-/return behavior of procedures, synchronization by monitors, and thread creation. The analysis is based on the observation that all conflicts already occur in a class of particularly restricted schedules. These restricted schedules are suited to constraint-system-based program analysis.

The formalization is based upon a flowgraph-based program model with an operational semantics as reference point.

Contents

1 Introduction

Conflicts are a common programming error in parallel programs. A conflict occurs if the same resource is accessed simultaneously by more than one process. Given a program π and two sets of control points U and V, the analysis problem is to decide whether there is an execution of π that simultaneously reaches one control point from U and one from V.

In this work, we use a flowgraph-based program model that extends a previously studied model [6] by reentrant monitors. In our model, programs can call recursive procedures, dynamically create new threads and synchronize via reentrant monitors. As usual in static program analysis, our program model abstracts away guarded branching by nondeterministic choice. We use an operational semantics as reference point for the correctness proofs. It models parallel execution by interleaving, i.e. just one thread is executed at any time and context switches may occur after every step. The next step is nondeterministically selected from all threads ready for execution. The analysis is based on a constraint system generated from the flowgraph. From its least solution, one can decide whether control points from U and V are simultaneously reachable or not.

It is notoriously hard to analyze concurrent programs with constraint systems because of the arbitrary fine-grained interleaving. The key idea behind our analysis is to use a restricted scheduling: While the interleaving semantics can switch the context after each step, the restricted scheduling just allows context switches at certain points of a thread's execution. We can show that each conflict is also reachable under this restricted scheduling. The restricted schedules can be easily analyzed with constraint systems as most of the complexity generated by arbitrary interleaving does no longer occur due to the restrictions. The remaining concurrency effects can be smoothly handled by using the concept of acquisition histories [5].

Related Work In [6] we present a constraint-system-based analysis for programs with thread creation and procedures but without monitors. The abstraction from synchronization is common in this line of research: There are automata-based techniques [1, 2, 3] as well as constraint-system-based techniques [7, 6] to analyze programs with procedures and either parallel calls or thread creation, but without any synchronization. In [5, 4] analysis techniques for interprocedural parallel programs with a fixed number of initial threads and nested locks are presented. These nested locks are not syntactically bound to the program structure, but assumed to be well-nested, that is any unlock statement is required to release the lock that was acquired last by the thread. Moreover, there is no support for reentrant locks[1]. We use monitors instead of locks. Monitors are syntactically bound to the program structure and thus well-nestedness is guaranteed statically. Additionally we directly support reentrant monitors. Our model cannot simulate well-nested locks where a lock statement and its corresponding unlock statement may be in different procedures (as in [5, 4]). As common programming languages like Java also use reentrant monitors rather than locks, we believe our model to be useful as well.

Document structure This document contains a commented formalization of these ideas as a collection of Isabelle/HOL theories. A more abstract description is in preparation. This document starts with formalization monitor consistent interleaving (Section 2) and acquisition histories (Section 3). Labeled transition systems are formalized in Section 4, and Section 5 defines the notion of interleaving semantics. Flowgraphs are defined in Section 6, and Section 7 describes their operational semantics. Section 8 contains the formalization of the restricted interleaving and Section 9 contains the constraint systems. Finally, the main result of this development – the correctness of the constraint systems w.r.t. to the operational semantics – is briefly stated in Section 10.

2 Monitor Consistent Interleaving

theory *ConsInterleave*
imports *Interleave Misc*
begin

The monitor consistent interleaving operator is defined on two lists of arbitrary elements, provided an abstraction function α that maps list elements to pairs of sets of monitors is available. $\alpha\ e = (M,\ M')$ intuitively means that step e enters the monitors in M and passes (enters and leaves) the monitors in M'. The consistent interleaving describes all interleavings of the two lists that are consistent w.r.t. the monitor usage.

2.1 Monitors of lists of monitor pairs

The following defines the set of all monitors that occur in a list of pairs of monitors. This definition is used in the following context: *mon-pl* (*map* α *w*) is the set of monitors used by a word *w* w.r.t. the abstraction α

definition
$\mathit{mon\text{-}pl}\ w == \mathit{foldl}\ (\cup)\ \{\}\ (\mathit{map}\ (\lambda e.\ \mathit{fst}\ e \cup \mathit{snd}\ e)\ w)$

lemma *mon-pl-empty*[*simp*]: $\mathit{mon\text{-}pl}\ []= \{\}$

[1] Reentrant locks can always be simulated by non-reentrant ones, at the cost of a worst-case exponential blowup of the program size

by (*unfold mon-pl-def*, *auto*)
lemma *mon-pl-cons*[*simp*]: *mon-pl* (*e*#*w*) = *fst e* ∪ *snd e* ∪ *mon-pl w*
by (*unfold mon-pl-def*) (*simp*, *subst foldl-un-empty-eq*, *auto*)

lemma *mon-pl-unconc*: !!*b*. *mon-pl* (*a*@*b*) = *mon-pl a* ∪ *mon-pl b*
by (*induct a*) *auto*

lemma *mon-pl-ileq*: $w \preceq w' \Longrightarrow$ *mon-pl w* ⊆ *mon-pl w′*
by (*induct rule*: *less-eq-list-induct*) *auto*

lemma *mon-pl-set*: *mon-pl w* = $\bigcup$ { *fst e* ∪ *snd e* | *e*. *e*∈*set w* }
by (*auto simp add*: *mon-pl-def foldl-set*) *blast*+

fun
cil :: ′*a list* ⇒ (′*a* ⇒ (′*m set* × ′*m set*)) ⇒ ′*a list* ⇒ ′*a list set*
(- ⊗- - [*64*,*64*,*64*] *64*) **where**
— Interleaving with the empty word results in the empty word
[] $\otimes_\alpha$ *w* = {*w*}
| *w* $\otimes_\alpha$ [] = {*w*}
— If both words are not empty, we can take the first step of one word, interleave the rest with the other word and then append the first step to all result set elements, provided it does not allocate a monitor that is used by the other word
| *e1*#*w1* $\otimes_\alpha$ *e2*#*w2* = (
if fst (α *e1*) ∩ *mon-pl* (*map* α (*e2*#*w2*)) = {} *then*
e1·(*w1* $\otimes_\alpha$ *e2*#*w2*)
else {}
) ∪ (
if fst (α *e2*) ∩ *mon-pl* (*map* α (*e1*#*w1*)) = {} *then*
e2·(*e1*#*w1* $\otimes_\alpha$ *w2*)
else {}
)

Note that this definition allows reentrant monitors, because it only checks that a monitor that is going to be entered by one word is not used in the *other* word. Thus the same word may enter the same monitor multiple times.

The next lemmas are some auxiliary lemmas to simplify the handling of the consistent interleaving operator.

lemma *cil-last-case-split*[*cases set*, *case-names left right*]:
⟦ *w*∈*e1*#*w1* $\otimes_\alpha$ *e2*#*w2*;
!!*w′*. ⟦*w*=*e1*#*w′*; *w′*∈(*w1* $\otimes_\alpha$ *e2*#*w2*);
fst (α *e1*) ∩ *mon-pl* (*map* α (*e2*#*w2*)) = {} ⟧ ⟹ *P*;
!!*w′*. ⟦*w*=*e2*#*w′*; *w′*∈(*e1*#*w1* $\otimes_\alpha$ *w2*);
fst (α *e2*) ∩ *mon-pl* (*map* α (*e1*#*w1*)) = {} ⟧ ⟹ *P*
⟧ ⟹ *P*
by (*auto elim*: *list-set-cons-cases split*: *if-split-asm*)

lemma *cil-cases*[*cases set*, *case-names both-empty left-empty right-empty app-left app-right*]:

$\llbracket\ w \in wa \otimes_{\alpha} wb;$
$\llbracket\ w=[];\ wa=[];\ wb=[]\ \rrbracket \Longrightarrow P;$
$\llbracket wa=[];\ w=wb \rrbracket \Longrightarrow P;$
$\llbracket\ w=wa;\ wb=[]\ \rrbracket \Longrightarrow P;$
$!!ea\ wa'\ w'.\ \llbracket w=ea\#w';\ wa=ea\#wa';\ w' \in wa' \otimes_{\alpha} wb;$
$fst\ (\alpha\ ea) \cap mon\text{-}pl\ (map\ \alpha\ wb) = \{\}\ \ \rrbracket \Longrightarrow P;$
$!!eb\ wb'\ w'.\ \llbracket w=eb\#w';\ wb=eb\#wb';\ w' \in wa \otimes_{\alpha} wb';$
$fst\ (\alpha\ eb) \cap mon\text{-}pl\ (map\ \alpha\ wa) = \{\}\ \ \rrbracket \Longrightarrow P$
$\rrbracket \Longrightarrow P$
proof (*induct wa* α *wb rule*: *cil.induct*)
case *1* **thus** *?case* **by** *simp* **next**
case *2* **thus** *?case* **by** *simp* **next**
case (*3 ea wa'* α *eb wb'*)
from *3.prems*(*1*) **show** *?thesis* **proof** (*cases rule*: *cil-last-case-split*)
case (*left w'*) **from** *3.prems*(*5*)[*OF left*(*1*) - *left*(*2*,*3*)] **show** *?thesis* **by** *simp* **next**
case (*right w'*) **from** *3.prems*(*6*)[*OF right*(*1*) - *right*(*2*,*3*)] **show** *?thesis* **by** *simp*
qed
qed

lemma *cil-induct'*[*case-names both-empty left-empty right-empty append*]: $\llbracket$
$\bigwedge\alpha.\ P\ \alpha\ []\ [];$
$\bigwedge\alpha\ ad\ ae.\ P\ \alpha\ []\ (ad\ \#\ ae);$
$\bigwedge\alpha\ z\ aa.\ P\ \alpha\ (z\ \#\ aa)\ [];$
$\bigwedge\alpha\ e1\ w1\ e2\ w2.\ \llbracket$
$\llbracket fst\ (\alpha\ e1) \cap mon\text{-}pl\ (map\ \alpha\ (e2\ \#\ w2)) = \{\}\rrbracket \Longrightarrow P\ \alpha\ w1\ (e2\ \#\ w2);$
$\llbracket fst\ (\alpha\ e2) \cap mon\text{-}pl\ (map\ \alpha\ (e1\ \#\ w1)) = \{\}\rrbracket \Longrightarrow P\ \alpha\ (e1\ \#\ w1)\ w2\rrbracket$
$\Longrightarrow P\ \alpha\ (e1\ \#\ w1)\ (e2\ \#\ w2)$
$\rrbracket \Longrightarrow P\ \alpha\ wa\ wb$
apply (*induct wa* α *wb rule*: *cil.induct*)
apply (*case-tac w*)
apply *auto*
done

lemma *cil-induct-fix*α: $\llbracket$
$P\ \alpha\ []\ [];$
$\bigwedge ad\ ae.\ P\ \alpha\ []\ (ad\ \#\ ae);$
$\bigwedge z\ aa.\ P\ \alpha\ (z\ \#\ aa)\ [];$
$\bigwedge e1\ w1\ e2\ w2.$
$\llbracket fst\ (\alpha\ e2) \cap mon\text{-}pl\ (map\ \alpha\ (e1\ \#\ w1)) = \{\} \longrightarrow P\ \alpha\ (e1\ \#\ w1)\ w2;$
$fst\ (\alpha\ e1) \cap mon\text{-}pl\ (map\ \alpha\ (e2\ \#\ w2)) = \{\} \longrightarrow P\ \alpha\ w1\ (e2\ \#\ w2)\rrbracket$
$\Longrightarrow P\ \alpha\ (e1\ \#\ w1)\ (e2\ \#\ w2)\rrbracket$
$\Longrightarrow P\ \alpha\ v\ w$
apply (*induct v* α *w rule*: *cil.induct*)
apply (*case-tac w*)
apply *auto*
done

lemma *cil-induct-fixα′*[*case-names both-empty left-empty right-empty append*]: $[\![$
$P\ \alpha\ []\ []$;
$\bigwedge ad\ ae.\ P\ \alpha\ []\ (ad\ \#\ ae)$;
$\bigwedge z\ aa.\ P\ \alpha\ (z\ \#\ aa)\ []$;
$\bigwedge e1\ w1\ e2\ w2.\ [\![$
$fst\ (\alpha\ e1) \cap mon\text{-}pl\ (map\ \alpha\ (e2\ \#\ w2)) = \{\} \Longrightarrow P\ \alpha\ w1\ (e2\ \#\ w2)$;
$fst\ (\alpha\ e2) \cap mon\text{-}pl\ (map\ \alpha\ (e1\ \#\ w1)) = \{\} \Longrightarrow P\ \alpha\ (e1\ \#\ w1)\ w2]\!]$
$\Longrightarrow P\ \alpha\ (e1\ \#\ w1)\ (e2\ \#\ w2)$
$]\!] \Longrightarrow P\ \alpha\ wa\ wb$
apply (*induct wa α wb rule*: *cil.induct*)
apply (*case-tac w*)
apply *auto*
done

lemma [*simp*]: $w \otimes_\alpha [] = \{w\}$
by (*cases w*, *auto*)

lemma *cil-contains-empty*[*rule-format*, *simp*]: $([] \in wa \otimes_\alpha wb) = (wa{=}[] \wedge wb{=}[])$
by (*induct wa α wb rule*: *cil.induct*) *auto*

lemma *cil-cons-cases*[*cases set*, *case-names left right*]: $[\![\ e\#w \in w1 \otimes_\alpha w2$;
$!!w1'.\ [\![w1{=}e\#w1';\ w{\in}w1' \otimes_\alpha w2;\ fst\ (\alpha\ e) \cap mon\text{-}pl\ (map\ \alpha\ w2) = \{\}\ ]\!] \Longrightarrow P$;
$!!w2'.\ [\![w2{=}e\#w2';\ w{\in}w1 \otimes_\alpha w2';\ fst\ (\alpha\ e) \cap mon\text{-}pl\ (map\ \alpha\ w1) = \{\}\ ]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
by (*cases rule*: *cil-cases*) *auto*

lemma *cil-set-induct*[*induct set*, *case-names empty left right*]: $!!\alpha\ w1\ w2.\ [\![$
$w{\in}w1 \otimes_\alpha w2$;
$!!\alpha.\ P\ []\ \alpha\ []\ []$;
$!!\alpha\ e\ w'\ w1'\ w2.\ [\![w'{\in}w1' \otimes_\alpha w2;\ fst\ (\alpha\ e) \cap mon\text{-}pl\ (map\ \alpha\ w2) = \{\}$;
$P\ w'\ \alpha\ w1'\ w2\ ]\!] \Longrightarrow P\ (e\#w')\ \alpha\ (e\#w1')\ w2$;
$!!\alpha\ e\ w'\ w2'\ w1.\ [\![w'{\in}w1 \otimes_\alpha w2';\ fst\ (\alpha\ e) \cap mon\text{-}pl\ (map\ \alpha\ w1) = \{\}$;
$P\ w'\ \alpha\ w1\ w2'\ ]\!] \Longrightarrow P\ (e\#w')\ \alpha\ w1\ (e\#w2')$
$]\!] \Longrightarrow P\ w\ \alpha\ w1\ w2$
by (*induct w*) (*auto intro*!: *cil-contains-empty elim*: *cil-cons-cases*)

lemma *cil-set-induct-fixα*[*induct set*, *case-names empty left right*]: $!!w1\ w2.\ [\![$
$w{\in}w1 \otimes_\alpha w2$;
$P\ []\ \alpha\ []\ []$;
$!!e\ w'\ w1'\ w2.\ [\![w'{\in}w1' \otimes_\alpha w2;\ fst\ (\alpha\ e) \cap mon\text{-}pl\ (map\ \alpha\ w2) = \{\}$;
$P\ w'\ \alpha\ w1'\ w2\ ]\!] \Longrightarrow P\ (e\#w')\ \alpha\ (e\#w1')\ w2$;
$!!e\ w'\ w2'\ w1.\ [\![w'{\in}w1 \otimes_\alpha w2';\ fst\ (\alpha\ e) \cap mon\text{-}pl\ (map\ \alpha\ w1) = \{\}$;
$P\ w'\ \alpha\ w1\ w2'\ ]\!] \Longrightarrow P\ (e\#w')\ \alpha\ w1\ (e\#w2')$
$]\!] \Longrightarrow P\ w\ \alpha\ w1\ w2$
by (*induct w*) (*auto intro*!: *cil-contains-empty elim*: *cil-cons-cases*)

lemma *cil-cons1*: $[\![w{\in}wa \otimes_\alpha wb;\ fst\ (\alpha\ e) \cap mon\text{-}pl\ (map\ \alpha\ wb) = \{\}]\!]$
$\Longrightarrow e\#w \in e\#wa \otimes_\alpha wb$
by (*cases wb*) *auto*

lemma *cil-cons2*: $[\![w{\in}wa{\otimes}_\alpha wb;\ fst\ (\alpha\ e) \cap mon\text{-}pl\ (map\ \alpha\ wa) = \{\}]\!]$
$\Longrightarrow e\#w \in wa \otimes_\alpha e\#wb$
by (*cases wa*) *auto*

2.2 Properties of consistent interleaving

— Consistent interleaving is a restriction of interleaving
lemma *cil-subset-il*: $w{\otimes}_\alpha w' \subseteq w{\otimes}w'$
apply (*induct w α w′ rule*: *cil.induct*)
apply *simp-all*
apply *safe*
apply *auto*
done

lemma *cil-subset-il′*: $w{\in}w1{\otimes}_\alpha w2 \Longrightarrow w{\in}w1{\otimes}w2$
using *cil-subset-il* **by** (*auto*)

— Consistent interleaving preserves the set of letters of both operands
lemma *cil-set*: $w{\in}w1{\otimes}_\alpha w2 \Longrightarrow set\ w = set\ w1 \cup set\ w2$
by (*induct rule*: *cil-set-induct-fixα*) *auto*
corollary *cil-mon-pl*: $w{\in}w1{\otimes}_\alpha w2$
$\Longrightarrow mon\text{-}pl\ (map\ \alpha\ w) = mon\text{-}pl\ (map\ \alpha\ w1) \cup mon\text{-}pl\ (map\ \alpha\ w2)$
by (*subst mon-pl-unconc*[*symmetric*]) (*simp add*: *mon-pl-set cil-set*, *blast 20*)

— Consistent interleaving preserves the length of both operands
lemma *cil-length*[*rule-format*]: $\forall\, w{\in}wa{\otimes}_\alpha wb.\ length\ w = length\ wa + length\ wb$
by (*induct rule*: *cil.induct*) *auto*

— Consistent interleaving contains all letters of each operand in the original order
lemma *cil-ileq*: $w{\in}w1{\otimes}_\alpha w2 \Longrightarrow w1{\preceq}w \wedge w2{\preceq}w$
by (*intro conjI cil-subset-il′ ileq-interleave*)

— Consistent interleaving is commutative and associative
lemma *cil-commute*: $w{\otimes}_\alpha w' = w'{\otimes}_\alpha w$
by (*induct rule*: *cil.induct*) *auto*

lemma *cil-assoc1*: $!!wl\ w1\ w2\ w3.\ [\![\ w{\in}wl{\otimes}_\alpha w3;\ wl{\in}w1{\otimes}_\alpha w2\ ]\!]$
$\Longrightarrow \exists\, wr.\ w{\in}w1{\otimes}_\alpha wr \wedge wr{\in}w2{\otimes}_\alpha w3$
proof (*induct w rule*: *length-compl-induct*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons e w*) **from** *Cons.prems*(*1*) **show** *?case* **proof** (*cases rule*: *cil-cons-cases*)
case (*left wl′*) **with** *Cons.prems*(*2*) **have** $e\#wl' \in w1{\otimes}_\alpha w2$ **by** *simp*
thus *?thesis* **proof** (*cases rule*: *cil-cons-cases*[*case-names left′ right′*])
case (*left′ w1′*)
from *Cons.hyps*[*OF - left*(*2*) *left′*(*2*)] **obtain** *wr* **where** *IHAPP*: $w \in w1' \otimes_\alpha wr\ wr \in w2 \otimes_\alpha w3$ **by** *blast*
have $e\#w{\in}e\#w1'{\otimes}_\alpha wr$ **proof** (*rule cil-cons1*[*OF IHAPP*(*1*)])
from *left left′ cil-mon-pl*[*OF IHAPP*(*2*)] **show** *fst* (*α e*) ∩ *mon-pl* (*map α*

wr) = {} **by** *auto*
qed
thus *?thesis* **using** *IHAPP*(*2*) *left'* **by** *blast*
next
case (*right' w2'*) **from** *Cons.hyps*[*OF* - *left*(*2*) *right'*(*2*)] **obtain** *wr* **where** *IHAPP*: $w \in w1 \otimes_\alpha wr$ $wr \in w2' \otimes_\alpha w3$ **by** *blast*
from *IHAPP*(*2*) *left* **have** $e\#wr \in e\#w2' \otimes_\alpha w3$ **by** (*auto intro*: *cil-cons1*)
moreover from *right' IHAPP*(*1*) **have** $e\#w \in w1 \otimes_\alpha e\#wr$ **by** (*auto intro*: *cil-cons2*)
ultimately show *?thesis* **using** *right'* **by** *blast*
qed
next
case (*right w3'*) **from** *Cons.hyps*[*OF* - *right*(*2*) *Cons.prems*(*2*)] **obtain** *wr* **where** *IHAPP*: $w \in w1 \otimes_\alpha wr$ $wr \in w2 \otimes_\alpha w3'$ **by** *blast*
from *IHAPP*(*2*) *right cil-mon-pl*[*OF Cons.prems*(*2*)] **have** $e\#wr \in w2 \otimes_\alpha e\#w3'$ **by** (*auto intro*: *cil-cons2*)
moreover from *IHAPP*(*1*) *right cil-mon-pl*[*OF Cons.prems*(*2*)] **have** $e\#w \in w1 \otimes_\alpha e\#wr$ **by** (*auto intro*: *cil-cons2*)
ultimately show *?thesis* **using** *right* **by** *blast*
qed
qed

lemma *cil-assoc2*:
assumes *A*: $w{\in}w1{\otimes_\alpha}wr$ **and** *B*: $wr{\in}w2{\otimes_\alpha}w3$
shows $\exists\, wl.\ w{\in}wl{\otimes_\alpha}w3 \wedge wl{\in}w1{\otimes_\alpha}w2$
proof –
from *A* **have** *A'*: $w{\in}wr{\otimes_\alpha}w1$ **by** (*simp add*: *cil-commute*)
from *B* **have** *B'*: $wr{\in}w3{\otimes_\alpha}w2$ **by** (*simp add*: *cil-commute*)
from *cil-assoc1*[*OF A' B'*] **obtain** *wl* **where** $w \in w3 \otimes_\alpha wl \wedge wl \in w2 \otimes_\alpha w1$ **by** *blast*
thus *?thesis* **by** (*auto simp add*: *cil-commute*)
qed

— Parts of the abstraction can be moved to the operands

lemma *cil-map*: $w{\in}w1 \otimes_{(\alpha\circ f)} w2 \Longrightarrow map\ f\ w \in map\ f\ w1 \otimes_\alpha map\ f\ w2$
proof (*induct rule*: *cil-set-induct-fixα*)
case *empty* **thus** *?case* **by** *auto*
next
case (*left e w' w1' w2*)
have $f\ e\ \#\ map\ f\ w' \in f\ e\ \#\ map\ f\ w1' \otimes_\alpha map\ f\ w2$ **proof** (*rule cil-cons1*)
from *left*(*2*) **have** $fst\ ((\alpha{\circ}f)\ e) \cap mon\text{-}pl\ (map\ \alpha\ (map\ f\ w2)) = \{\}$ **by** (*simp only*: *map-map*[*symmetric*])
thus $fst\ (\alpha\ (f\ e)) \cap mon\text{-}pl\ (map\ \alpha\ (map\ f\ w2)) = \{\}$ **by** (*simp only*: *o-apply*)
qed (*rule left*(*3*))
thus *?case* **by** *simp*
next
case (*right e w' w2' w1*)

have $f\ e\ \#\ map\ f\ w' \in map\ f\ w1 \otimes_\alpha f\ e\ \#\ map\ f\ w2'$ **proof** (*rule cil-cons2*)
from *right*(*2*) **have** $fst\ ((\alpha \circ f)\ e) \cap mon\text{-}pl\ (map\ \alpha\ (map\ f\ w1)) = \{\}$ **by** (*simp only*: *map-map*[*symmetric*])
thus $fst\ (\alpha\ (f\ e)) \cap mon\text{-}pl\ (map\ \alpha\ (map\ f\ w1)) = \{\}$ **by** (*simp only*: *o-apply*)
qed (*rule right*(*3*))
thus *?case* **by** *simp*
qed

end

3 Acquisition Histories

theory *AcquisitionHistory*
imports *ConsInterleave*
begin

The concept of *acquisition histories* was introduced by Kahlon, Ivancic, and Gupta [5] as a bounded size abstraction of executions that acquire and release locks that contains enough information to decide consistent interleavability. In this work, we use this concept for reentrant monitors. As in Section 2, we encode monitor usage information in pairs of sets of monitors, and regard lists of such pairs as (abstract) executions. An item (E, U) of such a list describes a sequence of steps of the concrete execution that first enters the monitors in E and then passes through the monitors in U. The monitors in E are never left by the execution. Note that due to the syntactic binding of monitors to the program structure, any execution of a single thread can be abstracted to a sequence of (E, U)-pairs. Restricting the possible schedules (see Section 8) will allow us to also abstract executions reaching a single program point to a sequence of such pairs.

We want to decide whether two executions are interleavable. The key observation of [5] is, that two executions e and e' are *not* interleavable if and only if there is a conflicting pair (m, m') of monitors, such that e enters (and never leaves) m and then uses m' and e' enters (and never leaves) m' and then uses m.

An acquisition history is a map from monitors to set of monitors. The acquisition history of an execution maps a monitor m that is allocated at the end of the execution to all monitors that are used after or in the same step that finally enters m. Monitors that are not allocated at the end of an execution are mapped to the empty set. Though originally used for a setting without reentrant monitors, acquisition histories also work for our setting with reentrant monitors.

This theory contains the definition of acquisition histories and acquisition history interleavability, an ordering on acquisition histories that reflects the blocking potential of acquisition histories, and a mapping function from paths to acquisition histories that is shown to be compatible with monitor consistent interleaving.

3.1 Definitions

Acquisition histories are modeled as functions from monitors to sets of monitors. Intuitively $m' \in h\ m$ models that an execution finally is in m, and monitor m' has been used (i.e. passed or entered) after or at the same time m has been finally entered. By convention, we have $m \in h\ m$ or $h\ m = \{\}$.

definition $ah == \{\ (h{::}'m \Rightarrow\ 'm\ set)\ .\ \forall\ \ m.\ h\ m = \{\} \lor m{\in}h\ m\ \}$

lemma *ah-cases*[*cases set*]: $[\![h{\in}ah;\ h\ m = \{\} \Longrightarrow P\ ;\ m \in h\ m \Longrightarrow P]\!] \Longrightarrow P$
  by (*unfold ah-def*) *blast*

3.2 Interleavability

Two acquisition histories *h1* and *h2* are considered interleavable, iff there is no conflicting pair of monitors *m1* and *m2*, where a pair of monitors *m1* and *m2* is called *conflicting* iff *m1* is used in *h2* after entering *m2* and, vice versa, *m2* is used in *h1* after entering *m1*.

definition
  $ah\text{-}il :: ('m \Rightarrow\ 'm\ set) \Rightarrow ('m \Rightarrow\ 'm\ set) \Rightarrow bool$ (**infix** [∗] *65*)
  where
  $h1\ [*]\ h2 == \neg(\exists\ m1\ m2.\ m1{\in}h2\ m2 \land m2 \in h1\ m1)$

From our convention, it follows (as expected) that the sets of entered monitors (lock-sets) of two interleavable acquisition histories are disjoint

lemma *ah-il-lockset-disjoint*:
  $[\![\ h1{\in}ah;\ h2{\in}ah;\ h1\ [*]\ h2\ ]\!] \Longrightarrow h1\ m = \{\} \lor h2\ m = \{\}$
  by (*unfold ah-il-def*) (*auto elim*: *ah-cases*)

Of course, acquisition history interleavability is commutative

lemma *ah-il-commute*: $h1\ [*]\ h2 \Longrightarrow h2\ [*]\ h1$
  by (*unfold ah-il-def*) *auto*

3.3 Used monitors

Let's define the monitors of an acquisition history, as all monitors that occur in the acquisition history

definition
  $mon\text{-}ah :: ('m \Rightarrow\ 'm\ set) \Rightarrow\ 'm\ set$
  where
  $mon\text{-}ah\ h == \bigcup \{\ h(m)\ |\ m.\ True\}$

3.4 Ordering

The element-wise subset-ordering on acquisition histories intuitively reflects the blocking potential: The bigger the acquisition history, the fewer acquisition histories are interleavable with it.

Note that the Isabelle standard library automatically lifts the subset ordering to functions, so we need no explicit definition here.

— The ordering is compatible with interleavability, i.e. smaller acquisition histories are more likely to be interleavable.
lemma *ah-leq-il*: $[\![\ h1\ [*]\ h2;\ h1' \leq h1;\ h2' \leq h2\ ]\!] \Longrightarrow h1'\ [*]\ h2'$
  by (*unfold ah-il-def le-fun-def* [**where** *'b='a set*]) *blast+*
lemma *ah-leq-il-left*: $[\![\ h1\ [*]\ h2;\ h1' \leq h1\ ]\!] \Longrightarrow h1'\ [*]\ h2$ **and**
    ah-leq-il-right: $[\![\ h1\ [*]\ h2;\ h2' \leq h2\ ]\!] \Longrightarrow h1\ [*]\ h2'$
  by (*unfold ah-il-def le-fun-def* [**where** *'b='a set*]) *blast+*

3.5 Acquisition histories of executions

Next we define a function that abstracts from executions (lists of enter/use pairs) to acquisition histories

primrec αah :: (*'m set* × *'m set*) *list* ⇒ *'m* ⇒ *'m set* **where**
  $\alpha ah\ []\ m = \{\}$
| $\alpha ah\ (e\#w)\ m = (if\ m{\in}fst\ e\ then\ fst\ e \cup snd\ e \cup mon\text{-}pl\ w\ else\ \alpha ah\ w\ m)$

— αah generates valid acquisition histories
lemma *αah-ah*: $\alpha ah\ w \in ah$
  apply (*induct w*)
  apply (*unfold ah-def*)
  apply *simp*
  apply (*fastforce split*: *if-split-asm*)
  done

lemma *αah-hd*: $[\![m{\in}fst\ e;\ x{\in}fst\ e \cup snd\ e \cup mon\text{-}pl\ w]\!] \Longrightarrow x{\in}\alpha ah\ (e\#w)\ m$
  by *auto*
lemma *αah-tl*: $[\![m{\notin}fst\ e;\ x{\in}\alpha ah\ w\ m]\!] \Longrightarrow x{\in}\alpha ah\ (e\#w)\ m$
  by *auto*

lemma *αah-cases*[*cases set*, *case-names hd tl*]: $[\![$
    $x{\in}\alpha ah\ w\ m;$
    $!!e\ w'.\ [\![w{=}e\#w';\ m{\in}fst\ e;\ x{\in}fst\ e \cup snd\ e \cup mon\text{-}pl\ w']\!] \Longrightarrow P;$
    $!!e\ w'.\ [\![w{=}e\#w';\ m{\notin}fst\ e;\ x{\in}\alpha ah\ w'\ m]\!] \Longrightarrow P$
  $]\!] \Longrightarrow P$
  by (*cases w*) (*simp-all split*: *if-split-asm*)

lemma *αah-cons-cases*[*cases set*, *case-names hd tl*]: $[\![$
    $x{\in}\alpha ah\ (e\#w')\ m;$
    $[\![m{\in}fst\ e;\ x{\in}fst\ e \cup snd\ e \cup mon\text{-}pl\ w']\!] \Longrightarrow P;$
    $[\![m{\notin}fst\ e;\ x{\in}\alpha ah\ w'\ m]\!] \Longrightarrow P$

$\rrbracket \Longrightarrow P$
by (*simp-all split*: *if-split-asm*)

lemma *mon-ah-subset*: *mon-ah* (αah *w*) $\subseteq$ *mon-pl w*
by (*induct w*) (*auto simp add*: *mon-ah-def*)

— Subwords generate smaller acquisition histories
lemma αah-*ileq*: $w1 \preceq w2 \Longrightarrow \alpha ah\ w1 \leq \alpha ah\ w2$
proof (*induct rule*: *less-eq-list-induct*)
case *empty* **thus** *?case* **by** (*unfold le-fun-def* [**where** $'b='a$ *set*], *simp*)
next
case (*drop* l' *l a*) **show** *?case*
proof (*unfold le-fun-def* [**where** $'b='a$ *set*], *intro allI subsetI*)
fix *m x*
assume *A*: $x \in \alpha ah\ l'\ m$
with *drop*(*2*) **have** $x \in \alpha ah\ l\ m$ **by** (*unfold le-fun-def* [**where** $'b='a$ *set*], *auto*)
moreover hence $x \in$*mon-pl l* **using** *mon-ah-subset*[*unfolded mon-ah-def*] **by** *fast*
ultimately show $x \in \alpha ah$ (a # l) *m* **by** *auto*
qed
next
case (*take a b* l' *l*) **show** *?case*
proof (*unfold le-fun-def* [**where** $'b='a$ *set*], *intro allI subsetI*)
fix *m x*
assume *A*: $x \in \alpha ah$ (a#l') *m*
thus $x \in \alpha ah$ (b # l) *m*
proof (*cases rule*: αah-*cons-cases*)
case *hd*
with *mon-pl-ileq*[*OF take.hyps*(*2*)] **and** $\langle a = b \rangle$
show *?thesis* **by** *auto*
next
case *tl*
with *take.hyps*(*3*)[*unfolded le-fun-def* [**where** $'b='a$ *set*]] **and** $\langle a = b \rangle$
show *?thesis* **by** *auto*
qed
qed
qed

We can now prove the relation of monitor consistent interleavability and interleavability of the acquisition histories.

lemma *ah-interleavable1*:
$w \in w1 \otimes_\alpha w2 \Longrightarrow \alpha ah$ (*map* α *w1*) [$*$] αah (*map* α *w2*)
— The lemma is shown by induction on the structure of the monitor consistent interleaving operator
proof (*induct w* α *w1 w2 rule*: *cil-set-induct-fix*α)
case *empty* **show** *?case* **by** (*simp add*: *ah-il-def*) — The base case is trivial by the definition of ([$*$])
next
— Case: First step comes from the left word

case (*left e w' w1' w2*) **show** *?case*
proof (*rule ccontr*) — We do a proof by contradiction
— Assume there is a conflicting pair in the acquisition histories
assume $\neg$ α*ah* (*map* α (*e* # *w1'*)) [*] α*ah* (*map* α *w2*)
then obtain *m1 m2* **where** *CPAIR*: *m1* $\in$ α*ah* (*map* α (*e*#*w1'*)) *m2* *m2* $\in$ α*ah* (*map* α *w2*) *m1* **by** (*unfold ah-il-def*, *blast*)
— It comes either from the first step or not
from *CPAIR*(*1*) **have** (*m2*$\in$*fst* (α *e*) $\wedge$ *m1* $\in$ *fst* (α *e*) $\cup$ *snd* (α *e*) $\cup$ *mon-pl* (*map* α *w1'*)) $\vee$ (*m2*$\notin$*fst* (α *e*) $\wedge$ *m1* $\in$ α*ah* (*map* α *w1'*) *m2*) (**is** *?CASE1* $\vee$ *?CASE2*)
by (*auto split*: *if-split-asm*)
moreover {
— Case: One monitor of the conflicting pair is entered in the first step of the left path
assume *?CASE1* **hence** *C*: *m2*$\in$*fst* (α *e*) **..**
— Because the paths are consistently interleavable, the monitors entered in the first step must not occur in the other path
from *left*(*2*) *mon-ah-subset*[*of map* α *w2*] **have** *fst* (α *e*) $\cap$ *mon-ah* (α*ah* (*map* α *w2*)) = {} **by** *auto*
— But this is a contradiction to being a conflicting pair
with *C CPAIR*(*2*) **have** *False* **by** (*unfold mon-ah-def*, *blast*)
} moreover {
— Case: The first monitor of the conflicting pair is entered after the first step of the left path
assume *?CASE2* **hence** *C*: *m1* $\in$ α*ah* (*map* α *w1'*) *m2* **..**
— But this is a contradiction to the induction hypothesis, that says that the acquisition histories of the tail of the left path and the right path are interleavable
with *left*(*3*) *CPAIR*(*2*) **have** *False* **by** (*unfold ah-il-def*, *blast*)
} ultimately show *False* **..**
qed
next
— Case: First step comes from the right word. This case is shown completely analogous
case (*right e w' w2' w1*) **show** *?case*
proof (*rule ccontr*)
assume $\neg$ α*ah* (*map* α *w1*) [*] α*ah* (*map* α (*e*#*w2'*))
then obtain *m1 m2* **where** *CPAIR*: *m1* $\in$ α*ah* (*map* α *w1*) *m2* *m2* $\in$ α*ah* (*map* α (*e*#*w2'*)) *m1* **by** (*unfold ah-il-def*, *blast*)
from *CPAIR*(*2*) **have** (*m1*$\in$*fst* (α *e*) $\wedge$ *m2* $\in$ *fst* (α *e*) $\cup$ *snd* (α *e*) $\cup$ *mon-pl* (*map* α *w2'*)) $\vee$ (*m1*$\notin$*fst* (α *e*) $\wedge$ *m2* $\in$ α*ah* (*map* α *w2'*) *m1*) (**is** *?CASE1* $\vee$ *?CASE2*)
by (*auto split*: *if-split-asm*)
moreover {
assume *?CASE1* **hence** *C*: *m1*$\in$*fst* (α *e*) **..**
from *right*(*2*) *mon-ah-subset*[*of map* α *w1*] **have** *fst* (α *e*) $\cap$ *mon-ah* (α*ah* (*map* α *w1*)) = {} **by** *auto*
with *C CPAIR*(*1*) **have** *False* **by** (*unfold mon-ah-def*, *blast*)
} moreover {
assume *?CASE2* **hence** *C*: *m2* $\in$ α*ah* (*map* α *w2'*) *m1* **..**

with *right*(*3*) *CPAIR*(*1*) **have** *False* **by** (*unfold ah-il-def*, *blast*)
} **ultimately show** *False* **..**
qed
qed

lemma *ah-interleavable2*:
assumes *A*: αah (*map* α *w1*) [$*$] αah (*map* α *w2*)
shows *w1* $\otimes_\alpha$ *w2* $\neq$ {}
— This lemma is shown by induction on the sum of the word lengths
proof –
— To apply this induction in Isabelle, we have to rewrite the lemma a bit
{ **fix** *n*
have !!*w1 w2*. ⟦αah (*map* α *w1*) [$*$] αah (*map* α *w2*); *n*=*length w1* + *length w2*⟧ $\Longrightarrow$ *w1* $\otimes_\alpha$ *w2* $\neq$ {}
proof (*induct n rule*: *nat-less-induct*[*case-names I*])
— We first rule out the cases that one of the words is empty
case (*I n w1 w2*) **show** *?case* **proof** (*cases w1*)
— If the first word is empty, the lemma is trivial
case *Nil* **with** *I.prems* **show** *?thesis* **by** *simp*
next
case (*Cons e1 w1'*) **note** *CONS1*=*this* **show** *?thesis* **proof** (*cases w2*)
— If the second word is empty, the lemma is also trivial
case *Nil* **with** *I.prems* **show** *?thesis* **by** *simp*
next
— The interesting case is if both words are not empty
case (*Cons e2 w2'*) **note** *CONS2*=*this*
— In this case, we check whether the first step of one of the words can safely be executed without blocking any steps of the other word
show *?thesis* **proof** (*cases fst* (α *e1*) $\cap$ *mon-pl* (*map* α *w2*) = {})
case *True* — The first step of the first word can safely be executed
— From the induction hypothesis, we get that there is a consistent interleaving of the rest of the first word and the second word
have *w1'*$\otimes_\alpha$*w2* $\neq$ {} **proof** –
from *I.prems*(*1*) *CONS1 ah-leq-il-left*[*OF* - α*ah-ileq*[*OF le-list-map*, *OF less-eq-list-drop*[*OF order-refl*]]] **have** αah (*map* α *w1'*) [$*$] αah (*map* α *w2*) **by** *fast*
moreover from *CONS1 I.prems*(*2*) **have** *length w1'*+*length w2* < *n* **by** *simp*
ultimately show *?thesis* **using** *I.hyps* **by** *blast*
qed
— And because the first step of the first word can be safely executed, we can prepend it to that consistent interleaving
with *cil-cons1*[*OF* - *True*] *CONS1* **show** *?thesis* **by** *blast*
next
case *False* **note** *C1*=*this*
show *?thesis* **proof** (*cases fst* (α *e2*) $\cap$ *mon-pl* (*map* α *w1*) = {})
case *True* — The first step of the second word can safely be executed
— This case is shown analogously to the latter one

have $w1 \otimes_\alpha w2' \neq \{\}$ **proof** −
from *I.prems*(*1*) *CONS2 ah-leq-il-right*[*OF - αah-ileq*[*OF le-list-map*, *OF less-eq-list-drop*[*OF order-refl*]]] **have** αah (*map* α *w1*) [∗] αah (*map* α *w2′*) **by** *fast*
moreover from *CONS2 I.prems*(*2*) **have** *length w1*+*length w2′* < *n* **by** *simp*
ultimately show *?thesis* **using** *I.hyps* **by** *blast*
qed
with *cil-cons2*[*OF - True*] *CONS2* **show** *?thesis* **by** *blast*
next
case *False* **note** *C2*=*this* — Neither first step can safely be executed. This is exactly the situation from that we can extract a conflicting pair
from *C1 C2* **obtain** *m1 m2* **where** *m1*∈*fst* (α *e1*) *m1*∈*mon-pl* (*map* α *w2*) *m2*∈*fst* (α *e2*) *m2*∈*mon-pl* (*map* α *w1*) **by** *blast*
with *CONS1 CONS2* **have** *m2* ∈ αah (*map* α *w1*) *m1 m1* ∈ αah (*map* α *w2*) *m2* **by** *auto*
— But by assumption, there are no conflicting pairs, thus we get a contradiction
with *I.prems*(*1*) **have** *False* **by** (*unfold ah-il-def*) *blast*
thus *?thesis* **..**
qed
qed
qed
qed
qed
} **with** *A* **show** *?thesis* **by** *blast*
qed

Finally, we can state the relationship between monitor consistent interleaving and interleaving of acquisition histories

theorem *ah-interleavable*:
(αah (*map* α *w1*) [∗] αah (*map* α *w2*)) ⟷ ($w1 \otimes_\alpha w2 \neq \{\}$)
using *ah-interleavable1 ah-interleavable2* **by** *blast*

3.6 Acquisition history backward update

We define a function to update an acquisition history backwards. This function is useful for constructing acquisition histories in backward constraint systems.

definition
ah-update :: ($'m \Rightarrow 'm$ *set*) ⇒ ($'m$ *set* ∗ $'m$ *set*) ⇒ $'m$ *set* ⇒ ($'m \Rightarrow 'm$ *set*)
where
ah-update h F M m == *if m*∈*fst F then fst F* ∪ *snd F* ∪ *M else h m*

Intuitively, *ah-update h* (*E*, *U*) *M m* means to prepend a step (*E*, *U*) to the acquisition history *h* of a path that uses monitors *M*. Note that we need the extra parameter *M*, since an acquisition history does not contain information about the monitors that are used on a path before the first monitor that will not be left has been entered.

lemma *ah-update-cons*: $\alpha ah\ (e\#w) = ah\text{-}update\ (\alpha ah\ w)\ e\ (mon\text{-}pl\ w)$
  by (*auto intro*!: *ext simp add*: *ah-update-def*)

The backward-update function is monotonic in the first and third argument as well as in the used monitors of the second argument. Note that it is, in general, not monotonic in the entered monitors of the second argument.

lemma *ah-update-mono*: $[\![h \le h'; F{=}F'; M{\subseteq}M']\!]$
  $\Longrightarrow ah\text{-}update\ h\ F\ M \le ah\text{-}update\ h'\ F'\ M'$
  by (*auto simp add*: *ah-update-def le-fun-def* [**where** $'b{=}'a\ set$])
lemma *ah-update-mono2*: $[\![h \le h'; U{\subseteq}U'; M{\subseteq}M']\!]$
  $\Longrightarrow ah\text{-}update\ h\ (E,U)\ M \le ah\text{-}update\ h'\ (E,U')\ M'$
  by (*auto simp add*: *ah-update-def le-fun-def* [**where** $'b{=}'a\ set$])

end

4 Labeled transition systems

theory *LTS*
imports *Main*
begin

Labeled transition systems (LTS) provide a model of a state transition system with named transitions.

4.1 Definitions

An LTS is modeled as a ternary relation between start configuration, transition label and end configuration

type-synonym $('c,'a)\ LTS = ('c \times 'a \times 'c)\ set$

Transitive reflexive closure

inductive-set
  $trcl :: ('c,'a)\ LTS \Rightarrow ('c,'a\ list)\ LTS$
  for t
  where
  empty[*simp*]: $(c,[],c) \in trcl\ t$
  | *cons*[*simp*]: $[\![\ (c,a,c') \in t;\ (c',w,c'') \in trcl\ t\ ]\!] \Longrightarrow (c,a\#w,c'') \in trcl\ t$

4.2 Basic properties of transitive reflexive closure

lemma *trcl-empty-cons*: $(c,[],c'){\in}trcl\ t \Longrightarrow (c{=}c')$
  by (*auto elim*: *trcl.cases*)
lemma *trcl-empty-simp*[*simp*]: $(c,[],c'){\in}trcl\ t = (c{=}c')$
  by (*auto elim*: *trcl.cases intro*: *trcl.intros*)

lemma *trcl-single*[*simp*]: $((c,[a],c') \in trcl\ t) = ((c,a,c') \in t)$
  by (*auto elim*: *trcl.cases*)
lemma *trcl-uncons*: $(c,a\#w,c')\in trcl\ t \Longrightarrow \exists\, ch\ .\ (c,a,ch)\in t \wedge (ch,w,c') \in trcl\ t$
  by (*auto elim*: *trcl.cases*)
lemma *trcl-uncons-cases*: $[\![$
    $(c,e\#w,c')\in trcl\ S$;
    $!!ch.\ [\![\ (c,e,ch)\in S;\ (ch,w,c')\in trcl\ S\ ]\!] \Longrightarrow P$
  $]\!] \Longrightarrow P$
  by (*blast dest*: *trcl-uncons*)
lemma *trcl-one-elem*: $(c,e,c')\in t \Longrightarrow (c,[e],c')\in trcl\ t$
  by *auto*

lemma *trcl-unconsE*[*cases set*, *case-names split*]: $[\![$
    $(c,e\#w,c')\in trcl\ S$;
    $!!ch.\ [\![(c,e,ch)\in S;\ (ch,w,c')\in trcl\ S]\!] \Longrightarrow P$
  $]\!] \Longrightarrow P$
  by (*blast dest*: *trcl-uncons*)
lemma *trcl-pair-unconsE*[*cases set*, *case-names split*]: $[\![$
    $((s,c),e\#w,(s',c'))\in trcl\ S$;
    $!!sh\ ch.\ [\![((s,c),e,(sh,ch))\in S;\ ((sh,ch),w,(s',c'))\in trcl\ S]\!] \Longrightarrow P$
  $]\!] \Longrightarrow P$
  by (*fast dest*: *trcl-uncons*)

lemma *trcl-concat*: $!!\ c\ .\ [\![\ (c,w1,c')\in trcl\ t;\ (c',w2,c'')\in trcl\ t\ ]\!]$
  $\Longrightarrow (c,w1@w2,c'')\in trcl\ t$
proof (*induct w1*)
  case *Nil* **thus** *?case* **by** (*subgoal-tac* $c=c'$) *auto*
next
  case (*Cons a w*) **thus** *?case* **by** (*auto dest*: *trcl-uncons*)
qed

lemma *trcl-unconcat*: $!!\ c\ .\ (c,w1@w2,c')\in trcl\ t$
  $\Longrightarrow \exists\, ch\ .\ (c,w1,ch)\in trcl\ t \wedge (ch,w2,c')\in trcl\ t$
proof (*induct w1*)
  case *Nil* **hence** $(c,[],c)\in trcl\ t \wedge (c,w2,c')\in trcl\ t$ **by** *auto*
  thus *?case* **by** *fast*
next
  case (*Cons a w1*) **note** *IHP* = *this*
  hence $(c,a\#(w1@w2),c')\in trcl\ t$ **by** *simp*
  with *trcl-uncons* **obtain** *chh* **where** $(c,a,chh)\in t \wedge (chh,w1@w2,c')\in trcl\ t$ **by** *fast*
  moreover with *IHP* **obtain** *ch* **where** $(chh,w1,ch)\in trcl\ t \wedge (ch,w2,c')\in trcl\ t$
by *fast*
  ultimately have $(c,a\#w1,ch)\in trcl\ t \wedge (ch,w2,c')\in trcl\ t$ **by** *auto*
  thus *?case* **by** *fast*
qed

4.2.1 Appending of elements to paths

lemma *trcl-rev-cons*: $\llbracket\ (c,w,ch)\in trcl\ T;\ (ch,e,c')\in T\ \rrbracket \Longrightarrow (c,w@[e],c')\in trcl\ T$
  by (*auto dest*: *trcl-concat iff add*: *trcl-single*)
lemma *trcl-rev-uncons*: $(c,w@[e],c')\in trcl\ T$
  $\Longrightarrow \exists\, ch.\ (c,w,ch)\in trcl\ T \wedge (ch,e,c')\in T$
  by (*force dest*: *trcl-unconcat*)
lemma *trcl-rev-induct*[*induct set*, *consumes 1*, *case-names empty snoc*]: !! c'. $\llbracket$
    $(c,w,c')\in trcl\ S$;
    !!$c.\ P\ c\ []\ c$;
    !!$c\ w\ c'\ e\ c''.\ \llbracket\ (c,w,c')\in trcl\ S;\ (c',e,c'')\in S;\ P\ c\ w\ c'\ \rrbracket \Longrightarrow P\ c\ (w@[e])\ c''$
  $\rrbracket \Longrightarrow P\ c\ w\ c'$
  by (*induct w rule*: *rev-induct*) (*auto dest*: *trcl-rev-uncons*)
lemma *trcl-rev-cases*: !!$c\ c'$. $\llbracket$
    $(c,w,c')\in trcl\ S$;
    $\llbracket w=[];\ c=c'\rrbracket\Longrightarrow P$;
    !!$ch\ e\ wh.\ \llbracket\ w=wh@[e];\ (c,wh,ch)\in trcl\ S;\ (ch,e,c')\in S\ \rrbracket\Longrightarrow P$
  $\rrbracket \Longrightarrow P$
  by (*induct w rule*: *rev-induct*) (*auto dest*: *trcl-rev-uncons*)

lemma *trcl-cons2*: $\llbracket\ (c,e,ch)\in T;\ (ch,f,c')\in T\ \rrbracket \Longrightarrow (c,[e,f],c')\in trcl\ T$
  by *auto*

4.2.2 Transitivity reasoning setup

declare *trcl-cons2*[*trans*] — It's important that this is declared before *trcl-concat*, because we want *trcl-concat* to be tried first by the transitivity reasoner
declare *cons*[*trans*]
declare *trcl-concat*[*trans*]
declare *trcl-rev-cons*[*trans*]

4.2.3 Monotonicity

lemma *trcl-mono*: !!$A\ B.\ A \subseteq B \Longrightarrow trcl\ A \subseteq trcl\ B$
  apply (*clarsimp*)
  apply (*erule trcl.induct*)
  apply *auto*
done

lemma *trcl-inter-mono*: $x\in trcl\ (S\cap R) \Longrightarrow x\in trcl\ S$ $x\in trcl\ (S\cap R) \Longrightarrow x\in trcl\ R$
proof –
  assume $x\in trcl\ (S\cap R)$
  with *trcl-mono*[*of* $S\cap R\ S$] **show** $x\in trcl\ S$ **by** *auto*
next
  assume $x\in trcl\ (S\cap R)$
  with *trcl-mono*[*of* $S\cap R\ R$] **show** $x\in trcl\ R$ **by** *auto*
qed

4.2.4 Special lemmas for reasoning about states that are pairs

lemmas *trcl-pair-induct* = *trcl.induct*[*of* (*xc1*,*xc2*) *xb* (*xa1*,*xa2*), *split-format* (*complete*), *consumes 1*, *case-names empty cons*]
lemmas *trcl-rev-pair-induct* = *trcl-rev-induct*[*of* (*xc1*,*xc2*) *xb* (*xa1*,*xa2*), *split-format* (*complete*), *consumes 1*, *case-names empty snoc*]

4.2.5 Invariants

lemma *trcl-prop-trans*[*cases set*, *consumes 1*, *case-names empty steps*]: $\llbracket$
  $(c,w,c')\in trcl\ S$;
  $\llbracket c=c';\ w=[]\rrbracket \Longrightarrow P$;
  $\llbracket c\in Domain\ S;\ c'\in Range\ (Range\ S)\rrbracket\Longrightarrow P$
 $\rrbracket \Longrightarrow P$
 apply (*erule-tac trcl-rev-cases*)
 apply *auto*
 apply (*erule trcl.cases*)
 apply *auto*
 done

end

5 Thread Tracking

theory *ThreadTracking*
imports *Main HOL−Library.Multiset LTS Misc*
begin

This theory defines some general notion of an interleaving semantics. It defines how to extend a semantics specified on a single thread and a context to a semantic on multisets of threads. The context is needed in order to keep track of synchronization.

5.1 Semantic on multiset configuration

The interleaving semantics is defined on a multiset of stacks. The thread to make the next step is nondeterministically chosen from all threads ready to make steps.

definition
 $gtr\ gtrs == \{\ (add\text{-}mset\ s\ c,e,add\text{-}mset\ s'\ c')\ |\ s\ c\ e\ s'\ c'\ .\ ((s,c),e,(s',c'))\in gtrs\ \}$

lemma *gtrI-s*: $((s,c),e,(s',c'))\in gtrs \Longrightarrow (add\text{-}mset\ s\ c,e,add\text{-}mset\ s'\ c')\in gtr\ gtrs$
 by (*unfold gtr-def*, *auto*)
lemma *gtrI*: $((s,c),w,(s',c'))\in trcl\ gtrs$
 $\Longrightarrow (add\text{-}mset\ s\ c,w,add\text{-}mset\ s'\ c')\in trcl\ (gtr\ gtrs)$
 by (*induct rule*: *trcl-pair-induct*) (*auto dest*: *gtrI-s*)

lemma *gtrE*: ⟦
  $(c,e,c')\in gtr\ T$;
  $!!s\ ce\ s'\ ce'$. ⟦ $c=add\text{-}mset\ s\ ce$; $c'=add\text{-}mset\ s'\ ce'$; $((s,ce),e,(s',ce'))\in T$ ⟧ $\Longrightarrow$ P
 ⟧ $\Longrightarrow P$
 by (*unfold gtr-def*) *auto*

lemma *gtr-empty-conf-s*[*simp*]:
 $(\{\#\},w,c')\notin gtr\ S$
 $(c,w,\{\#\})\notin gtr\ S$
 by (*auto elim*: *gtrE*)
lemma *gtr-empty-conf1*[*simp*]: $((\{\#\},w,c')\in trcl\ (gtr\ S)) \longleftrightarrow (w=[] \wedge c'=\{\#\})$
 by (*induct w*) (*auto dest*: *trcl-uncons*)
lemma *gtr-empty-conf2*[*simp*]: $((c,w,\{\#\})\in trcl\ (gtr\ S)) \longleftrightarrow (w=[] \wedge c=\{\#\})$
 by (*induct w rule*: *rev-induct*) (*auto dest*: *trcl-rev-uncons*)

lemma *gtr-find-thread*: ⟦
  $(c,e,c')\in gtr\ gtrs$;
  $!!s\ ce\ s'\ ce'$. ⟦$c=add\text{-}mset\ s\ ce$; $c'=add\text{-}mset\ s'\ ce'$; $((s,ce),e,(s',ce'))\in gtrs$⟧ $\Longrightarrow$ P
 ⟧ $\Longrightarrow P$
 by (*unfold gtr-def*) *auto*

lemma *gtr-step-cases*[*cases set*, *case-names loc other*]: ⟦
  $(add\text{-}mset\ s\ ce,e,c')\in gtr\ gtrs$;
  $!!s'\ ce'$. ⟦ $c'=add\text{-}mset\ s'\ ce'$; $((s,ce),e,(s',ce'))\in gtrs$ ⟧ $\Longrightarrow P$;
  $!!cc\ ss\ ss'\ ce'$. ⟦ $ce=add\text{-}mset\ ss\ cc$; $c'=add\text{-}mset\ ss'\ ce'$;
      $((ss,add\text{-}mset\ s\ cc),e,(ss',ce'))\in gtrs$ ⟧ $\Longrightarrow P$
 ⟧ $\Longrightarrow P$
 by (*auto elim*!: *gtr-find-thread mset-single-cases*)

lemma *gtr-rev-cases*[*cases set*, *case-names loc other*]: ⟦
  $(c,e,add\text{-}mset\ s'\ ce')\in gtr\ gtrs$;
  $!!s\ ce$. ⟦ $c=add\text{-}mset\ s\ ce$; $((s,ce),e,(s',ce'))\in gtrs$ ⟧ $\Longrightarrow P$;
  $!!cc\ ss\ ss'\ ce$. ⟦ $c=add\text{-}mset\ ss\ ce$; $ce'=add\text{-}mset\ ss'\ cc$;
      $((ss,ce),e,(ss',add\text{-}mset\ s'\ cc))\in gtrs$ ⟧ $\Longrightarrow P$
 ⟧ $\Longrightarrow P$
 by (*auto elim*!: *gtr-find-thread mset-single-cases*)

5.2 Invariants

lemma *gtr-preserve-s*: ⟦
  $(c,e,c')\in gtr\ T$;
  $P\ c$;
  $!!s\ c\ s'\ c'\ e$. ⟦$P\ (add\text{-}mset\ s\ c)$; $((s,c),e,(s',c'))\in T$⟧ $\Longrightarrow P\ (add\text{-}mset\ s'\ c')$
 ⟧ $\Longrightarrow P\ c'$
 by (*unfold gtr-def*) *blast*

lemma *gtr-preserve*: ⟦

$(c,w,c')\in trcl\ (gtr\ T)$;
$P\ c$;
$!!s\ c\ s'\ c'\ e.\ [\![P\ (add\text{-}mset\ s\ c);\ ((s,c),e,(s',c'))\in T]\!] \Longrightarrow P\ (add\text{-}mset\ s'\ c')$
$]\!] \Longrightarrow P\ c'$
apply (*induct rule*: *trcl.induct*)
apply *simp*
apply (*subgoal-tac P c'*)
apply *blast*
apply (*blast intro*: *gtr-preserve-s*)
done

5.3 Context preservation assumption

We now assume that the original semantics does not modify threads in the context, i.e. it may only add new threads to the context and use the context to obtain monitor information, but not change any existing thread in the context. This assumption is valid for our semantics, where the context is just needed to determine the set of allocated monitors. It allows us to generally derive some further properties of such semantics.

locale *env-no-step* =
fixes *gtrs* :: $(('s\times's\ multiset),'l)\ LTS$
assumes *env-no-step-s*[*cases set*, *case-names csp*]:
$[\![((s,c),e,(s',c'))\in gtrs;\ !!csp.\ c'=csp+c \Longrightarrow P\ ]\!] \Longrightarrow P$

— The property of not changing existing threads in the context transfers to paths
lemma (**in** *env-no-step*) *env-no-step*[*cases set*, *case-names csp*]: $[\![$
$((s,c),w,(s',c'))\in trcl\ gtrs$;
$!!\ csp.\ c'=csp+c \Longrightarrow P$
$]\!] \Longrightarrow P$
proof –
have $((s,c),w,(s',c'))\in trcl\ gtrs \Longrightarrow \exists\ csp.\ c'=csp+c$ **proof** (*induct rule*: *trcl-pair-induct*)
case *empty* **thus** *?case* **by** (*auto intro*: *exI*[*of* - {#}])
next
case (*cons s c e sh ch w s' c'*) **note** *IHP=this*
from *env-no-step-s*[*OF IHP(1)*] **obtain** *csph* **where** *ch=csph+c* **by** *auto*
moreover from *IHP(3)* **obtain** *csp'* **where** *c'=csp'+ch* **by** *auto*
ultimately have *c'=csp'+csph+c* **by** (*simp add*: *union-assoc*)
thus *?case* **by** *blast*
qed
moreover assume $((s,c),w,(s',c'))\in trcl\ gtrs\ !!\ csp.\ c'=csp+c \Longrightarrow P$
ultimately show *?thesis* **by** *blast*
qed

The following lemma can be used to make a case distinction how a step operated on a given thread in the end configuration:

loc The thread made the step

spawn The thread was spawned by the step *env* The thread was not involved in the step

lemma (**in** *env-no-step*) *rev-cases-p*[*cases set*, *case-names loc spawn env*]:
  assumes *STEP*: $(c,e,\textit{add-mset } s' \; ce') \in gtr \; gtrs$ **and**
  LOC: $!!s \; ce. \; [\![\; c=\{\#s\#\}+ce; \; ((s,ce),e,(s',ce')) \in gtrs \;]\!] \Longrightarrow P$ **and**
  SPAWN: $!!ss \; ss' \; ce \; csp.$
    $[\![\; c=\textit{add-mset } ss \; ce; \; ce'=\textit{add-mset } ss' \; (csp+ce);$
    $((ss,ce),e,(ss',\textit{add-mset } s' \; (csp+ce))) \in gtrs \;]\!]$
    $\Longrightarrow P$ **and**
  ENV: $!!ss \; ss' \; ce \; csp.$
    $[\![\; c=\textit{add-mset } ss \; (\textit{add-mset } s' \; ce); \; ce'=\textit{add-mset } ss' \; (csp+ce);$
    $((ss,\textit{add-mset } s' \; ce),e,(ss',csp+(\textit{add-mset } s' \; ce))) \in gtrs \;]\!]$
    $\Longrightarrow P$
  shows *P*
proof (*rule gtr-rev-cases*[*OF STEP*], *goal-cases*)
  case *1* **thus** *?thesis* **using** *LOC* **by** *auto*
next
  case *CASE*: (*2 cc ss ss' ce*)
  hence *CASE'*: $c = \{\#ss\#\} + ce \; ce' = \{\#ss'\#\} + cc \; ((ss, ce), e, ss', \{\#s'\#\} + cc) \in gtrs$ **by** *simp-all*
  from *env-no-step-s*[*OF CASE'*(*3*)] **obtain** *csp* **where** *EQ*: $\textit{add-mset } s' \; cc = csp + ce$ **by** *auto*
  thus *?thesis* **proof** (*cases rule*: *mset-unplusm-dist-cases*)
    case *left* **note** *CC=this*
    with *CASE'* **have** $ce'=\{\#ss'\#\} + (csp-\{\#s'\#\}) + ce$ **by** (*auto simp add*: *union-assoc*)
    moreover from *CC*(*2*) **have** $\{\#s'\#\}+cc = \{\#s'\#\} + (csp-\{\#s'\#\}) + ce$ **by** (*simp add*: *union-assoc*)
    ultimately show *?thesis* **using** *CASE'*(*1,3*) *CASE*(*2*) *SPAWN* **by** *auto*
  next
    case *right* **note** *CC=this*
    from *CC*(*1*) *CASE'*(*1*) **have** $c=\textit{add-mset } ss \; (\textit{add-mset } s' \; (ce - \{\#s'\#\}))$ **by** (*simp add*: *union-assoc*)
    moreover from *CC*(*2*) *CASE'*(*2*) **have** $ce'=\textit{add-mset } ss' \; (csp+(ce-\{\#s'\#\}))$ **by** (*simp add*: *union-assoc*)
    moreover from *CC*(*2*) **have** $\textit{add-mset } s' \; cc = csp+(\textit{add-mset } s' \; (ce-\{\#s'\#\}))$ **by** (*simp add*: *union-ac*)
    ultimately show *?thesis* **using** *CASE'*(*3*) *CASE*(*3*) *CC*(*1*) *ENV* **by** *metis*
  qed
qed

5.4 Explicit local context

In the multiset semantics, a single thread has no identity. This may become a problem when reasoning about a fixed thread during an execution. For example, in our constraint-system-based approach the operational characterization of the least solution of the constraint system requires to state properties of the steps of the initial thread in some execution. With the multiset semantics, we are unable to identify those steps among all steps.

There are many solutions to this problem, for example, using thread ids either as part of the thread's configuration or as part of the whole configuration by using lists of stacks or maps from ids to stacks as configuration datatype.

In the following we present a special solution that is strong enough to suit our purposes but not meant as a general solution.

Instead of identifying every single thread uniquely, we only distinguish one thread as the *local* thread. The other threads are *environment* threads. We then attach to every step the information whether it was on the local or on some environment thread.

We call this semantics *loc/env*-semantics in contrast to the *multiset*-semantics of the last section.

5.4.1 Lifted step datatype

datatype *'a el-step* = *LOC 'a* | *ENV 'a*

definition
loc w == *filter* (*λe. case e of LOC a ⇒ True* | *ENV a ⇒ False*) *w*

definition
env w == *filter* (*λe. case e of LOC a ⇒ False* | *ENV a ⇒ True*) *w*

definition
le-rem-s e == *case e of LOC a ⇒ a* | *ENV a ⇒ a*

Standard simplification lemmas

lemma *loc-env-simps*[*simp*]:
loc [] = []
env [] = []
by (*unfold loc-def env-def*) *auto*

lemma *loc-single*[*simp*]: *loc* [*a*] = (*case a of LOC e ⇒* [*a*] | *ENV e ⇒* [])
by (*unfold loc-def*) (*auto split*: *el-step.split*)
lemma *loc-uncons*[*simp*]:
loc (*a#b*) = (*case a of LOC e ⇒* [*a*] | *ENV e ⇒* [])@*loc b*
by (*unfold loc-def*) (*auto split*: *el-step.split*)
lemma *loc-unconc*[*simp*]: *loc* (*a*@*b*) = *loc a* @ *loc b*
by (*unfold loc-def*, *simp*)

lemma *env-single*[*simp*]: *env* [*a*] = (*case a of LOC e ⇒* [] | *ENV e ⇒* [*a*])
by (*unfold env-def*) (*auto split*: *el-step.split*)
lemma *env-uncons*[*simp*]:
env (*a#b*) = (*case a of LOC e ⇒* [] | *ENV e ⇒* [*a*]) @ *env b*
by (*unfold env-def*) (*auto split*: *el-step.split*)
lemma *env-unconc*[*simp*]: *env* (*a*@*b*) = *env a* @ *env b*
by (*unfold env-def*, *simp*)

The following simplification lemmas are for converting between paths of the multiset- and loc/env-semantics

lemma *le-rem-simps* [*simp*]:
  le-rem-s (*LOC a*) = *a*
  le-rem-s (*ENV a*) = *a*
  by (*unfold le-rem-s-def*, *auto*)
lemma *le-rem-id-simps*[*simp*]:
  le-rem-s∘*LOC* = *id*
  le-rem-s∘*ENV* = *id*
  by (*auto intro*: *ext*)

lemma *le-rem-id-map*[*simp*]:
  map le-rem-s (*map LOC w*) = *w*
  map le-rem-s (*map ENV w*) = *w*
  by *auto*

lemma *env-map-env* [*simp*]: *env* (*map ENV w*) = *map ENV w*
  by (*unfold env-def*) *simp*
lemma *env-map-loc* [*simp*]: *env* (*map LOC w*) = []
  by (*unfold env-def*) *simp*
lemma *loc-map-env* [*simp*]: *loc* (*map ENV w*) = []
  by (*unfold loc-def*) *simp*
lemma *loc-map-loc* [*simp*]: *loc* (*map LOC w*) = *map LOC w*
  by (*unfold loc-def*) *simp*

5.4.2 Definition of the loc/env-semantics

type-synonym *′s el-conf* = (*′s* × *′s multiset*)

inductive-set
  gtrp :: (*′s el-conf*,*′l*) *LTS* ⇒ (*′s el-conf*,*′l el-step*) *LTS*
  for *S*
  where
  gtrp-loc: ((*s*,*c*),*e*,(*s′*,*c′*))∈*S* ⟹ ((*s*,*c*),*LOC e*,(*s′*,*c′*))∈*gtrp S*
  | *gtrp-env*: ((*s*,*add-mset sl c*),*e*,(*s′*,*add-mset sl c′*))∈*S*
        ⟹ ((*sl*,*add-mset s c*),*ENV e*,(*sl*,*add-mset s′ c′*))∈*gtrp S*

5.4.3 Relation between multiset- and loc/env-semantics

lemma *gtrp2gtr-s*:
  ((*s*,*c*),*e*,(*s′*,*c′*))∈*gtrp T* ⟹ (*add-mset s c*,*le-rem-s e*,*add-mset s′ c′*)∈*gtr T*
proof (*cases rule*: *gtrp.cases*, *auto intro*: *gtrI-s*)
  fix *c c′ e ss ss′* **assume** ((*ss*,*add-mset s c*),*e*,(*ss′*,*add-mset s c′*))∈*T*
  hence (*add-mset ss* (*add-mset s c*),*e*,*add-mset ss′* (*add-mset s c′*)) ∈ *gtr T* **by** (*auto intro*!: *gtrI-s*)
  thus (*add-mset s* (*add-mset ss c*), *e*, *add-mset s* (*add-mset ss′ c′*)) ∈ *gtr T* **by** (*auto simp add*: *add-mset-commute*)
qed

lemma *gtrp2gtr*:
$((s,c),w,(s',c'))\in trcl\ (gtrp\ T)$
$\Longrightarrow (add\text{-}mset\ s\ c, map\ le\text{-}rem\text{-}s\ w, add\text{-}mset\ s'\ c')\in trcl\ (gtr\ T)$
by (*induct rule*: *trcl-pair-induct*) (*auto dest*: *gtrp2gtr-s*)

lemma (**in** *env-no-step*) *gtr2gtrp-s*[*cases set*, *case-names gtrp*]:
assumes *A*: $(add\text{-}mset\ s\ c,e,c')\in gtr\ gtrs$
and *CASE*: $!!s'\ ce'\ ee.\ [\![c'=add\text{-}mset\ s'\ ce';\ e=le\text{-}rem\text{-}s\ ee;$
$((s,c),ee,(s',ce'))\in gtrp\ gtrs]\!]$
$\Longrightarrow P$
shows *P*
using *A*
proof (*cases rule*: *gtr-step-cases*)
case (*loc* s' ce') **hence** $((s,c),LOC\ e,(s',ce'))\in gtrp\ gtrs$ **by** (*blast intro*: *gtrp-loc*)
with *loc*(*1*) **show** *?thesis* **by** (*rule-tac CASE*) *auto*
next
case (*other cc ss ss' ce'*) **from** *env-no-step-s*[*OF other*(*3*)] **obtain** *csp* **where** *CE'FMT*: $ce'=csp + (add\text{-}mset\ s\ cc)$ **.**
with *other*(*3*) **have** $((ss,add\text{-}mset\ s\ cc),e,(ss',add\text{-}mset\ s\ (csp+cc)))\in gtrs$ **by** *auto*
from *gtrp-env*[*OF this*] *other*(*1*) **have** $((s,\ c),\ ENV\ e,\ s,\ \{\#ss'\#\} + (csp + cc)) \in gtrp\ gtrs$ **by** *simp*
moreover from *other CE'FMT* **have** $c' = \{\#s\#\} + (\{\#ss'\#\} + (csp + cc))$ **by** (*simp add*: *union-ac*)
ultimately show *?thesis* **by** (*rule-tac CASE*) *auto*
qed

lemma (**in** *env-no-step*) *gtr2gtrp*[*cases set*, *case-names gtrp*]:
assumes *A*: $(add\text{-}mset\ s\ c,w,c')\in trcl\ (gtr\ gtrs)$
and *CASE*: $!!s'\ ce'\ ww.\ [\![c'=add\text{-}mset\ s'\ ce';\ w=map\ le\text{-}rem\text{-}s\ ww;$
$((s,c),ww,(s',ce'))\in trcl\ (gtrp\ gtrs)]\!]$
$\Longrightarrow P$
shows *P*
proof –
have $!!s\ c.\ (add\text{-}mset\ s\ c,w,c')\in trcl\ (gtr\ gtrs) \Longrightarrow \exists s'\ ce'\ ww.\ c'=add\text{-}mset\ s'\ ce' \wedge w=map\ le\text{-}rem\text{-}s\ ww \wedge ((s,c),ww,(s',ce'))\in trcl\ (gtrp\ gtrs)$ **proof** (*induct w*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons e w*) **then obtain** *ch* **where** *SPLIT*: $(add\text{-}mset\ s\ c,e,ch)\in gtr\ gtrs$ $(ch,w,c')\in trcl\ (gtr\ gtrs)$ **by** (*auto dest*: *trcl-uncons*)
from *gtr2gtrp-s*[*OF SPLIT*(*1*)] **obtain** *sh ceh ee* **where** *FS*: $ch = add\text{-}mset\ sh\ ceh$ $e = le\text{-}rem\text{-}s\ ee$ $((s,\ c),\ ee,\ sh,\ ceh) \in gtrp\ gtrs$ **by** *blast*
moreover from *FS*(*1*) *SPLIT*(*2*) *Cons.hyps* **obtain** *s' ce' ww* **where** *IH*: $c'=add\text{-}mset\ s'\ ce'$ $w=map\ le\text{-}rem\text{-}s\ ww$ $((sh,ceh),ww,(s',ce'))\in trcl\ (gtrp\ gtrs)$ **by** *blast*
ultimately have $((s,c),ee\#ww,(s',ce'))\in trcl\ (gtrp\ gtrs)$ $e\#w = map\ le\text{-}rem\text{-}s\ (ee\#ww)$ **by** *auto*
with *IH*(*1*) **show** *?case* **by** *iprover*
qed

with *A CASE* **show** *?thesis* **by** *blast*
qed

5.4.4 Invariants

lemma *gtrp-preserve-s*:
assumes *A*: $((s,c),e,(s',c'))\in gtrp\ T$
and *INIT*: $P\ (add\text{-}mset\ s\ c)$
and *PRES*: $!!s\ c\ s'\ c'\ e.\ [\![P\ (add\text{-}mset\ s\ c);\ ((s,c),e,(s',c'))\in T]\!]$
$\Longrightarrow P\ (add\text{-}mset\ s'\ c')$
shows $P\ (add\text{-}mset\ s'\ c')$
proof –
from *gtr-preserve-s*[*OF gtrp2gtr-s*[*OF A*], **where** $P=P$, *OF INIT*] *PRES* **show** $P\ (add\text{-}mset\ s'\ c')$ **by** *blast*
qed

lemma *gtrp-preserve*:
assumes *A*: $((s,c),w,(s',c'))\in trcl\ (gtrp\ T)$
and *INIT*: $P\ (add\text{-}mset\ s\ c)$
and *PRES*: $!!s\ c\ s'\ c'\ e.\ [\![P\ (add\text{-}mset\ s\ c);\ ((s,c),e,(s',c'))\in T]\!]$
$\Longrightarrow P\ (add\text{-}mset\ s'\ c')$
shows $P\ (add\text{-}mset\ s'\ c')$
proof –
from *gtr-preserve*[*OF gtrp2gtr*[*OF A*], **where** $P=P$, *OF INIT*] *PRES* **show** $P\ (add\text{-}mset\ s'\ c')$ **by** *blast*
qed

end

6 Flowgraphs

theory *Flowgraph*
imports *Main Misc*
begin

We use a flowgraph-based program model that extends the one we used previously [6]. A program is represented as an edge annotated graph and a set of procedures. The nodes of the graph are partitioned by the procedures, i.e. every node belongs to exactly one procedure. There are no edges between nodes of different procedures. Every procedure has a distinguished entry and return node and a set of monitors it synchronizes on. Additionally, the program has a distinguished *main* procedure. The edges are annotated with statements. A statement is either a base statement, a procedure call or a thread creation (spawn). Procedure calls and thread creations refer to the called procedure or to the initial procedure of the spawned thread, respectively.

We require that the main procedure and any initial procedure of a spawned thread does not to synchronize on any monitors. This avoids that spawning of a procedure together with entering a monitor is available in our model as an atomic step, which would be an unrealistic assumption for practical problems. Technically, our model would become strictly more powerful without this assumption.

If we allowed this, our model would become strictly more powerful,

6.1 Definitions

datatype $('p,'ba)$ *edgeAnnot* = *Base* $'ba$ | *Call* $'p$ | *Spawn* $'p$
type-synonym $('n,'p,'ba)$ *edge* = $('n \times ('p,'ba)$ *edgeAnnot* $\times\ 'n)$

record $('n,'p,'ba,'m)$ *flowgraph-rec* =
  edges :: $('n,'p,'ba)$ *edge set* — Set of annotated edges
  main :: $'p$ — Main procedure
  entry :: $'p \Rightarrow 'n$ — Maps a procedure to its entry point
  return :: $'p \Rightarrow 'n$ — Maps a procedure to its return point
  mon :: $'p \Rightarrow 'm$ *set* — Maps procedures to the set of monitors they allocate
  proc-of :: $'n \Rightarrow 'p$ — Maps a node to the procedure it is contained in

definition
  initialproc fg p == *p*=*main fg* ∨ (∃ *u v*. (*u*,*Spawn p*,*v*)∈*edges fg*)

lemma *main-is-initial*[*simp*]: *initialproc fg* (*main fg*)
  by (*unfold initialproc-def*) *simp*

locale *flowgraph* =
  fixes *fg* :: $('n,'p,'ba,'m,'more)$ *flowgraph-rec-scheme* (**structure**)

  — Edges are inside procedures only
  assumes *edges-part*: (*u*,*a*,*v*)∈*edges fg* ⟹ *proc-of fg u* = *proc-of fg v*
  — The entry point of a procedure must be in that procedure
  assumes *entry-valid*[*simp*]: *proc-of fg* (*entry fg p*) = *p*
  — The return point of a procedure must be in that procedure
  assumes *return-valid*[*simp*]: *proc-of fg* (*return fg p*) = *p*
  — Initial procedures do not synchronize on any monitors
  assumes *initial-no-mon*[*simp*]: *initialproc fg p* ⟹ *mon fg p* = {}

6.2 Basic properties

lemma (**in** *flowgraph*) *spawn-no-mon*[*simp*]:
  (*u*, *Spawn p*, *v*) ∈ *edges fg* ⟹ *mon fg p* = {}
  using *initial-no-mon* **by** (*unfold initialproc-def*, *blast*)
lemma (**in** *flowgraph*) *main-no-mon*[*simp*]: *mon fg* (*main fg*) = {}
  using *initial-no-mon* **by** (*unfold initialproc-def*, *blast*)

lemma (**in** *flowgraph*) *entry-return-same-proc*[*simp*]:

entry fg p = *return fg p'* ⟹ *p*=*p'*
apply (*subgoal-tac proc-of fg* (*entry fg p*) = *proc-of fg* (*return fg p'*))
apply (*simp* (*no-asm-use*))
by *simp*

lemma (**in** *flowgraph*) *entry-entry-same-proc*[*simp*]:
entry fg p = *entry fg p'* ⟹ *p*=*p'*
apply (*subgoal-tac proc-of fg* (*entry fg p*) = *proc-of fg* (*entry fg p'*))
apply (*simp* (*no-asm-use*))
by *simp*

lemma (**in** *flowgraph*) *return-return-same-proc*[*simp*]:
return fg p = *return fg p'* ⟹ *p*=*p'*
apply (*subgoal-tac proc-of fg* (*return fg p*) = *proc-of fg* (*entry fg p'*))
apply (*simp* (*no-asm-use*))
by *simp*

6.3 Extra assumptions for flowgraphs

In order to simplify the definition of our restricted schedules (cf. Section 8), we make some extra constraints on flowgraphs. Note that these are no real restrictions, as we can always rewrite flowgraphs to match these constraints, preserving the set of conflicts. We leave it to future work to consider such a rewriting formally.

The background of this restrictions is that we want to start an execution of a thread with a procedure call that never returns. This will allow easier technical treatment in Section 8. Here we enforce this semantic restrictions by syntactic properties of the flowgraph.

The return node of a procedure is called *isolated*, if it has no incoming edges and is different from the entry node. A procedure with an isolated return node will never return. See Section 8.1 for a proof of this.

definition
isolated-ret fg p ==
(∀ *u l*. ¬(*u*,*l*,*return fg p*)∈*edges fg*) ∧ *entry fg p* ≠ *return fg p*

The following syntactic restrictions guarantee that each thread's execution starts with a non-returning call. See Section 8.1 for a proof of this.

locale *eflowgraph* = *flowgraph* +
— Initial procedure's entry node isn't equal to its return node
assumes *initial-no-ret*: *initialproc fg p* ⟹ *entry fg p* ≠ *return fg p*
— The only outgoing edges of initial procedures' entry nodes are call edges to procedures with isolated return node
assumes *initial-call-no-ret*: ⟦*initialproc fg p*; (*entry fg p*,*l*,*v*)∈*edges fg*⟧
⟹ ∃ *p'*. *l*=*Call p'* ∧ *isolated-ret fg p'*

6.4 Example Flowgraph

This section contains a check that there exists a (non-trivial) flowgraph, i.e. that the assumptions made in the *flowgraph* and *eflowgraph* locales are consistent and have at least one non-trivial model.

```
definition
  example-fg == (|
    edges = {((0::nat,0::nat),Call 1,(0,1)), ((1,0),Spawn 0,(1,0)),
             ((1,0),Call 0, (1,0))},
    main = 0,
    entry = λp. (p,0),
    return = λp. (p,1),
    mon = λp. if p=1 then {0} else {},
    proc-of= λ (p,x). p |)

lemma exists-eflowgraph: eflowgraph example-fg
  apply (unfold-locales)
  apply (unfold example-fg-def)
  apply simp
  apply fast
  apply simp
  apply simp
  apply (simp add: initialproc-def)
  apply (simp add: initialproc-def)
  apply (simp add: initialproc-def isolated-ret-def)
  done


end
```

7 Operational Semantics

```
theory Semantics
imports Main Flowgraph HOL−Library.Multiset LTS Interleave ThreadTracking
begin
```

7.1 Configurations and labels

The state of a single thread is described by a stack of control nodes. The top node is the current control node and the nodes deeper in the stack are stored return addresses. The configuration of a whole program is described by a multiset of stacks.

Note that we model stacks as lists here, the first element being the top element.

type-synonym *'n conf* = (*'n list*) *multiset*

A step is labeled according to the executed edge. Additionally, we introduce a label for a procedure return step, that has no corresponding edge.

datatype $('p,'ba)$ *label* = *LBase 'ba* | *LCall 'p* | *LRet* | *LSpawn 'p*

7.2 Monitors

The following defines the monitors of nodes, stacks, configurations, step labels and paths (sequences of step labels)

definition
— The monitors of a node are the monitors the procedure of the node synchronizes on
mon-n fg n == *mon fg* (*proc-of fg n*)

definition
— The monitors of a stack are the monitors of all its nodes
mon-s fg s == $\bigcup$ { *mon-n fg n* | *n* . *n* ∈ *set s* }

definition
— The monitors of a configuration are the monitors of all its stacks
mon-c fg c == $\bigcup$ { *mon-s fg s* | *s* . *s* ∈# *c* }

— The monitors of a step label are the monitors of procedures that are called by this step
definition *mon-e* :: ('b, 'c, 'd, 'a, 'e) *flowgraph-rec-scheme* ⇒ ('c, 'f) *label* ⇒ 'a *set* **where**
mon-e fg e = (*case e of* (*LCall p*) ⇒ *mon fg p* | - ⇒ {})

lemma *mon-e-simps* [*simp*]:
mon-e fg (*LBase a*) = {}
mon-e fg (*LCall p*) = *mon fg p*
mon-e fg (*LRet*) = {}
mon-e fg (*LSpawn p*) = {}
by (*simp-all add*: *mon-e-def*)

— The monitors of a path are the monitors of all procedures that are called on the path
definition
mon-w fg w == $\bigcup$ { *mon-e fg e* | *e*. *e* ∈ *set w*}

lemma *mon-s-alt*: *mon-s fg s* == $\bigcup$ (*mon fg* ‘ *proc-of fg* ‘ *set s*)
by (*unfold mon-s-def mon-n-def*) (*auto intro*!: *eq-reflection*)
lemma *mon-c-alt*: *mon-c fg c* == $\bigcup$ (*mon-s fg* ‘ *set-mset c*)
by (*unfold mon-c-def set-mset-def*) (*auto intro*!: *eq-reflection*)
lemma *mon-w-alt*: *mon-w fg w* == $\bigcup$ (*mon-e fg* ‘ *set w*)
by (*unfold mon-w-def*) (*auto intro*!: *eq-reflection*)

lemma *mon-sI*: ⟦*n*∈*set s*; *m*∈*mon-n fg n*⟧ ⟹ *m*∈*mon-s fg s*
by (*unfold mon-s-def*, *auto*)
lemma *mon-sD*: *m*∈*mon-s fg s* ⟹ ∃ *n*∈*set s*. *m*∈*mon-n fg n*

by (*unfold mon-s-def*, *auto*)

lemma *mon-n-same-proc*:
proc-of fg n = *proc-of fg n′* $\Longrightarrow$ *mon-n fg n* = *mon-n fg n′*
by (*unfold mon-n-def*, *simp*)
lemma *mon-s-same-proc*:
proc-of fg ‘ set s = *proc-of fg ‘ set s′* $\Longrightarrow$ *mon-s fg s* = *mon-s fg s′*
by (*unfold mon-s-alt*, *simp*)

lemma (**in** *flowgraph*) *mon-of-entry*[*simp*]: *mon-n fg* (*entry fg p*) = *mon fg p*
by (*unfold mon-n-def*, *simp add*: *entry-valid*)
lemma (**in** *flowgraph*) *mon-of-ret*[*simp*]: *mon-n fg* (*return fg p*) = *mon fg p*
by (*unfold mon-n-def*, *simp add*: *return-valid*)

lemma *mon-c-single*[*simp*]: *mon-c fg* {#*s*#} = *mon-s fg s*
by (*unfold mon-c-def*) *auto*
lemma *mon-s-single*[*simp*]: *mon-s fg* [*n*] = *mon-n fg n*
by (*unfold mon-s-def*) *auto*
lemma *mon-s-empty*[*simp*]: *mon-s fg* [] = {}
by (*unfold mon-s-def*) *auto*
lemma *mon-c-empty*[*simp*]: *mon-c fg* {#} = {}
by (*unfold mon-c-def*) *auto*

lemma *mon-s-unconc*: *mon-s fg* (*a*@*b*) = *mon-s fg a* ∪ *mon-s fg b*
by (*unfold mon-s-def*) *auto*
lemma *mon-s-uncons*[*simp*]: *mon-s fg* (*a*#*as*) = *mon-n fg a* ∪ *mon-s fg as*
by (*rule mon-s-unconc*[**where** *a*=[*a*], *simplified*])

lemma *mon-c-union-conc*: *mon-c fg* (*a*+*b*) = *mon-c fg a* ∪ *mon-c fg b*
by (*unfold mon-c-def*) *auto*

lemma *mon-c-add-mset-unconc*: *mon-c fg* (*add-mset x b*) = *mon-s fg x* ∪ *mon-c fg b*
by (*unfold mon-c-def*) *auto*

lemmas *mon-c-unconc* = *mon-c-union-conc mon-c-add-mset-unconc*

lemma *mon-cI*: ⟦*s* ∈# *c*; *m*∈*mon-s fg s*⟧ $\Longrightarrow$ *m*∈*mon-c fg c*
by (*unfold mon-c-def*, *auto*)
lemma *mon-cD*: ⟦*m*∈*mon-c fg c*⟧ $\Longrightarrow$ ∃ *s*. *s* ∈# *c* ∧ *m*∈*mon-s fg s*
by (*unfold mon-c-def*, *auto*)

lemma *mon-s-mono*: *set s* ⊆ *set s′* $\Longrightarrow$ *mon-s fg s* ⊆ *mon-s fg s′*
by (*unfold mon-s-def*) *auto*
lemma *mon-c-mono*: *c*⊆#*c′* $\Longrightarrow$ *mon-c fg c* ⊆ *mon-c fg c′*
by (*unfold mon-c-def*) (*auto dest*: *mset-subset-eqD*)

lemma *mon-w-empty*[*simp*]: *mon-w fg* [] = {}
by (*unfold mon-w-def*, *auto*)

lemma *mon-w-single*[*simp*]: *mon-w fg* [*e*] = *mon-e fg e*
by (*unfold mon-w-def*, *auto*)
lemma *mon-w-unconc*: *mon-w fg* (*wa*@*wb*) = *mon-w fg wa* ∪ *mon-w fg wb*
by (*unfold mon-w-def*) *auto*
lemma *mon-w-uncons*[*simp*]: *mon-w fg* (*e*#*w*) = *mon-e fg e* ∪ *mon-w fg w*
by (*rule mon-w-unconc*[**where** *wa*=[*e*], *simplified*])
lemma *mon-w-ileq*: $w \preceq w' \Longrightarrow$ *mon-w fg w* ⊆ *mon-w fg w'*
by (*induct rule*: *less-eq-list-induct*) *auto*

7.3 Valid configurations

We call a configuration *valid* if each monitor is owned by at most one thread.

definition
valid fg c == ∀ *s s'*. {#*s*, *s'*#} ⊆# *c* ⟶ *mon-s fg s* ∩ *mon-s fg s'* = {}

lemma *valid-empty*[*simp*, *intro*!]: *valid fg* {#}
by (*unfold valid-def*, *auto*)

lemma *valid-single*[*simp*, *intro*!]: *valid fg* {#*s*#}
by (*unfold valid-def subset-mset-def*) *auto*

lemma *valid-split1*:
valid fg (*c*+*c'*) ⟹ *valid fg c* ∧ *valid fg c'* ∧ *mon-c fg c* ∩ *mon-c fg c'* = {}
apply (*unfold valid-def*)
apply (*auto simp add*: *mset-le-incr-right*)
apply (*drule mon-cD*)+
apply *auto*
apply (*subgoal-tac* {#*s*#}+{#*sa*#} ⊆# *c*+*c'*)
apply (*auto dest*!: *multi-member-split*)
done

lemma *valid-split2*:
⟦*valid fg c*; *valid fg c'*; *mon-c fg c* ∩ *mon-c fg c'* = {}⟧ ⟹ *valid fg* (*c*+*c'*)
apply (*unfold valid-def*)
apply (*intro impI allI*)
apply (*erule mset-2dist2-cases*)
apply *simp-all*
apply (*blast intro*: *mon-cI*)+
done

lemma *valid-union-conc*:
valid fg (*c*+*c'*) ⟷ (*valid fg c* ∧ *valid fg c'* ∧ *mon-c fg c* ∩ *mon-c fg c'* = {})
by (*blast dest*: *valid-split1 valid-split2*)

lemma *valid-add-mset-conc*:
valid fg (*add-mset x c'*) ⟷ (*valid fg c'* ∧ *mon-s fg x* ∩ *mon-c fg c'* = {})
unfolding *add-mset-add-single*[*of x c'*] *valid-union-conc* **by** (*auto simp*: *mon-s-def*)

lemmas *valid-unconc* = *valid-union-conc valid-add-mset-conc*

lemma *valid-no-mon*: *mon-c fg c* = {} $\Longrightarrow$ *valid fg c*
proof (*unfold valid-def*, *intro allI impI*)
fix *s s'*
assume *A*: *mon-c fg c* = {} **and** *B*: {#*s*, *s'*#} ⊆# *c*
from *mon-c-mono*[*OF B*, *of fg*] *A* **have** *mon-s fg s* = {} *mon-s fg s'* = {} **by** (*auto simp add*: *mon-c-unconc*)
thus *mon-s fg s* ∩ *mon-s fg s'* = {} **by** *blast*
qed

7.4 Configurations at control points

— A stack is *at U* if its top node is from the set *U*
primrec *atU-s* :: *'n set* ⇒ *'n list* ⇒ *bool* **where**
atU-s U [] = *False*
| *atU-s U* (*u*#*r*) = (*u*∈*U*)

lemma *atU-s-decomp*[*simp*]: *atU-s U* (*s*@*s'*) = (*atU-s U s* ∨ (*s*=[] ∧ *atU-s U s'*))
by (*induct s*) *auto*

— A configuration is *at U* if it contains a stack that is at *U*
definition
atU U c == ∃ *s*. *s* ∈# *c* ∧ *atU-s U s*

lemma *atU-fmt*: ⟦*atU U c*; !!*ui r*. ⟦*ui*#*r* ∈# *c*; *ui*∈*U*⟧ $\Longrightarrow$ *P*⟧ $\Longrightarrow$ *P*
apply (*unfold atU-def*)
apply *auto*
apply (*case-tac s*)
apply *auto*
done

lemma *atU-union-cases*[*case-names left right*, *consumes 1*]: ⟦
atU U (*c1*+*c2*);
atU U c1 $\Longrightarrow$ *P*;
atU U c2 $\Longrightarrow$ *P*
⟧ $\Longrightarrow$ *P*
by (*unfold atU-def*) (*blast elim*: *mset-un-cases*)

lemma *atU-add*: *atU U c* $\Longrightarrow$ *atU U* (*c*+*ce*) ∧ *atU U* (*ce*+*c*)
by (*unfold atU-def*) (*auto simp add*: *union-ac*)

lemma *atU-union*[*simp*]: *atU U* (*c1*+*c2*) = (*atU U c1* ∨ *atU U c2*)
by (*auto simp add*: *atU-add elim*: *atU-union-cases*)

lemma *atU-empty*[*simp*]: ¬*atU U* {#}
by (*unfold atU-def*, *auto*)
lemma *atU-single*[*simp*]: *atU U* {#*s*#} = *atU-s U s*
by (*unfold atU-def*, *auto*)
lemma *atU-single-top*[*simp*]: *atU U* {#*u*#*r*#} = (*u*∈*U*)

by (*auto*)

lemma *atU-add-mset*[*simp*]: *atU U* (*add-mset c c2*) = (*atU-s U c* ∨ *atU U c2*)
unfolding *add-mset-add-single*[*of c c2*] *atU-union* **by** *auto*

lemma *atU-xchange-stack*: *atU U* (*add-mset* (*u*#*r*) *c*) ⟹ *atU U* (*add-mset* (*u*#*r*′) *c*)
by (*simp*)

— A configuration is *simultaneously at U* and *V* if it contains a stack at *U* and another one at *V*
definition
atUV U V c == ∃ *su sv*. {#*su*#}+{#*sv*#} ⊆# *c* ∧ *atU-s U su* ∧ *atU-s V sv*

lemma *atUV-empty*[*simp*]: ¬*atUV U V* {#}
by (*unfold atUV-def*) *auto*
lemma *atUV-single*[*simp*]: ¬*atUV U V* {#*s*#}
by (*unfold atUV-def*) *auto*

lemma *atUV-union*[*simp*]:
atUV U V (*c1*+*c2*) ⟷
(
(*atUV U V c1*) ∨
(*atUV U V c2*) ∨
(*atU U c1* ∧ *atU V c2*) ∨
(*atU V c1* ∧ *atU U c2*)
)
apply (*unfold atUV-def atU-def*)
apply (*auto elim*!: *mset-2dist2-cases intro*: *mset-le-incr-right iff add*: *mset-le-mono-add-single*)
apply (*subst union-commute*)
apply (*auto iff add*: *mset-le-mono-add-single*)
done

lemma *atUV-add-mset*[*simp*]:
atUV U V (*add-mset c c2*) ⟷
(
(*atUV U V c2*) ∨
(*atU U* {#*c*#} ∧ *atU V c2*) ∨
(*atU V* {#*c*#} ∧ *atU U c2*)
)
unfolding *add-mset-add-single*[*of c c2*]
unfolding *atUV-union*
by *auto*

lemma *atUV-union-cases*[*case-names left right lr rl*, *consumes 1*]: ⟦
atUV U V (*c1*+*c2*);
atUV U V c1 ⟹ *P*;
atUV U V c2 ⟹ *P*;
⟦*atU U c1*; *atU V c2*⟧ ⟹ *P*;

$\llbracket atU\ V\ c1;\ atU\ U\ c2 \rrbracket \Longrightarrow P$
$\rrbracket \Longrightarrow P$
by *auto*

7.5 Operational semantics

7.5.1 Semantic reference point

We now define our semantic reference point. We assess correctness and completeness of analyses relative to this reference point.

inductive-set
refpoint :: ('n,'p,'ba,'m,'more) *flowgraph-rec-scheme* ⇒
('n *conf* × ('p,'ba) *label* × 'n *conf*) *set*
for *fg*
where
— A base edge transforms the top node of one stack and leaves the other stacks untouched.
refpoint-base: ⟦ (*u*,*Base a*,*v*)∈*edges fg*; *valid fg* ({#*u*#*r*#}+*c*) ⟧
⟹ (*add-mset* (*u*#*r*) *c*,*LBase a*,*add-mset* (*v*#*r*) *c*)∈*refpoint fg* |
— A call edge transforms the top node of a stack and then pushes the entry node of the called procedure onto that stack. It can only be executed if all monitors the called procedure synchronizes on are available. Reentrant monitors are modeled here by checking availability of monitors just against the other stacks, not against the stack of the thread that executes the call. The other stacks are left untouched.
refpoint-call: ⟦ (*u*,*Call p*,*v*)∈*edges fg*; *valid fg* ({#*u*#*r*#}+*c*);
mon fg p ∩ *mon-c fg c* = {} ⟧
⟹ (*add-mset* (*u*#*r*) *c*,*LCall p*, *add-mset* (*entry fg p*#*v*#*r*) *c*)∈*refpoint fg* |
— A return step pops a return node from a stack. There is no corresponding flowgraph edge for a return step. The other stacks are left untouched.
refpoint-ret: ⟦ *valid fg* ({#*return fg p*#*r*#}+*c*) ⟧
⟹ (*add-mset* (*return fg p*#*r*) *c*,*LRet*,(*add-mset r c*))∈*refpoint fg* |
— A spawn edge transforms the top node of a stack and adds a new stack to the environment, with the entry node of the spawned procedure at the top and no stored return addresses. The other stacks are also left untouched.
refpoint-spawn: ⟦ (*u*,*Spawn p*,*v*)∈*edges fg*; *valid fg* (*add-mset* (*u*#*r*) *c*) ⟧
⟹ (*add-mset* (*u*#*r*) *c*,*LSpawn p*, *add-mset* (*v*#*r*) (*add-mset* [*entry fg p*] *c*))∈*refpoint fg*

Instead of working directly with the reference point semantics, we define the operational semantics of flowgraphs by describing how a single stack is transformed in a context of environment threads, and then use the theory developed in Section 5 to derive an interleaving semantics. Note that this semantics is also defined for invalid configurations (cf. Section 7.3). In Section 7.6.1 we will show that it preserves validity of a configuration, and in Section 7.6.2 we show that it is equivalent to the reference point semantics on valid configurations.

inductive-set
trss :: ('n,'p,'ba,'m,'more) *flowgraph-rec-scheme* ⇒

$(('n\ list * 'n\ conf) * ('p,'ba)\ label * ('n\ list * 'n\ conf))\ set$
for *fg*
where
trss-base: $[\![(u,Base\ a,v) \in edges\ fg]\!] \Longrightarrow$
$((u\#r,c),\ LBase\ a,\ (v\#r,c)\ ) \in trss\ fg$
| *trss-call*: $[\![(u,Call\ p,v) \in edges\ fg;\ mon\ fg\ p \cap mon\text{-}c\ fg\ c = \{\}\ ]\!] \Longrightarrow$
$((u\#r,c),LCall\ p,\ ((entry\ fg\ p)\#v\#r,c)) \in trss\ fg$
| *trss-ret*: $((((return\ fg\ p)\#r),c),LRet,(r,c)) \in trss\ fg$
| *trss-spawn*: $[\![\ (u,Spawn\ p,v) \in edges\ fg\ ]\!] \Longrightarrow$
$((u\#r,c),LSpawn\ p,(v\#r,add\text{-}mset\ [entry\ fg\ p]\ c)) \in trss\ fg$

— The interleaving semantics is generated using the general techniques from Section 5
abbreviation *tr* **where** *tr fg* == *gtr* (*trss fg*)
— We also generate the loc/env-semantics
abbreviation *trp* **where** *trp fg* == *gtrp* (*trss fg*)

7.6 Basic properties

7.6.1 Validity

lemma (**in** *flowgraph*) *trss-valid-preserve-s*:
$[\![valid\ fg\ (add\text{-}mset\ s\ c);\ ((s,c),e,(s',c')) \in trss\ fg]\!] \Longrightarrow valid\ fg\ (add\text{-}mset\ s'\ c')$
apply (*erule trss.cases*)
apply (*simp-all add*: *valid-unconc mon-c-unconc*)
by (*blast dest*: *mon-n-same-proc edges-part*)+

lemma (**in** *flowgraph*) *trss-valid-preserve*:
$[\![((s,c),w,(s',c')) \in trcl\ (trss\ fg);\ valid\ fg\ (\{\#s\#\}+c)]\!] \Longrightarrow valid\ fg\ (\{\#s'\#\}+c')$
by (*induct rule*: *trcl-pair-induct*) (*auto intro*: *trss-valid-preserve-s*)

lemma (**in** *flowgraph*) *tr-valid-preserve-s*:
$[\![(c,e,c') \in tr\ fg;\ valid\ fg\ c]\!] \Longrightarrow valid\ fg\ c'$
by (*rule gtr-preserve-s*[**where** *P*=*valid fg*]) (*auto dest*: *trss-valid-preserve-s*)

lemma (**in** *flowgraph*) *tr-valid-preserve*:
$[\![(c,w,c') \in trcl\ (tr\ fg);\ valid\ fg\ c]\!] \Longrightarrow valid\ fg\ c'$
by (*rule gtr-preserve*[**where** *P*=*valid fg*]) (*auto dest*: *trss-valid-preserve-s*)

lemma (**in** *flowgraph*) *trp-valid-preserve-s*:
$[\![((s,c),e,(s',c')) \in trp\ fg;\ valid\ fg\ (add\text{-}mset\ s\ c)]\!] \Longrightarrow valid\ fg\ (add\text{-}mset\ s'\ c')$
by (*rule gtrp-preserve-s*[**where** *P*=*valid fg*]) (*auto dest*: *trss-valid-preserve-s*)

lemma (**in** *flowgraph*) *trp-valid-preserve*:
$[\![((s,c),w,(s',c')) \in trcl\ (trp\ fg);\ valid\ fg\ (\{\#s\#\}+c)]\!] \Longrightarrow valid\ fg\ (add\text{-}mset\ s'\ c')$
by (*rule gtrp-preserve*[**where** *P*=*valid fg*]) (*auto dest*: *trss-valid-preserve-s*)

7.6.2 Equivalence to reference point

— The equivalence between the semantics that we derived using the techniques from Section 5 and the semantic reference point is shown nearly automatically.

lemma *refpoint-eq-s*: *valid fg c* $\Longrightarrow$ $((c,e,c')\in$*refpoint fg*$)$ $\longleftrightarrow$ $((c,e,c')\in$*tr fg*$)$
  apply *rule*
  apply (*erule refpoint.cases*)
    apply (*auto intro*: *gtrI-s trss.intros simp add*: *union-assoc add-mset-commute*)
  apply (*erule gtrE*)
  apply (*erule trss.cases*)
  apply (*auto intro*: *refpoint.intros simp add*: *union-assoc*[*symmetric*] *add-mset-commute*)
  done

lemma (**in** *flowgraph*) *refpoint-eq*:
  valid fg c $\Longrightarrow$ $((c,w,c')\in$*trcl* (*refpoint fg*)$)$ $\longleftrightarrow$ $((c,w,c')\in$*trcl* (*tr fg*)$)$
proof –
  have $((c,w,c')\in$*trcl* (*refpoint fg*)$)$ $\Longrightarrow$ *valid fg c* $\Longrightarrow$ $((c,w,c')\in$*trcl* (*tr fg*)$)$ **by** (*induct rule*: *trcl.induct*) (*auto simp add*: *refpoint-eq-s tr-valid-preserve-s*)
  moreover have $((c,w,c')\in$*trcl* (*tr fg*)$)$ $\Longrightarrow$ *valid fg c* $\Longrightarrow$ $((c,w,c')\in$*trcl* (*refpoint fg*)$)$ **by** (*induct rule*: *trcl.induct*) (*auto simp add*: *refpoint-eq-s tr-valid-preserve-s*)
  ultimately show *valid fg c* $\Longrightarrow$ $((c,w,c')\in$*trcl* (*refpoint fg*)$)$ = $((c,w,c')\in$*trcl* (*tr fg*)$)$ **..**
qed

7.6.3 Case distinctions

lemma *trss-c-cases-s*[*cases set*, *case-names no-spawn spawn*]: $[\![$
    $((s,c),e,(s',c'))\in$*trss fg*;
    $[\![$ $c'=c$ $]\!]$ $\Longrightarrow$ P;
    !!*p u v*. $[\![$ *e*=*LSpawn p*; (*u*,*Spawn p*,*v*)∈*edges fg*;
        hd s=*u*; *hd s'*=*v*; c'={#[*entry fg p*]#}+*c* $]\!]$ $\Longrightarrow$ P
  $]\!]$ $\Longrightarrow$ P
by (*auto elim*!: *trss.cases*)
lemma *trss-c-fmt-s*: $[\![((s,c),e,(s',c'))\in$*trss fg*$]\!]$
  $\Longrightarrow$ ∃ *csp*. c'=*csp*+*c* ∧
    (*csp*={#} ∨ (∃ *p*. *e*=*LSpawn p* ∧ *csp*={#[*entry fg p*]#}))
  by (*force elim*!: *trss-c-cases-s*)

lemma (**in** *flowgraph*) *trss-c'-split-s*: $[\![$
    $((s,c),e,(s',c'))\in$*trss fg*;
    !!*csp*. $[\![c'$=*csp*+*c*; *mon-c fg csp*={}$]\!]$ $\Longrightarrow$ P
  $]\!]$ $\Longrightarrow$ P
  apply (*erule trss-c-cases-s*)
  apply (*subgoal-tac* c'={#}+*c*)
  apply (*fastforce*)
  apply *auto*
  done

lemma *trss-c-cases*[*cases set*, *case-names c-case*]: !!*s c*. $[\![$
    $((s,c),w,(s',c'))\in$*trcl* (*trss fg*);

!!*csp*. ⟦ c'=*csp*+*c*; !!*s*. *s* ∈# *csp* ⟹ ∃ *p u v*. *s*=[*entry fg p*] ∧
(*u*,*Spawn p*,*v*)∈*edges fg* ∧
initialproc fg p⟧
⟹ *P*
⟧ ⟹ *P*
proof (*induct w*)
case *Nil* **note** *A*=*this*
hence s'=*s* c'=*c* **by** *simp-all*
hence c'={#}+*c* **by** *simp*
from *A*(*2*)[*OF this*] **show** *P* **by** *simp*
next
case (*Cons e w*) **note** *IHP*=*this*
then obtain *sh ch* **where** *SPLIT1*: ((*s*,*c*),*e*,(*sh*,*ch*))∈*trss fg* **and** *SPLIT2*: ((*sh*,*ch*),*w*,(s',c'))∈*trcl* (*trss fg*) **by** (*fastforce dest*: *trcl-uncons*)
from *SPLIT2* **show** *?case* **proof** (*rule IHP*(*1*))
fix *csp*
assume *C'FMT*: c'=*csp*+*ch* **and** *CSPFMT*: !!*s*. *s* ∈# *csp* ⟹ ∃ *p u v*. *s*=[*entry fg p*] ∧ (*u*,*Spawn p*, *v*)∈*edges fg* ∧ *initialproc fg p*
from *SPLIT1* **show** *?thesis*
proof (*rule trss-c-cases-s*)
assume *ch*=*c* **with** *C'FMT CSPFMT IHP*(*3*) **show** *?case* **by** *blast*
next
fix *p*
assume *EFMT*: *e*=*LSpawn p* **and** *CHFMT*: *ch*={#[*entry fg p*]#}+*c*
with *C'FMT* **have** c'=({#[*entry fg p*]#}+*csp*)+*c* **by** (*simp add*: *union-ac*)
moreover
from *EFMT SPLIT1* **have** ∃ *u v*. (*u*,*Spawn p*, *v*)∈*edges fg* **by** (*blast elim*!: *trss.cases*)
hence !!*s*. *s* ∈# {#[*entry fg p*]#} + *csp* ⟹ ∃ *p u v*. *s*=[*entry fg p*] ∧ (*u*,*Spawn p*, *v*)∈*edges fg* ∧ *initialproc fg p* **using** *CSPFMT* **by** (*unfold initialproc-def*, *erule-tac mset-un-cases*) (*auto*)
ultimately show *?case* **using** *IHP*(*3*) **by** *blast*
qed
qed
qed

lemma (**in** *flowgraph*) *c-of-initial-no-mon*:
assumes *A*: !!*s*. *s* ∈# *csp* ⟹ ∃ *p*. *s*=[*entry fg p*] ∧ *initialproc fg p*
shows *mon-c fg csp* = {}
by (*unfold mon-c-def*) (*auto dest*: *A initial-no-mon*)

lemma (**in** *flowgraph*) *trss-c-no-mon-s*:
assumes *A*: ((*s*,*c*),*e*,(s',c'))∈*trss fg*
shows *mon-c fg* c' = *mon-c fg c*
using *A*
proof (*erule-tac trss-c-cases-s*)
assume c'=*c* **thus** *?thesis* **by** *simp*
next

fix *p* **assume** *EFMT*: *e=LSpawn p* **and** *C′FMT*: *c′={#[entry fg p]#} + c*
from *EFMT* **obtain** *u v* **where** (*u,Spawn p,v*)∈*edges fg* **using** *A* **by** (*auto elim*: *trss.cases*)
with *spawn-no-mon* **have** *mon-c fg* {#[*entry fg p*]#} = {} **by** *simp*
with *C′FMT* **show** *?thesis* **by** (*simp add*: *mon-c-unconc*)
qed

corollary (**in** *flowgraph*) *trss-c-no-mon*:
((*s*,*c*),*w*,(*s′*,*c′*))∈*trcl* (*trss fg*) ⟹ *mon-c fg c′* = *mon-c fg c*
apply (*auto elim*!: *trss-c-cases simp add*: *mon-c-unconc*)
proof −
fix *csp x*
assume *x*∈*mon-c fg csp*
then obtain *s* **where** *s* ∈# *csp* **and** *M*: *x*∈*mon-s fg s* **by** (*unfold mon-c-def*, *auto*)
moreover assume ∀ *s*. *s* ∈# *csp* ⟶ (∃ *p*. *s* = [*entry fg p*] ∧ (∃ *u v*. (*u*, *Spawn p*, *v*) ∈ *edges fg*) ∧ *initialproc fg p*)
ultimately obtain *p u v* **where** *s*=[*entry fg p*] **and** (*u*,*Spawn p*,*v*)∈*edges fg* **by** *blast*
hence *mon-s fg s* = {} **by** (*simp*)
with *M* **have** *False* **by** *simp*
thus *x*∈*mon-c fg c* **..**
qed

lemma (**in** *flowgraph*) *trss-spawn-no-mon-step*[*simp*]:
((*s*,*c*),*LSpawn p*, (*s′*,*c′*))∈*trss fg* ⟹ *mon fg p* = {}
by (*auto elim*: *trss.cases*)

lemma *trss-no-empty-s*[*simp*]: (([],*c*),*e*,(*s′*,*c′*))∈*trss fg* = *False*
by (*auto elim*!: *trss.cases*)
lemma *trss-no-empty*[*simp*]:
assumes *A*: (([],*c*),*w*,(*s′*,*c′*))∈*trcl* (*trss fg*)
shows *w*=[] ∧ *s′*=[] ∧ *c*=*c′*
proof −
note *A*
moreover {
fix *s*
have ((*s*,*c*),*w*,(*s′*,*c′*))∈*trcl* (*trss fg*) ⟹ *s*=[] ⟹ *w*=[] ∧ *s′*=[] ∧ *c*=*c′*
by (*induct rule*: *trcl-pair-induct*) *auto*
} **ultimately show** *?thesis* **by** *blast*
qed

lemma *trs-step-cases*[*cases set*, *case-names NO-SPAWN SPAWN*]:
assumes *A*: (*c*,*e*,*c′*)∈*tr fg*
assumes *A-NO-SPAWN*: !!*s ce s′ csp*. ⟦

$((s,ce),e,(s',ce))\in trss\ fg$;
$c=\{\#s\#\}+ce$; $c'=\{\#s'\#\}+ce$
$]\!] \Longrightarrow P$
assumes *A-SPAWN*: !!$s\ ce\ s'\ p$. $[\![$
$((s,ce),LSpawn\ p,(s',\{\#[entry\ fg\ p]\#\}+ce))\in trss\ fg$;
$c=\{\#s\#\}+ce$;
$c'=\{\#s'\#\}+\{\#[entry\ fg\ p]\#\}+ce$;
$e=LSpawn\ p$
$]\!] \Longrightarrow P$
shows P
proof −
from *A* **show** *?thesis* **proof** (*erule-tac gtr-find-thread*)
fix $s\ ce\ s'\ ce'$
assume *FMT*: $c = add\text{-}mset\ s\ ce\ c' = add\text{-}mset\ s'\ ce'$
assume *B*: $((s,\ ce),\ e,\ s',\ ce') \in trss\ fg$ **thus** *?thesis* **proof** (*cases rule*: *trss-c-cases-s*)
case *no-spawn* **thus** *?thesis* **using** *FMT B* **by** (−) (*rule A-NO-SPAWN*, *auto*)

next
case (*spawn p*) **thus** *?thesis* **using** *FMT B* **by** (−) (*rule A-SPAWN*, *auto simp add*: *union-assoc*)
qed
qed
qed

7.7 Advanced properties

7.7.1 Stack composition / decomposition

lemma *trss-stack-comp-s*:
$((s,c),e,(s',c'))\in trss\ fg \Longrightarrow ((s@r,c),e,(s'@r,c'))\in trss\ fg$
by (*auto elim*!: *trss.cases intro*: *trss.intros*)

lemma *trss-stack-comp*:
$((s,c),w,(s',c'))\in trcl\ (trss\ fg) \Longrightarrow ((s@r,c),w,(s'@r,c'))\in trcl\ (trss\ fg)$
proof (*induct rule*: *trcl-pair-induct*)
case *empty* **thus** *?case* **by** *auto*
next
case (*cons s c e sh ch w s' c'*) **note** *IHP=this*
from *trss-stack-comp-s*[*OF IHP*(*1*)] **have** $((s\ @\ r,\ c),\ e,\ sh\ @\ r,\ ch) \in trss\ fg$ **.**
also note *IHP*(*3*)
finally show *?case* **.**
qed

lemma *trss-stack-decomp-s*: $[\![\ ((s@r,c),e,(s',c'))\in trss\ fg;\ s\neq[]\ ]\!]$
$\Longrightarrow \exists\, sp'.\ s'=sp'@r \wedge ((s,c),e,(sp',c'))\in trss\ fg$
by (*cases s*, *simp*) (*auto intro*: *trss.intros elim*!: *trss.cases*)

lemma *trss-find-return*: $[\![$
$((s@r,c),w,(r,c'))\in trcl\ (trss\ fg)$;

!!*wa wb ch*. ⟦ *w*=*wa*@*wb*; ((*s*,*c*),*wa*,([],*ch*))∈*trcl* (*trss fg*);
((*r*,*ch*),*wb*,(*r*,*c'*))∈*trcl* (*trss fg*) ⟧ ⟹ *P*
⟧ ⟹ *P*
— If $s = []$, the proposition follows trivially
apply (*cases s*=[])
apply *fastforce*
proof –
— For $s \neq []$, we use induction by w
have *IM*: !!*s c*. ⟦ ((*s*@*r*,*c*),*w*,(*r*,*c'*))∈*trcl* (*trss fg*); *s*≠[] ⟧ ⟹ ∃ *wa wb ch*. *w*=*wa*@*wb* ∧ ((*s*,*c*),*wa*,([],*ch*))∈*trcl* (*trss fg*) ∧ ((*r*,*ch*),*wb*,(*r*,*c'*))∈*trcl* (*trss fg*)
proof (*induct w*)
case *Nil* **thus** *?case* **by** (*auto*)
next
case (*Cons e w*) **note** *IHP*=*this*
then obtain *sh ch* **where** *SPLIT1*: ((*s*@*r*,*c*),*e*,(*sh*,*ch*))∈*trss fg* **and** *SPLIT2*: ((*sh*,*ch*),*w*,(*r*,*c'*))∈*trcl* (*trss fg*) **by** (*fast dest*: *trcl-uncons*)
{ **assume** *CASE*: *e*=*LRet*
with *SPLIT1* **obtain** *p* **where** *EDGE*: *s*@*r*=*return fg p* # *sh c*=*ch* **by** (*auto elim*!: *trss.cases*)
with *IHP*(*3*) **obtain** *ss* **where** *SHFMT*: *s*=*return fg p* # *ss sh*=*ss*@*r* **by** (*cases s*, *auto*)
{ **assume** *CC*: *ss*≠[]
with *SHFMT* **have** ∃ *ss*. *ss*≠[] ∧ *sh*=*ss*@*r* **by** *blast*
} **moreover** {
assume *CC*: *ss*=[]
with *CASE SHFMT EDGE* **have** ((*s*,*c*),[*e*],([],*ch*))∈*trcl* (*trss fg*) *e*#*w*=[*e*]@*w* **by** (*auto intro*: *trss-ret*)
moreover from *SPLIT2 SHFMT CC* **have** ((*r*,*ch*),*w*,(*r*,*c'*))∈*trcl* (*trss fg*) **by** *simp*
ultimately have *?case* **by** *blast*
} **ultimately have** *?case* ∨ (∃ *ss*. *ss*≠[] ∧ *sh*=*ss*@*r*) **by** *blast*
} **moreover** {
assume *e*≠*LRet*
with *SPLIT1 IHP*(*3*) **have** (∃ *ss*. *ss*≠[] ∧ *sh*=*ss*@*r*) **by** (*force elim*!: *trss.cases simp add*: *append-eq-Cons-conv*)
} **moreover** {
assume (∃ *ss*. *ss*≠[] ∧ *sh*=*ss*@*r*)
then obtain *ss* **where** *CASE*: *ss*≠[] *sh*=*ss*@*r* **by** *blast*
with *SPLIT2* **have** ((*ss*@*r*, *ch*), *w*, *r*, *c'*) ∈ *trcl* (*trss fg*) **by** *simp*
from *IHP*(*1*)[*OF this CASE*(*1*)] **obtain** *wa wb ch'* **where** *IHAPP*: *w*=*wa*@*wb* ((*ss*,*ch*),*wa*,([],*ch'*))∈*trcl* (*trss fg*) ((*r*,*ch'*),*wb*,(*r*,*c'*))∈*trcl* (*trss fg*) **by** *blast*
moreover from *CASE SPLIT1* **have** ((*s* @ *r*, *c*), *e*, *ss*@*r*, *ch*) ∈ *trss fg* **by** *simp*
from *trss-stack-decomp-s*[*OF this IHP*(*3*)] **have** ((*s*, *c*), *e*, *ss*, *ch*) ∈ *trss fg* **by** *auto*
with *IHAPP* **have** ((*s*, *c*), *e*#*wa*, ([],*ch'*)) ∈ *trcl* (*trss fg*) **by** (*rule-tac trcl.cons*)
moreover from *IHAPP* **have** *e*#*w*=(*e*#*wa*)@*wb* **by** *auto*
ultimately have *?case* **by** *blast*
} **ultimately show** *?case* **by** *blast*

qed
assume $((s\ @\ r,\ c),\ w,\ r,\ c') \in$ *trcl* (*trss fg*) $s \neq []$!!*wa wb ch*. ⟦ *w*=*wa*@*wb*; ((*s*,*c*),*wa*,([],*ch*))∈*trcl* (*trss fg*); ((*r*,*ch*),*wb*,(*r*,*c'*))∈*trcl* (*trss fg*) ⟧ ⟹ *P* **thus** *P* **by** (*blast dest*: *IM*)
qed

lemma *trss-return-cases*[*cases set*]: !!*u r c*. ⟦
((*u*#*r*,*c*),*w*,(*r'*,*c'*))∈*trcl* (*trss fg*);
!! *s' u'*. ⟦ *r'*=*s'*@*u'*#*r*; (([*u*],*c*),*w*,(*s'*@[*u'*],*c'*))∈*trcl* (*trss fg*) ⟧ ⟹ *P*;
!! *wa wb ch*. ⟦ *w*=*wa*@*wb*; (([*u*],*c*),*wa*,([],*ch*))∈*trcl* (*trss fg*);
((*r*,*ch*),*wb*,(*r'*,*c'*))∈*trcl* (*trss fg*) ⟧ ⟹ *P*
⟧ ⟹ *P*
proof (*induct w rule*: *length-compl-induct*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons e w*) **note** *IHP*=*this*
then obtain *sh ch* **where** *SPLIT1*: ((*u*#*r*,*c*),*e*,(*sh*,*ch*))∈*trss fg* **and** *SPLIT2*: ((*sh*,*ch*),*w*,(*r'*,*c'*))∈*trcl* (*trss fg*) **by** (*fast dest*: *trcl-uncons*)
{
fix *ba q*
assume *CASE*: *e*=*LBase ba* ∨ *e*=*LSpawn q*
with *SPLIT1* **obtain** *v* **where** *E*: *sh*=*v*#*r* (([*u*],*c*),*e*,([*v*],*ch*))∈*trss fg* **by** (*auto elim*!: *trss.cases intro*: *trss.intros*)
with *SPLIT2* **have** ((*v*#*r*,*ch*),*w*,(*r'*,*c'*))∈*trcl* (*trss fg*) **by** *simp*
hence *?case* **proof** (*cases rule*: *IHP*(*1*)[*of w*, *simplified*, *cases set*])
case (*1 s' u'*) **note** *CC*=*this*
with *E*(*2*) **have** (([*u*],*c*),*e*#*w*,(*s'*@[*u'*],*c'*))∈*trcl* (*trss fg*) **by** *simp*
from *IHP*(*3*)[*OF CC*(*1*) *this*] **show** *?thesis* **.**
next
case (*2 wa wb ct*) **note** *CC*=*this*
with *E*(*2*) **have** (([*u*],*c*),*e*#*wa*,([],*ct*))∈*trcl* (*trss fg*) *e*#*w* = (*e*#*wa*)@*wb* **by** *simp-all*
from *IHP*(*4*)[*OF this*(*2*,*1*) *CC*(*3*)] **show** *?thesis* **.**
qed
} moreover {
assume *CASE*: *e*=*LRet*
with *SPLIT1* **have** *sh*=*r* (([*u*],*c*),[*e*],([],*ch*))∈*trcl* (*trss fg*) **by** (*auto elim*!: *trss.cases intro*: *trss.intros*)
with *IHP*(*4*)[*OF* - *this*(*2*)] *SPLIT2* **have** *?case* **by** *auto*
} moreover {
fix *q*
assume *CASE*: *e*=*LCall q*
with *SPLIT1* **obtain** *u'* **where** *SHFMT*: *sh*=*entry fg q* # *u'* # *r* (([*u*],*c*),*e*,(*entry fg q* # [*u'*],*ch*))∈*trss fg* **by** (*auto elim*!: *trss.cases intro*: *trss.intros*)
with *SPLIT2* **have** ((*entry fg q* # *u'* # *r*,*ch*),*w*,(*r'*,*c'*))∈*trcl* (*trss fg*) **by** *simp*
hence *?case* **proof** (*cases rule*: *IHP*(*1*)[*of w*, *simplified*, *cases set*])
case (*1 st ut*) **note** *CC*=*this*
from *trss-stack-comp*[*OF CC*(*2*), **where** *r*=[*u'*]] **have** ((*entry fg q*#[*u'*], *ch*),

w, (*st* @ [*ut*]) @ [*u'*], *c'*) ∈ *trcl* (*trss fg*) **by** *auto*
with *SHFMT*(*2*) **have** (([*u*],*c*),*e*#*w*, (*st* @ [*ut*]) @ [*u'*], *c'*) ∈ *trcl* (*trss fg*) **by** *auto*
from *IHP*(*3*)[*OF - this*] *CC*(*1*) **show** *?thesis* **by** *simp*
next
case (*2 wa wb ct*) **note** *CC*=*this*
from *trss-stack-comp*[*OF CC*(*2*), **where** *r*=[*u'*]] **have** ((*entry fg q* # [*u'*], *ch*), *wa*, [*u'*], *ct*) ∈ *trcl* (*trss fg*) **by** *simp*
with *SHFMT* **have** *PREPATH*: (([*u*],*c*),*e*#*wa*, [*u'*], *ct*) ∈ *trcl* (*trss fg*) **by** *simp*
from *CC* **have** *L*: *length wb*≤*length w* **by** *simp*
from *CC*(*3*) **show** *?case* **proof** (*cases rule*: *IHP*(*1*)[*OF L*, *cases set*])
case (*1 s'' u''*) **note** *CCC*=*this* **from** *trcl-concat*[*OF PREPATH CCC*(*2*)] *CC*(*1*) **have** (([*u*],*c*),*e*#*w*,(*s''*@[*u''*],*c'*))∈*trcl* (*trss fg*) **by** (*simp*)
from *IHP*(*3*)[*OF CCC*(*1*) *this*] **show** *?thesis* **.**
next
case (*2 wba wbb c''*) **note** *CCC*=*this* **from** *trcl-concat*[*OF PREPATH CCC*(*2*)] *CC*(*1*) *CCC*(*1*) **have** *e*#*w* = (*e*#*wa*@*wba*)@*wbb* (([*u*], *c*), *e* # *wa* @ *wba*, [], *c''*) ∈ *trcl* (*trss fg*) **by** *auto*
from *IHP*(*4*)[*OF this CCC*(*3*)] **show** *?thesis* **.**
qed
qed
} **ultimately show** *?case* **by** (*cases e*, *auto*)
qed

lemma (**in** *flowgraph*) *trss-find-call*:
!!*v r' c'*. ⟦ (([*sp*],*c*),*w*,(*v*#*r'*,*c'*)) ∈ *trcl* (*trss fg*); *r'*≠[] ⟧
⟹ ∃ *rh ch p wa wb*.
w=*wa*@(*LCall p*)#*wb* ∧
proc-of fg v = *p* ∧
(([*sp*],*c*),*wa*,(*rh*,*ch*))∈*trcl* (*trss fg*) ∧
((*rh*,*ch*),*LCall p*,((*entry fg p*)#*r'*,*ch*))∈*trss fg* ∧
(([*entry fg p*],*ch*),*wb*,([*v*],*c'*))∈*trcl* (*trss fg*)
proof (*induct w rule*: *length-compl-rev-induct*)
case *Nil* **thus** *?case* **by** (*auto*)
next
case (*snoc w e*) **note** *IHP*=*this*
then obtain *rh ch* **where** *SPLIT1*: (([*sp*],*c*),*w*,(*rh*,*ch*))∈*trcl* (*trss fg*) **and** *SPLIT2*: ((*rh*,*ch*),*e*,(*v*#*r'*,*c'*))∈*trss fg* **by** (*fast dest*: *trcl-rev-uncons*)

{
assume ∃ *u*. *rh*=*u*#*r'*
then obtain *u* **where** *RHFMT*[*simp*]: *rh*=*u*#*r'* **by** *blast*
with *SPLIT2* **have** *proc-of fg u* = *proc-of fg v* **by** (*auto elim*: *trss.cases intro*: *edges-part*)
moreover from *IHP*(*1*)[*of w u r' ch*, *OF - SPLIT1*[*simplified*] *IHP*(*3*)] **obtain** *rt ct p wa wb* **where**
IHAPP: *w* = *wa* @ *LCall p* # *wb proc-of fg u* = *p* (([*sp*], *c*), *wa*, (*rt*, *ct*)) ∈ *trcl* (*trss fg*) ((*rt*, *ct*), *LCall p*, *entry fg p* # *r'*, *ct*) ∈ *trss fg*

$(([entry\ fg\ p],\ ct),\ wb,\ ([u],\ ch)) \in trcl\ (trss\ fg)$ **by** (*blast*)
moreover
have $(([entry\ fg\ p],\ ct),\ wb@[e],\ ([v],\ c')) \in trcl\ (trss\ fg)$ **proof** −
note *IHAPP*(*5*)
also from *SPLIT2* **have** $(([u],ch),e,([v],c')) \in trss\ fg$ **by** (*auto elim*!: *trss.cases intro*!: *trss.intros*)
finally show *?thesis* **.**
qed
moreover from *IHAPP* **have** $w@[e] = wa\ @\ LCall\ p\ \#\ (wb@[e])$ **by** *auto*
ultimately have *?case* **by** *auto*
}
moreover have $(\exists\, u.\ rh{=}u\#r') \vee\ ?case$
proof (*rule trss.cases*[*OF SPLIT2*], *simp-all*, *goal-cases*) — Cases for base- and spawn edge are discharged automatically
— Case: call-edge
case (*1 ca p r u vv*) **with** *SPLIT1 SPLIT2* **show** *?case* **by** *fastforce*
next
— Case: return edge
case *CC*: (*2 q r ca*)
hence [*simp*]: $rh{=}(return\ fg\ q)\#v\#r'$ **by** *simp*
with *IHP*(*1*)[*of w* (*return fg q*) $v\#r'$ *ch*, *OF* - *SPLIT1*[*simplified*]] **obtain** *rt ct wa wb* **where**
IHAPP: $w = wa\ @\ LCall\ q\ \#\ wb$ $(([sp],\ c),\ wa,\ rt,\ ct) \in trcl\ (trss\ fg)$ $((rt,\ ct),\ LCall\ q,\ entry\ fg\ q\ \#\ v\ \#\ r',\ ct) \in trss\ fg$
$(([entry\ fg\ q],\ ct),\ wb,\ [return\ fg\ q],\ ch) \in trcl\ (trss\ fg)$ **by** *force*
then obtain *u* **where** *RTFMT* [*simp*]: $rt{=}u\#r'$ **and** *PROC-OF-U*: $proc\text{-}of\ fg\ u = proc\text{-}of\ fg\ v$ **by** (*auto elim*: *trss.cases intro*: *edges-part*)
from *IHAPP*(*1*) **have** *LENWA*: $length\ wa \le length\ w$ **by** *auto*
from *IHP*(*1*)[*OF LENWA IHAPP*(*2*)[*simplified*] *IHP*(*3*)] **obtain** *rhh chh p waa wab* **where**
IHAPP': $wa{=}waa@LCall\ p\ \#\ wab$ $proc\text{-}of\ fg\ u = p$ $(([sp],c),waa,(rhh,chh)){\in}trcl\ (trss\ fg)$ $((rhh,chh),LCall\ p,\ (entry\ fg\ p\#r',chh)){\in}trss\ fg$
$(([entry\ fg\ p],chh),wab,([u],ct)){\in}trcl\ (trss\ fg)$
by *blast*
from *IHAPP IHAPP' PROC-OF-U* **have** $w@[e]{=}waa@LCall\ p\#(wab@LCall\ q\#wb@[e]) \wedge proc\text{-}of\ fg\ v = p$ **by** *auto*
moreover have $(([entry\ fg\ p],chh),wab@(LCall\ q)\#wb@[e],([v],c')){\in}trcl\ (trss\ fg)$ **proof** −
note *IHAPP'*(*5*)
also from *IHAPP* **have** $(([u],\ ct),\ LCall\ q,\ entry\ fg\ q\ \#\ [v],\ ct) \in trss\ fg$ **by** (*auto elim*!: *trss.cases intro*!: *trss.intros*)
also from *trss-stack-comp*[*OF IHAPP*(*4*)] **have** $((entry\ fg\ q\#[v],ct),wb,(return\ fg\ q\#[v],ch)){\in}trcl\ (trss\ fg)$ **by** *simp*
also from *CC* **have** $((return\ fg\ q\#[v],ch),e,([v],c')){\in}trss\ fg$ **by** (*auto intro*: *trss-ret*)
finally show *?thesis* **by** *simp*
qed
moreover note *IHAPP' CC*
ultimately show *?case* **by** *auto*

qed
ultimately show *?case* **by** *blast*
qed

— This lemma is better suited for application in soundness proofs of constraint systems than *flowgraph.trss-find-call*
lemma (**in** *flowgraph*) *trss-find-call′*:
assumes *A*: $(([sp],c),w,(return\ fg\ p\#[u'],c')) \in trcl\ (trss\ fg)$
and *EX*: $!!uh\ ch\ wa\ wb.\ [\![$
$w{=}wa@(LCall\ p)\#wb;$
$(([sp],c),wa,([uh],ch)){\in}trcl\ (trss\ fg);$
$(([uh],ch),LCall\ p,((entry\ fg\ p)\#[u'],ch)){\in}trss\ fg;$
$(uh,Call\ p,u'){\in}edges\ fg;$
$(([entry\ fg\ p],ch),wb,([return\ fg\ p],c')){\in}trcl\ (trss\ fg)$
$]\!] \Longrightarrow P$
shows *P*
proof –
from *trss-find-call*[*OF A*] **obtain** *rh ch wa wb* **where** *FC*:
$w = wa\ @\ LCall\ p\ \#\ wb$
$(([sp],\ c),\ wa,\ rh,\ ch) \in trcl\ (trss\ fg)$
$((rh,\ ch),\ LCall\ p,\ [entry\ fg\ p,\ u'],\ ch) \in trss\ fg$
$(([entry\ fg\ p],\ ch),\ wb,\ [return\ fg\ p],\ c') \in trcl\ (trss\ fg)$
by *auto*
moreover from *FC(3)* **obtain** *uh* **where** *ADD*: $rh{=}[uh]$ $(uh,Call\ p,u'){\in}edges\ fg$ **by** (*auto elim*: *trss.cases*)
ultimately show *?thesis* **using** *EX* **by** *auto*
qed

lemma (**in** *flowgraph*) *trss-bot-proc-const*:
$!!s'\ u'\ c'.\ ((s@[u],c),w,(s'@[u'],c')){\in}trcl\ (trss\ fg)$
$\Longrightarrow proc\text{-}of\ fg\ u = proc\text{-}of\ fg\ u'$
proof (*induct w rule*: *rev-induct*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*snoc e w*) **note** *IHP=this* **then obtain** *sh ch* **where** *SPLIT1*: $((s@[u],c),w,(sh,ch)){\in}trcl\ (trss\ fg)$ **and** *SPLIT2*: $((sh,ch),e,(s'@[u'],c')){\in}trss\ fg$ **by** (*fast dest*: *trcl-rev-uncons*)
from *SPLIT2* **have** $sh{\neq}[]$ **by** (*auto elim!*: *trss.cases*)
then obtain *ssh uh* **where** *SHFMT*: $sh{=}ssh@[uh]$ **by** (*blast dest*: *list-rev-decomp*)
with *IHP(1)*[*of ssh uh ch*] *SPLIT1* **have** $proc\text{-}of\ fg\ u = proc\text{-}of\ fg\ uh$ **by** *auto*
also from *SPLIT2 SHFMT* **have** $proc\text{-}of\ fg\ uh = proc\text{-}of\ fg\ u'$ **by** (*cases rule*: *trss.cases*) (*cases ssh*, *auto simp add*: *edges-part*)+
finally show *?case* **.**
qed

— Specialized version of *flowgraph.trss-bot-proc-const*that comes in handy for precision proofs of constraint systems
lemma (**in** *flowgraph*) *trss-er-path-proc-const*:
$(([entry\ fg\ p],c),w,([return\ fg\ q],c')){\in}trcl\ (trss\ fg) \Longrightarrow p{=}q$
using *trss-bot-proc-const*[*of* [] *entry fg p* - - [] *return fg q*, *simplified*] **.**

lemma *trss-2empty-to-2return*: $\llbracket$ $((s,c),w,([],c'))\in trcl\ (trss\ fg)$; $s\neq[]$ $\rrbracket \Longrightarrow$
$\exists\, w'\ p.\ w=w'@[LRet] \wedge ((s,c),w',([return\ fg\ p],c'))\in trcl\ (trss\ fg)$
proof –
assume *A*: $((s,c),w,([],c'))\in trcl\ (trss\ fg)$ $s\neq[]$
hence $w\neq[]$ **by** *auto*
then obtain w' *e* **where** *WD*: $w=w'@[e]$ **by** (*blast dest*: *list-rev-decomp*)
with *A*(*1*) **obtain** *sh ch* **where** *SPLIT*: $((s,c),w',(sh,ch))\in trcl\ (trss\ fg)$ $((sh,ch),e,([],c'))\in trss\ fg$ **by** (*fast dest*: *trcl-rev-uncons*)
from *SPLIT*(*2*) **obtain** *p* **where** $e=LRet$ $sh=[return\ fg\ p]$ $ch=c'$ **by** (*cases rule*: *trss.cases*, *auto*)
with *SPLIT*(*1*) *WD* **show** *?thesis* **by** *blast*
qed

lemma *trss-2return-to-2empty*: $\llbracket$ $((s,c),w,([return\ fg\ p],c'))\in trcl\ (trss\ fg)$ $\rrbracket$
$\Longrightarrow ((s,c),w@[LRet],([],c'))\in trcl\ (trss\ fg)$
apply (*subgoal-tac* $(([return\ fg\ p],c'),LRet,([],c'))\in trss\ fg$)
by (*auto dest*: *trcl-rev-cons intro*: *trss.intros*)

7.7.2 Adding threads

lemma *trss-env-increasing-s*: $((s,c),e,(s',c'))\in trss\ fg \Longrightarrow c\subseteq\# c'$
by (*auto elim*!: *trss.cases*)
lemma *trss-env-increasing*: $((s,c),w,(s',c'))\in trcl\ (trss\ fg) \Longrightarrow c\subseteq\# c'$
by (*induct rule*: *trcl-pair-induct*) (*auto dest*: *trss-env-increasing-s order-trans*)

7.7.3 Conversion between environment and monitor restrictions

lemma *trss-mon-e-no-ctx*:
$((s,c),e,(s',c'))\in trss\ fg \Longrightarrow mon\text{-}e\ fg\ e \cap mon\text{-}c\ fg\ c = \{\}$
by (*erule trss.cases*) *auto*
lemma (**in** *flowgraph*) *trss-mon-w-no-ctx*:
$((s,c),w,(s',c'))\in trcl\ (trss\ fg) \Longrightarrow mon\text{-}w\ fg\ w \cap mon\text{-}c\ fg\ c = \{\}$
by (*induct rule*: *trcl-pair-induct*) (*auto dest*: *trss-mon-e-no-ctx simp add*: *trss-c-no-mon-s*)

lemma (**in** *flowgraph*) *trss-modify-context-s*:
$!!cn.\ \llbracket((s,c),e,(s',c'))\in trss\ fg;\ mon\text{-}e\ fg\ e \cap mon\text{-}c\ fg\ cn = \{\}\rrbracket$
$\Longrightarrow \exists\, csp.\ c'=csp+c \wedge mon\text{-}c\ fg\ csp = \{\} \wedge ((s,cn),e,(s',csp+cn))\in trss\ fg$
by (*erule trss.cases*) (*auto intro*!: *trss.intros*)

lemma (**in** *flowgraph*) *trss-modify-context*[*rule-format*]:
$\llbracket((s,c),w,(s',c'))\in trcl\ (trss\ fg)\rrbracket$
$\Longrightarrow \forall\, cn.\ mon\text{-}w\ fg\ w \cap mon\text{-}c\ fg\ cn = \{\}$
$\longrightarrow (\exists\, csp.\ c'=csp+c \wedge mon\text{-}c\ fg\ csp = \{\} \wedge$
$((s,cn),w,(s',csp+cn))\in trcl\ (trss\ fg))$
proof (*induct rule*: *trcl-pair-induct*)
case *empty* **thus** *?case* **by** *simp*
next
case (*cons s c e sh ch w s' c'*) **note** *IHP=this* **show** *?case*
proof (*intro allI impI*)

fix *cn*
assume *MON*: *mon-w fg* (*e* # *w*) ∩ *mon-c fg cn* = {}
from *trss-modify-context-s*[*OF IHP*(*1*)] *MON* **obtain** *csph* **where** *S1*: *ch* = *csph* + *c mon-c fg csph*={} ((*s*, *cn*), *e*, *sh*, *csph* + *cn*) ∈ *trss fg* **by** *auto*
with *MON* **have** *mon-w fg w* ∩ *mon-c fg* (*csph*+*cn*) = {} **by** (*auto simp add*: *mon-c-unconc*)
with *IHP*(*3*)[*rule-format*] **obtain** *csp* **where** *S2*: *c′*=*csp*+*ch mon-c fg csp*={} ((*sh*,*csph*+*cn*),*w*,(*s′*,*csp*+(*csph*+*cn*)))∈*trcl* (*trss fg*) **by** *blast*
from *S1 S2* **have** *c′*=(*csp*+*csph*)+*c mon-c fg* (*csp*+*csph*)={} ((*s*,*cn*),*e*#*w*,(*s′*,(*csp*+*csph*)+*cn*))∈*trcl* (*trss fg*) **by** (*auto simp add*: *union-assoc mon-c-unconc*)
thus ∃ *csp*. *c′* = *csp* + *c* ∧ *mon-c fg csp* = {} ∧ ((*s*, *cn*), *e* # *w*, *s′*, *csp* + *cn*) ∈ *trcl* (*trss fg*) **by** *blast*
qed
qed

lemma *trss-add-context-s*:
⟦((*s*,*c*),*e*,(*s′*,*c′*))∈*trss fg*; *mon-e fg e* ∩ *mon-c fg ce* = {}⟧
⟹ ((*s*,*c*+*ce*),*e*,(*s′*,*c′*+*ce*))∈*trss fg*
by (*auto elim*!: *trss.cases intro*!: *trss.intros simp add*: *union-assoc mon-c-unconc*)

lemma *trss-add-context*:
⟦((*s*,*c*),*w*,(*s′*,*c′*))∈*trcl* (*trss fg*); *mon-w fg w* ∩ *mon-c fg ce* = {}⟧
⟹ ((*s*,*c*+*ce*),*w*,(*s′*,*c′*+*ce*))∈*trcl* (*trss fg*)
proof (*induct rule*: *trcl-pair-induct*)
case *empty* **thus** *?case* **by** *simp*
next
case (*cons s c e sh ch w s′ c′*) **note** *IHP*=*this*
from *IHP*(*4*) **have** *MM*: *mon-e fg e* ∩ *mon-c fg ce* = {} *mon-w fg w* ∩ *mon-c fg ce* = {} **by** *auto*
from *trcl.cons*[*OF trss-add-context-s*[*OF IHP*(*1*) *MM*(*1*)] *IHP*(*3*)[*OF MM*(*2*)]] **show** *?case* **.**
qed

lemma *trss-drop-context-s*: ⟦ ((*s*,*c*+*ce*),*e*,(*s′*,*c′*+*ce*))∈*trss fg* ⟧
⟹ ((*s*,*c*),*e*,(*s′*,*c′*))∈*trss fg* ∧ *mon-e fg e* ∩ *mon-c fg ce* = {}
by (*erule trss.cases*) (*auto intro*!: *trss.intros simp add*: *mon-c-unconc union-assoc*[*of* - *c ce*, *symmetric*])

lemma *trss-drop-context*: !!*s c*. ⟦ ((*s*,*c*+*ce*),*w*,(*s′*,*c′*+*ce*))∈*trcl* (*trss fg*) ⟧
⟹ ((*s*,*c*),*w*,(*s′*,*c′*))∈*trcl* (*trss fg*) ∧ *mon-w fg w* ∩ *mon-c fg ce* = {}
proof (*induct w*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons e w*) **note** *IHP*=*this*
then obtain *sh ch* **where** *SPLIT*: ((*s*,*c*+*ce*),*e*,(*sh*,*ch*))∈*trss fg* ((*sh*,*ch*),*w*,(*s′*,*c′*+*ce*))∈ *trcl* (*trss fg*) **by** (*fast dest*: *trcl-uncons*)
from *trss-c-fmt-s*[*OF SPLIT*(*1*)] **obtain** *csp* **where** *CHFMT*: *ch* = (*csp* + *c*) + *ce* **by** (*auto simp add*: *union-assoc*)
from *CHFMT trss-drop-context-s SPLIT*(*1*) **have** ((*s*,*c*),*e*,(*sh*,*csp*+*c*))∈*trss fg*

mon-e fg e ∩ *mon-c fg ce* = {} **by** *blast+*
moreover from *CHFMT IHP*(*1*) *SPLIT*(*2*) **have** ((*sh*,*csp*+*c*),*w*,(*s*′,*c*′))∈*trcl* (*trss fg*) *mon-w fg w* ∩ *mon-c fg ce* = {} **by** *blast+*
ultimately show *?case* **by** *auto*
qed

lemma *trss-xchange-context-s*:
assumes *A*: ((*s*,*c*),*e*,(*s*′,*csp*+*c*))∈*trss fg*
and *M*:*mon-c fg cn* ⊆ *mon-c fg c*
shows ((*s*,*cn*),*e*,(*s*′,*csp*+*cn*))∈*trss fg*
proof −
from *trss-drop-context-s*[*of* - {#}, *simplified*, *OF A*] **have** *DC*: ((*s*, {#}), *e*, *s*′, *csp*) ∈ *trss fg mon-e fg e* ∩ *mon-c fg c* = {} **by** *simp-all*
with *M* **have** *mon-e fg e* ∩ *mon-c fg cn* = {} **by** *auto*
from *trss-add-context-s*[*OF DC*(*1*) *this*] **show** *?thesis* **by** *auto*
qed

lemma *trss-xchange-context*:
assumes *A*: ((*s*,*c*),*w*,(*s*′,*csp*+*c*))∈*trcl* (*trss fg*)
and *M*:*mon-c fg cn* ⊆ *mon-c fg c*
shows ((*s*,*cn*),*w*,(*s*′,*csp*+*cn*))∈*trcl* (*trss fg*)
proof −
from *trss-drop-context*[*of* - {#}, *simplified*, *OF A*] **have** *DC*: ((*s*, {#}), *w*, *s*′, *csp*) ∈ *trcl* (*trss fg*) *mon-w fg w* ∩ *mon-c fg c* = {} **by** *simp-all*
with *M* **have** *mon-w fg w* ∩ *mon-c fg cn* = {} **by** *auto*
from *trss-add-context*[*OF DC*(*1*) *this*] **show** *?thesis* **by** *auto*
qed

lemma *trss-drop-all-context-s*[*cases set*, *case-names dropped*]:
assumes *A*: ((*s*,*c*),*e*,(*s*′,*c*′))∈*trss fg*
and *C*: !!*csp*. ⟦ *c*′=*csp*+*c*; ((*s*,{#}),*e*,(*s*′,*csp*))∈*trss fg* ⟧ ⟹ *P*
shows *P*
using *A* **proof** (*cases rule*: *trss-c-cases-s*)
case *no-spawn* **with** *trss-xchange-context-s*[*of s c e s*′ {#} *fg* {#}] *A C* **show** *P* **by** *auto*
next
case (*spawn p u v*) **with** *trss-xchange-context-s*[*of s c e s*′ {#[*entry fg p*]#} *fg* {#}] *A C* **show** *P* **by** *auto*
qed

lemma *trss-drop-all-context*[*cases set*, *case-names dropped*]:
assumes *A*: ((*s*,*c*),*w*,(*s*′,*c*′))∈*trcl* (*trss fg*)
and *C*: !!*csp*. ⟦ *c*′=*csp*+*c*; ((*s*,{#}),*w*,(*s*′,*csp*))∈*trcl* (*trss fg*)⟧ ⟹ *P*
shows *P*
using *A* **proof** (*cases rule*: *trss-c-cases*)
case (*c-case csp*) **with** *trss-xchange-context*[*of s c w s*′ *csp fg* {#}] *A C* **show** *P* **by** *auto*
qed

lemma *tr-add-context-s*:
 $[\![$ $(c,e,c')\in tr\ fg$; $mon\text{-}e\ fg\ e \cap mon\text{-}c\ fg\ ce = \{\}$ $]\!] \Longrightarrow (c+ce,e,c'+ce)\in tr\ fg$
 by (*erule gtrE*) (*auto simp add*: *mon-c-unconc union-assoc intro*: *gtrI-s dest*: *trss-add-context-s*)

lemma *tr-add-context*:
 $[\![$ $(c,w,c')\in trcl\ (tr\ fg)$; $mon\text{-}w\ fg\ w \cap mon\text{-}c\ fg\ ce = \{\}$ $]\!]$
 $\Longrightarrow (c+ce,w,c'+ce)\in trcl\ (tr\ fg)$
proof (*induct rule*: *trcl.induct*)
 case *empty* **thus** *?case* **by** *auto*
next
 case (*cons c e c' w c''*) **note** *IHP=this*
 from *tr-add-context-s*[*OF IHP*(*1*), *of ce*] *IHP*(*4*) **have** $(c + ce, e, c' + ce) \in tr\ fg$ **by** *auto*
 also from *IHP*(*3*,*4*) **have** $(c' + ce, w, c'' + ce) \in trcl\ (tr\ fg)$ **by** *auto*
 finally show *?case* **.**
qed

end

8 Normalized Paths

theory *Normalization*
imports *Main ThreadTracking Semantics ConsInterleave*
begin

The idea of normalized paths is to consider particular schedules only. While the original semantics allows a context switch to occur after every single step, we now define a semantics that allows context switches only before non-returning calls or after a thread has reached its final stack. We then show that this semantics is able to reach the same set of configurations as the original semantics.

8.1 Semantic properties of restricted flowgraphs

It makes the formalization smoother, if we assume that every thread's execution begins with a non-returning call. For this purpose, we defined syntactic restrictions on flowgraphs already (cf. Section 6.3). We now show that these restrictions have the desired semantic effect.

— Procedures with isolated return nodes will never return
lemma (**in** *eflowgraph*) *iso-ret-no-ret*: !!*u c*. $[\![$
 isolated-ret fg p;
 proc-of fg u = *p*;
 $u \neq$ *return fg p*;
 $(([u],c),w,([return\ fg\ p'],c'))\in trcl\ (trss\ fg)$
 $]\!] \Longrightarrow$ *False*
proof (*induct w rule*: *length-compl-induct*)

case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons e w*) **note** *IHP=this*
then obtain *sh ch* **where** *SPLIT1*: (([*u*],*c*),*e*,(*sh*,*ch*))∈*trss fg* **and** *SPLIT2*: ((*sh*,*ch*),*w*,([*return fg p'*],*c'*))∈*trcl* (*trss fg*) **by** (*fast dest*: *trcl-uncons*)
show *?case* **proof** (*cases e*)
case *LRet* **with** *SPLIT1 IHP*(*3*,*4*) **show** *False* **by** (*auto elim*!: *trss.cases*)
next
case *LBase* **with** *SPLIT1 IHP*(*2*,*3*) **obtain** *v* **where** *A*: *sh*=[*v*] *proc-of fg v* = *p v*≠*return fg p* **by** (*force elim*!: *trss.cases simp add*: *edges-part isolated-ret-def*)
with *IHP SPLIT2* **show** *False* **by** *auto*
next
case (*LSpawn q*) **with** *SPLIT1 IHP*(*2*,*3*) **obtain** *v* **where** *A*: *sh*=[*v*] *proc-of fg v* = *p v*≠*return fg p* **by** (*force elim*!: *trss.cases simp add*: *edges-part isolated-ret-def*)
with *IHP SPLIT2* **show** *False* **by** *auto*
next
case (*LCall q*) **with** *SPLIT1 IHP*(*2*,*3*) **obtain** *uh* **where** *A*: *sh*=*entry fg q*#[*uh*] *proc-of fg uh* = *p uh*≠*return fg p* **by** (*force elim*!: *trss.cases simp add*: *edges-part isolated-ret-def*)
with *SPLIT2* **have** *B*: ((*entry fg q*#[*uh*],*ch*),*w*,([*return fg p'*],*c'*))∈*trcl* (*trss fg*) **by** *simp*
from *trss-return-cases*[*OF B*] **obtain** *w1 w2 ct* **where** *C*: *w*=*w1*@*w2 length w2* ≤ *length w* ((([*entry fg q*],*ch*),*w1*,([],*ct*))∈*trcl* (*trss fg*) (([*uh*],*ct*),*w2*,([*return fg p'*],*c'*))∈*trcl* (*trss fg*) **by** (*auto*)
from *IHP*(*1*)[*OF C*(*2*) *IHP*(*2*) *A*(*2*,*3*) *C*(*4*)] **show** *False* **.**
qed
qed

— The first step of an initial procedure is a call
lemma (**in** *eflowgraph*) *initial-starts-with-call*:
⟦ (([*entry fg p*],*c*),*e*,(*s'*,*c'*))∈*trss fg*; *initialproc fg p* ⟧
⟹ ∃ *p'*. *e*=*LCall p'* ∧ *isolated-ret fg p'*
by (*auto elim*!: *trss.cases dest*: *initial-call-no-ret initial-no-ret entry-return-same-proc*)

— There are no same-level paths starting from the entry node of an initial procedure
lemma (**in** *eflowgraph*) *no-sl-from-initial*:
assumes *A*: *w*≠[] *initialproc fg p*
(([*entry fg p*],*c*),*w*,([*v*],*c'*))∈*trcl* (*trss fg*)
shows *False*
proof −
from *A* **obtain** *sh ch e w'* **where** *SPLIT*: (([*entry fg p*],*c*),*e*,(*sh*,*ch*))∈*trss fg* ((*sh*,*ch*),*w'*,([*v*],*c'*))∈*trcl* (*trss fg*) **by** (*cases w*, *simp*, *fast dest*: *trcl-uncons*)
from *initial-starts-with-call*[*OF SPLIT*(*1*) *A*(*2*)] **obtain** *p'* **where** *CE*: *e*=*LCall p' isolated-ret fg p'* **by** *blast*
with *SPLIT*(*1*) **obtain** *u'* **where** *sh*=*entry fg p'*#[*u'*] **by** (*auto elim*!: *trss.cases*)
with *SPLIT*(*2*) **have** ((*entry fg p'*#[*u'*],*ch*),*w'*,([*v*],*c'*))∈*trcl* (*trss fg*) **by** *simp*
then obtain *wa ct* **where** (([*entry fg p'*],*ch*),*wa*,([],*ct*))∈*trcl* (*trss fg*) **by** (*erule-tac trss-return-cases*, *auto*)
then obtain *wa' p''* **where** (([*entry fg p'*],*ch*),*wa'*,([*return fg p''*],*ct*))∈*trcl* (*trss*

fg) **by** (*blast dest*: *trss-2empty-to-2return*)
from *iso-ret-no-ret*[*OF CE*(*2*) - - *this*] *CE*(*2*)[*unfolded isolated-ret-def*] **show** *?thesis* **by** *simp*
qed

— There are no same-level or returning paths starting from the entry node of an initial procedure
lemma (**in** *eflowgraph*) *no-retsl-from-initial*:
assumes *A*: $w \neq []$
initialproc fg p
$(([entry\ fg\ p],c),w,(r',c')) \in trcl\ (trss\ fg)$
$length\ r' \leq 1$
shows *False*
proof (*cases* r')
case *Nil* **with** *A*(*3*) **have** $(([entry\ fg\ p],c),w,([],c')) \in trcl\ (trss\ fg)$ **by** *simp*
from *trss-2empty-to-2return*[*OF this*, *simplified*] **obtain** w' q **where** *B*: $w=w'@[LRet]$ $(([entry\ fg\ p],\ c),\ w',\ [return\ fg\ q],\ c') \in trcl\ (trss\ fg)$ **by** (*blast*)
show *?thesis* **proof** (*cases* w')
case *Nil* **with** *B* **have** $p=q$ $entry\ fg\ p = return\ fg\ p$ **by** (*auto dest*: *trcl-empty-cons entry-return-same-proc*)
with *A*(*2*) *initial-no-ret* **show** *False* **by** *blast*
next
case *Cons* **hence** $w' \neq []$ **by** *simp*
from *no-sl-from-initial*[*OF this A*(*2*) *B*(*2*)] **show** *False* **.**
qed
next
case (*Cons u rr*) **with** *A*(*4*) **have** $r'=[u]$ **by** *auto*
with *no-sl-from-initial*[*OF A*(*1*,*2*)] *A*(*3*) **show** *False* **by** *auto*
qed

8.2 Definition of normalized paths

In order to describe the restricted schedules, we define an operational semantics that performs an atomically scheduled sequence of steps in one step, called a *macrostep*. Context switches may occur after macrosteps only. We call this the *normalized semantics* and a sequence of macrosteps a *normalized path*.

Since we ensured that every path starts with a non-returning call, we can define a macrostep as an initial call followed by a same-level path[2] of the called procedure. This has the effect that context switches are either performed before a non-returning call (if the thread makes a further macrostep in the future) or after the thread has reached its final configuration.

As for the original semantics, we first define the normalized semantics on a single thread with a context and then use the theory developed in Section 5

[2]Same-level paths are paths with balanced calls and returns. The stack-level at the beginning of their execution is the same as at the end, and during the execution, the stack never falls below the initial level.

to derive interleaving semantics on multisets and configurations with an explicit local thread (loc/env-semantics, cf. Section 5.4).

inductive-set
  ntrs :: ($'n$,$'p$,$'ba$,$'m$,$'more$) *flowgraph-rec-scheme* $\Rightarrow$
      (($'n$ *list* $\times$ $'n$ *conf*) $\times$ ($'p$,$'ba$) *label list* $\times$ ($'n$ *list* $\times$ $'n$ *conf*)) *set*
  for *fg*
  where
  — A macrostep transforms one thread by first calling a procedure and then doing a same-level path
  ntrs-step: $[\![((u\#r,ce),LCall\ p,\ (entry\ fg\ p\ \#\ u'\ \#\ r,ce))\in trss\ fg;$
    $(([entry\ fg\ p],ce),w,([v],ce'))\in trcl\ (trss\ fg)]\!] \Longrightarrow$
    $((u\#r,ce),LCall\ p\#w,(v\#u'\#r,ce'))\in ntrs\ fg$

abbreviation *ntr* **where** *ntr fg* == *gtr* (*ntrs fg*)
abbreviation *ntrp* **where** *ntrp fg* == *gtrp* (*ntrs fg*)

interpretation *ntrs*: *env-no-step ntrs fg*
  apply (*rule env-no-step.intro*)
  apply (*erule ntrs.cases*)
  apply *clarsimp*
  apply (*erule trss-c-cases*)
  apply *auto*
  done

8.3 Representation property for reachable configurations

In this section, we show that a configuration is reachable if and only if it is reachable via a normalized path.

The first direction is to show that a normalized path is also a path. This follows from the definitions. Note that we first show that a single macrostep corresponds to a path and then generalize the result to sequences of macrosteps

lemma *ntrs-is-trss-s*: $((s,c),w,(s',c'))\in ntrs\ fg \Longrightarrow ((s,c),w,(s',c'))\in trcl\ (trss\ fg)$
proof (*erule ntrs.cases*, *auto*)
  fix $p\ r\ u\ u'\ v\ w$
  assume *A*: $((u\ \#\ r,\ c),\ LCall\ p,\ entry\ fg\ p\ \#\ u'\ \#\ r,\ c) \in trss\ fg\ (([entry\ fg\ p],\ c),\ w,\ [v],\ c') \in trcl\ (trss\ fg)$
  from *trss-stack-comp*[*OF A*(*2*), *of* $u'\#r$] **have** $((entry\ fg\ p\ \#\ u'\ \#\ r,\ c),\ w,\ v\ \#\ u'\ \#\ r,\ c') \in trcl\ (trss\ fg)$ **by** *simp*
  with *A*(*1*) **show** $((u\ \#\ r,\ c),\ LCall\ p\ \#\ w,\ v\ \#\ u'\ \#\ r,\ c') \in trcl\ (trss\ fg)$ **by** *auto*
qed

lemma *ntrs-is-trss*: $((s,c),w,(s',c'))\in trcl\ (ntrs\ fg)$
  $\Longrightarrow ((s,c),foldl\ (@)\ []\ w,(s',c'))\in trcl\ (trss\ fg)$
proof (*induct rule*: *trcl-pair-induct*)
  case *empty* **thus** *?case* **by** *simp*

next
  case (*cons s c e sh ch w s′ c′*) **note** *IHP=this*
  from *trcl-concat*[*OF ntrs-is-trss-s*[*OF IHP*(*1*)] *IHP*(*3*)] *foldl-conc-empty-eq*[*of e w*] **show** *?case* **by** *simp*
qed

lemma *ntr-is-tr-s*: $(c,w,c')\in ntr\ fg \Longrightarrow (c,w,c')\in trcl\ (tr\ fg)$
  by (*erule gtrE*) (*auto dest*: *ntrs-is-trss-s intro*: *gtrI*)

lemma *ntr-is-tr*: $(c,ww,c')\in trcl\ (ntr\ fg) \Longrightarrow (c,foldl\ (@)\ []\ ww,c')\in trcl\ (tr\ fg)$
proof (*induct rule*: *trcl.induct*)
  case *empty* **thus** *?case* **by** *auto*
next
  case (*cons c ee c′ ww c″*) **note** *IHP=this*
  from *trcl-concat*[*OF ntr-is-tr-s*[*OF IHP*(*1*)] *IHP*(*3*)] *foldl-conc-empty-eq*[*of ee ww*] **show** *?case* **by** (*auto*)
qed

The other direction requires to prove that for each path reaching a configuration there is also a normalized path reaching the same configuration. We need an auxiliary lemma for this proof, that is a kind of append rule: *Given a normalized path reaching some configuration c, and a same level or returning path from some stack in c, we can derive a normalized path to c modified according to the same-level path.* We cannot simply append the same-level or returning path as a macrostep, because it does not start with a non-returning call. Instead, we will have to append it to some macrostep in the normalized path, i.e. move it „left" into the normalized path.

Intuitively, we can describe the concept of the proof as follows: Due to the restrictions we made on flowgraphs, a same-level or returning path cannot be the first steps on a thread. Hence there is a last macrostep that was executed on the thread. When this macrostep was executed, all threads held less monitors then they do at the end of the execution, because the set of monitors held by every single thread is increasing during the execution of a normalized path. Thus we can append the same-level or returning path to the last macrostep on that thread. As a same-level or returning path does not allocate any monitors, the following macrosteps remain executable. If we have a same-level path, appending it to a macrostep yields a valid macrostep again and we are done. Appending a returning path to a macrostep yields a same-level path. In this case we inductively repeat our argument.

The actual proof is strictly inductive; it either appends the same-level path to the *last* macrostep or inductively repeats the argument.

lemma (**in** *eflowgraph*) *ntr-sl-move-left*: !!*ce u r w r′ ce′*.
  ⟦ ({#[*entry fg p*]#},*ww*,{# *u*#*r* #}+*ce*)∈*trcl* (*ntr fg*);
    (([*u*],*ce*),*w*,(*r′*,*ce′*))∈*trcl* (*trss fg*);
    initialproc fg p;
    length r′ ≤ *1*; *w*≠[]

$\rrbracket \Longrightarrow \exists\, ww'.$ ({#[*entry fg p*]#}, *ww'*,{# *r'*@*r* #}+*ce'*)∈*trcl* (*ntr fg*)
proof (*induct ww rule*: *rev-induct*)
case *Nil* **note** *CC=this* **hence** *u=entry fg p* **by** *auto*
— If the normalized path is empty, we get a contradiction, because there is no same-level path from the initial configuration of a thread
with *CC*(*2*) *no-retsl-from-initial*[*OF CC*(*5*,*3*) - *CC*(*4*)] **have** *False* **by** *blast*
thus *?case* **..**
next
case (*snoc ee ww*) **note** *IHP=this*
— In the induction step, we extract the last macrostep
then obtain *ch* **where** *SPLIT*: ({#[*entry fg p*]#},*ww*,*ch*)∈*trcl* (*ntr fg*) (*ch*,*ee*,{# *u*#*r* #}+*ce*)∈*ntr fg* **by** (*fast dest*: *trcl-rev-uncons*)
— The last macrostep first executes a call and then a same-level path
from *SPLIT*(*2*) **obtain** *q wws uh rh ceh uh' vt cet* **where**
STEPFMT: *ee=LCall q*#*wws ch=add-mset* (*uh*#*rh*) *ceh add-mset* (*u*#*r*) *ce* = *add-mset* (*vt*#*uh'*#*rh*) *cet* ((*uh*#*rh*,*ceh*),*LCall q*,(*entry fg q*#*uh'*#*rh*,*ceh*))∈*trss fg* (([*entry fg q*],*ceh*),*wws*,([*vt*],*cet*))∈*trcl* (*trss fg*)
by (*auto elim*!: *gtrE ntrs.cases*[*simplified*])
— Make a case distinction whether the last step was executed on the same thread as the sl/ret-path or not
from *STEPFMT*(*3*) **show** *?case* **proof** (*cases rule*: *mset-single-cases'*)
— If the sl/ret path was executed on the same thread as the last macrostep
case *loc* **note** *CASE=this* **hence** *C'*: *u=vt r=uh'*#*rh ce=cet* **by** *auto*
— we append it to the last macrostep.
with *STEPFMT*(*5*) *IHP*(*3*) **have** *NEWPATH*: (([*entry fg q*],*ceh*),*wws*@*w*,(*r'*,*ce'*))∈*trcl* (*trss fg*) **by** (*simp add*: *trcl-concat*)
— We then distinguish whether we appended a same-level or a returning path
show *?thesis* **proof** (*cases r'*)
— If we appended a same-level path
case (*Cons v'*) — Same-level path **with** *IHP*(*5*) **have** *CC*: *r'*=[*v'*] **by** *auto*
— The macrostep still ends with a same-level path
with *NEWPATH* **have** (([*entry fg q*],*ceh*),*wws*@*w*,([*v'*],*ce'*))∈*trcl* (*trss fg*) **by** *simp*
— and thus remains a valid macrostep
from *gtrI-s*[*OF ntrs-step*[*OF STEPFMT*(*4*), *simplified*, *OF this*]] **have** (*add-mset* (*uh* # *rh*) *ceh*, *LCall q* # *wws*@*w*, *add-mset* (*v'* # *uh'* # *rh*) *ce'*) ∈ *ntr fg* **.**
— that we can append to the prefix of the normalized path to get our proposition
with *STEPFMT*(*2*) *SPLIT*(*1*) *CC C'*(*2*) **have** ({#[*entry fg p*]#},*ww*@[*LCall q*#*wws*@*w*],{# *r'*@*r* #} + *ce'*)∈*trcl* (*ntr fg*) **by** (*auto simp add*: *trcl-rev-cons*)
thus *?thesis* **by** *blast*
next
— If we appended a returning path
case *Nil* **note** *CC=this*
— The macrostep now ends with a returning path, and thus gets a same-level path
have *NEWSL*: (([*uh*], *ceh*), *LCall q* # *wws* @ *w*, [*uh'*], *ce'*) ∈ *trcl* (*trss fg*)
proof –
from *STEPFMT*(*4*) **have** (([*uh*],*ceh*),*LCall q*,(*entry fg q*#[*uh'*],*ceh*))∈*trss*

fg **by** (*auto elim*!: *trss.cases intro*: *trss.intros*)
also from *trss-stack-comp*[*OF NEWPATH*] *CC* **have** ((*entry fg q*#[*uh'*],*ceh*),*wws*@*w*,([*uh'*],*ce'*))$\in$*trcl* (*trss fg*) **by** *auto*
finally show *?thesis* **.**
qed
— Hence we can apply the induction hypothesis and get the proposition
from *IHP*(*1*)[*OF - NEWSL*] *SPLIT STEPFMT*(*2*) *IHP*(*4*) *CC C'*(*2*) **show** *?thesis* **by** *auto*
qed
next
— If the sl/ret path was executed on a different thread than the last macrostep
case (*env cc*) **note** *CASE=this*
— we first look at the context after the last macrostep. It consists of the threads that already have been there and the threads that have been spawned by the last macrostep
from *STEPFMT*(*5*) **obtain** *cspt* **where** *CETFMT*: *cet=cspt+ceh* !!*s*. *s* $\in$# *cspt* $\Longrightarrow$ $\exists$ *p*. *s*=[*entry fg p*] $\wedge$ *initialproc fg p*
by (*unfold initialproc-def*) (*erule trss-c-cases*, *blast*)
— The spawned threads do not hold any monitors yet
hence *CSPT-NO-MON*: *mon-c fg cspt* = {} **by** (*simp add*: *c-of-initial-no-mon*)
— We now distinguish whether the sl/ret path is executed on a thread that was just spawned or on a thread that was already there
from *CASE*(*1*) *CETFMT*(*1*) **have** *u*#*r* $\in$# *cspt+ceh* **by** *auto*
thus *?thesis* **proof** (*cases rule*: *mset-un-cases*[*cases set*])
— The sl/ret path cannot have been executed on a freshly spawned thread due to the restrictions we made on the flowgraph
case *left* — Thread was spawned **with** *CETFMT* **obtain** *q* **where** *u=entry fg q r*=[] *initialproc fg q* **by** *auto*
with *IHP*(*3*,*5*,*6*) *no-retsl-from-initial* **have** *False* **by** *blast*
thus *?thesis* **..**
next
— Hence let's assume the sl/ret path is executed on a thread that was already there before the last macrostep
case *right* **note** *CC=this*
— We can write the configuration before the last macrostep in a way that one sees the thread that executed the sl/ret path
hence *CEHFMT*: *ceh*={# *u*#*r* #}+(*ceh*−{# *u*#*r* #}) **by** *auto*
have *CHFMT*: *ch* = {# *u*#*r* #} + ({# *uh*#*rh* #}+(*ceh*−{# *u*#*r* #})) **proof** −
from *CEHFMT STEPFMT*(*2*) **have** *ch* = {# *uh*#*rh* #} + ({# *u*#*r* #}+(*ceh*−{# *u*#*r* #})) **by** *simp*
thus *?thesis* **by** (*auto simp add*: *union-ac*)
qed
— There are not more monitors than after the last macrostep
have *MON-CE*: *mon-c fg* ({# *uh*#*rh* #}+(*ceh*−{# *u*#*r* #})) $\subseteq$ *mon-c fg ce* **proof** −
have *mon-n fg uh* $\subseteq$ *mon-n fg uh'* **using** *STEPFMT*(*4*) **by** (*auto elim*!: *trss.cases dest*: *mon-n-same-proc edges-part*)
moreover have *mon-c fg* (*ceh*−{#*u*#*r*#}) $\subseteq$ *mon-c fg cc* **proof** −

from *CASE*(*3*) *CETFMT* **have** $cc{=}(cspt{+}ceh){-}\{\#u\#r\#\}$ **by** *simp*
with *CC* **have** $cc = cspt{+}(ceh{-}\{\#u\#r\#\})$ **by** *auto*
with *CSPT-NO-MON* **show** *?thesis* **by** (*auto simp add*: *mon-c-unconc*)
qed
ultimately show *?thesis* **using** *CASE*(*2*) **by** (*auto simp add*: *mon-c-unconc*)
qed

— The same-level path preserves the threads in its environment and the threads that it creates hold no monitors

from *IHP*(*3*) **obtain** csp' **where** $CE'FMT$: $ce'{=}csp'{+}ce$ $mon\text{-}c\ fg\ csp' = \{\}$ **by** (−) (*erule trss-c-cases*, *blast intro*!: *c-of-initial-no-mon*)

— We can execute the sl/ret-path also from the configuration before the last step

from *trss-xchange-context*[*OF* - *MON-CE*] *IHP*(*3*) $CE'FMT$ **have** *NSL*: $(([u], \{\#uh \# rh\#\} + (ceh - \{\#u \# r\#\})),\ w,\ r',\ csp' + (\{\#uh \# rh\#\} + (ceh - \{\#u \# r\#\}))) \in trcl\ (trss\ fg)$ **by** *auto*

— And with the induction hypothesis we get a normalized path

from *IHP*(*1*)[*OF* - *NSL IHP*(*4*,*5*,*6*)] *SPLIT*(*1*) *CHFMT* **obtain** ww' **where** *NNPATH*: $(\{\#[entry\ fg\ p]\#\},\ ww',\ \{\#r' @ r\#\} + (csp' + (\{\#uh \# rh\#\} + (ceh - \{\#u \# r\#\})))) \in trcl\ (ntr\ fg)$ **by** *blast*

— We now show that the last macrostep can also be executed from the new configuration, after the sl/ret path has been executed (on another thread)

have $(\{\#r' @ r\#\} + (csp' + (\{\#uh \# rh\#\} + (ceh - \{\#u \# r\#\}))),\ ee,\ \{\#vt \# uh' \# rh\#\} + (cspt + (\{\#r' @ r\#\} + (csp' + (ceh - \{\#u \# r\#\}))))) \in ntr\ fg$

proof −

— This is because the sl/ret path has not allocated any monitors

have *MON-CEH*: $mon\text{-}c\ fg\ (\{\#r' @ r\#\} + (csp' + (ceh - \{\#u \# r\#\}))) \subseteq mon\text{-}c\ fg\ ceh$ **proof** −

from *IHP*(*3*,*5*) *trss-bot-proc-const*[*of* [] *u ce w* [] - ce'] *mon-n-same-proc* **have** $mon\text{-}s\ fg\ r' \subseteq mon\text{-}n\ fg\ u$ **by** (*cases* r') (*simp*, *force*)

moreover from *CEHFMT* **have** $mon\text{-}c\ fg\ ceh = mon\text{-}c\ fg\ (\{\#u \# r\#\} + (ceh - \{\#u \# r\#\}))$ **by** *simp* — Need to state this explicitly because of recursive simp rule $ceh = \{\#u \# r\#\} + (ceh - \{\#u \# r\#\})$

ultimately show *?thesis* **using** $CE'FMT$(*2*) **by** (*auto simp add*: *mon-c-unconc mon-s-unconc*)

qed

— And we can reassemble the macrostep within the new context

note *trss-xchange-context-s*[*OF* - *MON-CEH*, **where** $csp{=}\{\#\}$, *simplified*, *OF STEPFMT*(*4*)]

moreover from *trss-xchange-context*[*OF* - *MON-CEH*, *of* [*entry fg q*] *wws* [*vt*] *cspt*] *STEPFMT*(*5*) *CETFMT*(*1*) **have**
$(([entry\ fg\ q], \{\#r' @ r\#\} + (csp' + (ceh - \{\#u \# r\#\}))),\ wws,\ [vt],\ cspt + (\{\#r' @ r\#\} + (csp' + (ceh - \{\#u \# r\#\})))) \in trcl\ (trss\ fg)$ **by** *blast*

moreover note *STEPFMT*(*1*)

ultimately have $((uh\#rh,(\{\#r' @ r\#\} + (csp' + (ceh - \{\#u \# r\#\})))),ee,(vt\#uh'\#rh,cspt{+}(\{\#r' @ r\#\} + (csp' + (ceh - \{\#u \# r\#\}))))) {\in} ntrs\ fg$ **by** (*auto intro*: *ntrs.intros*)

from *gtrI-s*[*OF this*] **show** *?thesis* **by** (*simp add*: *add-mset-commute*)

qed

— Finally we append the last macrostep to the normalized paths we obtained by the induction hypothesis

from *trcl-rev-cons*[*OF NNPATH this*] **have** $(\{\#[entry\ fg\ p]\#\},\ ww' @ [ee], \{\#vt\ \#\ uh'\ \#\ rh\#\} + (cspt + (\{\#r' @ r\#\} + (csp' + (ceh - \{\#u\ \#\ r\#\}))))) \in trcl\ (ntr\ fg)$ **.**

— And show that we got the right configuration

moreover from *CC CETFMT CASE*(*3*)[*symmetric*] *CASE*(*2*) *CE'FMT*(*1*) **have** $\{\#vt\ \#\ uh'\ \#\ rh\#\} + (cspt + (\{\#r' @ r\#\} + (csp' + (ceh - \{\#u\ \#\ r\#\})))) = \{\#\ r'@r\ \#\}+ce'$ **by** (*simp add*: *union-ac*)

ultimately show *?thesis* **by** *auto*

qed

qed

qed

Finally we can prove: *Any reachable configuration can also be reached by a normalized path.* With *eflowgraph.ntr-sl-move-left* we can easily show this lemma With *eflowgraph.ntr-sl-move-left* we can easily show this by induction on the reaching path. For the empty path, the proposition follows trivially. Else we consider the last step. If it is a call, we can execute it as a macrostep and get the proposition. Otherwise the last step is a same-level (Base, Spawn) or returning (Ret) path of length 1, and we can append it to the normalized path using *eflowgraph.ntr-sl-move-left*.

lemma (**in** *eflowgraph*) *normalize*: ⟦
$(cstart,w,c')\in trcl\ (tr\ fg)$;
$cstart=\{\#\ [entry\ fg\ p]\ \#\}$;
initialproc fg p ⟧
$\Longrightarrow \exists\, w'.\ (\{\#\ [entry\ fg\ p]\ \#\},w',c')\in trcl\ (ntr\ fg)$

— The lemma is shown by induction on the reaching path

proof (*induct rule*: *trcl-rev-induct*)

— The empty case is trivial, as the empty path is also a valid normalized path

case *empty* **thus** *?case* **by** (*auto intro*: *exI*[*of* - []])

next

case (*snoc cstart w c e c'*) **note** *IHP=this*

— In the inductive case, we can assume that we have an already normalized path and need to append a last step

then obtain w' **where** *IHP'*: $(\{\#\ [entry\ fg\ p]\ \#\},w',c)\in trcl\ (ntr\ fg)$ $(c,e,c')\in tr\ fg$ **by** *blast*

— We make explicit the thread on that this last step was executed

from *gtr-find-thread*[*OF IHP'*(*2*)] **obtain** *s ce s' ce'* **where** *TSTEP*: $c = add\text{-}mset\ s\ ce$ $c' = add\text{-}mset\ s'\ ce'$ $((s,\ ce),\ e,\ (s',\ ce')) \in trss\ fg$ **by** *blast*

— The proof is done by a case distinction whether the last step was a call or not

{

— Last step was a procedure call

fix *q*

assume *CASE*: *e=LCall q*

— As it is the last step, the procedure call will not return and thus is a valid macrostep

have $(c,LCall\ q\ \#\ [],\ c')\in ntr\ fg$ **using** *TSTEP CASE* **by** (*auto elim*!: *trss.cases*

intro!: *ntrs.intros gtrI-s trss.intros*)
— That can be appended to the initial normalized path
from *trcl-rev-cons*[*OF IHP′*(*1*) *this*] **have** *?case* **by** *blast*
} moreover {
— Last step was no procedure call
fix *q a*
assume *CASE*: $e=LBase\ a \vee e=LSpawn\ q \vee e=LRet$
— Then it is a same-level or returning path
with *TSTEP*(*3*) **obtain** *u r r′* **where** *SLR*: $s=u\#r$ $s'=r'@r$ $length\ r' \leq 1$ $(([u],ce),[e],(r',ce')) \in trcl\ (trss\ fg)$ **by** (*force elim*!: *trss.cases intro*!: *trss.intros*)
— That can be appended to the normalized path using the $[\![(\{\#[entry\ fg\ ?p]\#\}, ?ww, \{\#?u\ \#\ ?r\#\} + ?ce) \in trcl\ (ntr\ fg); (([?u], ?ce), ?w, ?r', ?ce') \in trcl\ (trss\ fg); initialproc\ fg\ ?p; length\ ?r' \leq 1; ?w \neq []]\!] \Longrightarrow \exists\, ww'. (\{\#[entry\ fg\ ?p]\#\}, ww', \{\#?r'\ @\ ?r\#\} + ?ce') \in trcl\ (ntr\ fg)$ - lemma
from *ntr-sl-move-left*[*OF - SLR*(*4*) *IHP*(*5*) *SLR*(*3*)] *IHP′*(*1*) *TSTEP*(*1*) *SLR*(*1*) **obtain** *ww′* **where** $(\{\#[entry\ fg\ p]\#\}, ww', \{\#r'\ @\ r\#\} + ce') \in trcl\ (ntr\ fg)$ **by** *auto*
with *SLR*(*2*) *TSTEP*(*2*) **have** *?case* **by** *auto*
} ultimately show *?case* **by** (*cases e*, *auto*)
qed

As the main result of this section we get: *A configuration is reachable if and only if it is also reachable via a normalized path*:

theorem (**in** *eflowgraph*) *ntr-repr*:
$(\exists\, w. (\{\#[entry\ fg\ (main\ fg)]\#\},w,c) \in trcl\ (tr\ fg))$
$\longleftrightarrow (\exists\, w. (\{\#[entry\ fg\ (main\ fg)]\#\},w,c) \in trcl\ (ntr\ fg))$
by (*auto simp add*: *initialproc-def intro*!: *normalize ntr-is-tr*)

8.4 Properties of normalized path

Like a usual path, also a macrostep modifies one thread, spawns some threads and preserves the state of all the other threads. The spawned threads do not make any steps, thus they stay in their initial configurations.

lemma *ntrs-c-cases-s*[*cases set*]: $[\![$
$((s,c),w,(s',c')) \in ntrs\ fg;$
$!!csp.\ [\![\ c'=csp+c;\ !!s.\ s \in\#\ csp \Longrightarrow \exists\, p\ u\ v.\ s=[entry\ fg\ p] \wedge$
$(u,Spawn\ p,v) \in edges\ fg \wedge$
$initialproc\ fg\ p$
$]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
by (*auto dest*!: *ntrs-is-trss-s elim*!: *trss-c-cases*)

lemma *ntrs-c-cases*[*cases set*]: $[\![$
$((s,c),ww,(s',c')) \in trcl\ (ntrs\ fg);$
$!!csp.\ [\![\ c'=csp+c;\ !!s.\ s \in\#\ csp \Longrightarrow \exists\, p\ u\ v.\ s=[entry\ fg\ p] \wedge$
$(u,Spawn\ p,v) \in edges\ fg \wedge$
$initialproc\ fg\ p$
$]\!] \Longrightarrow P$

$\rrbracket \Longrightarrow P$
by (*auto dest*!: *ntrs-is-trss elim*!: *trss-c-cases*)

8.4.1 Validity

Like usual paths, also normalized paths preserve validity of the configurations.

lemmas (**in** *flowgraph*) *ntrs-valid-preserve-s* = *trss-valid-preserve*[*OF ntrs-is-trss-s*]
lemmas (**in** *flowgraph*) *ntr-valid-preserve-s* = *tr-valid-preserve*[*OF ntr-is-tr-s*]
lemmas (**in** *flowgraph*) *ntrs-valid-preserve* = *trss-valid-preserve*[*OF ntrs-is-trss*]
lemmas (**in** *flowgraph*) *ntr-valid-preserve* = *tr-valid-preserve*[*OF ntr-is-tr*]
lemma (**in** *flowgraph*) *ntrp-valid-preserve-s*:
assumes A: $((s,c),e,(s',c'))\in$*ntrp fg*
and V: *valid fg* (*add-mset s c*)
shows *valid fg* (*add-mset* s' c')
using *ntr-valid-preserve-s*[*OF gtrp2gtr-s*[*OF A*] *V*] **by** *assumption*
lemma (**in** *flowgraph*) *ntrp-valid-preserve*:
assumes A: $((s,c),e,(s',c'))\in$*trcl* (*ntrp fg*)
and V: *valid fg* (*add-mset s c*)
shows *valid fg* (*add-mset* s' c')
using *ntr-valid-preserve*[*OF gtrp2gtr*[*OF A*] *V*] **by** *assumption*

8.4.2 Monitors

The following defines the set of monitors used by a normalized path and shows its basic properties:

definition
mon-ww fg ww == *foldl* ($\cup$) {} (*map* (*mon-w fg*) *ww*)

definition
mon-loc fg ww == *mon-ww fg* (*map le-rem-s* (*loc ww*))

definition
mon-env fg ww == *mon-ww fg* (*map le-rem-s* (*env ww*))

lemma *mon-ww-empty*[*simp*]: *mon-ww fg* [] = {}
by (*unfold mon-ww-def*, *auto*)
lemma *mon-ww-uncons*[*simp*]:
mon-ww fg (*ee#ww*) = *mon-w fg ee* $\cup$ *mon-ww fg ww*
by (*unfold mon-ww-def*, *auto simp add*: *foldl-un-empty-eq*[*of mon-w fg ee*])
lemma *mon-ww-unconc*:
mon-ww fg (*ww1@ww2*) = *mon-ww fg ww1* $\cup$ *mon-ww fg ww2*
by (*induct ww1*) *auto*

lemma *mon-env-empty*[*simp*]: *mon-env fg* [] = {}
by (*unfold mon-env-def*) *auto*
lemma *mon-env-single*[*simp*]:
mon-env fg [*e*] = (*case e of LOC a* $\Rightarrow$ {} | *ENV a* $\Rightarrow$ *mon-w fg a*)

by (*unfold mon-env-def*) (*auto split*: *el-step.split*)
lemma *mon-env-uncons*[*simp*]:
mon-env fg ($e\#w$)
$= ($*case e of LOC a* $\Rightarrow \{\} \mid$ *ENV a* $\Rightarrow$ *mon-w fg a*) $\cup$ *mon-env fg w*
by (*unfold mon-env-def*) (*auto split*: *el-step.split*)
lemma *mon-env-unconc*:
mon-env fg (*w1*@*w2*) = *mon-env fg w1* $\cup$ *mon-env fg w2*
by (*unfold mon-env-def*) (*auto simp add*: *mon-ww-unconc*)

lemma *mon-loc-empty*[*simp*]: *mon-loc fg* [] = {}
by (*unfold mon-loc-def*) *auto*
lemma *mon-loc-single*[*simp*]:
mon-loc fg [*e*] = (*case e of ENV a* $\Rightarrow \{\} \mid$ *LOC a* $\Rightarrow$ *mon-w fg a*)
by (*unfold mon-loc-def*) (*auto split*: *el-step.split*)
lemma *mon-loc-uncons*[*simp*]:
mon-loc fg ($e\#w$)
$= ($*case e of ENV a* $\Rightarrow \{\} \mid$ *LOC a* $\Rightarrow$ *mon-w fg a*) $\cup$ *mon-loc fg w*
by (*unfold mon-loc-def*) (*auto split*: *el-step.split*)
lemma *mon-loc-unconc*:
mon-loc fg (*w1*@*w2*) = *mon-loc fg w1* $\cup$ *mon-loc fg w2*
by (*unfold mon-loc-def*) (*auto simp add*: *mon-ww-unconc*)

lemma *mon-ww-of-foldl*[*simp*]: *mon-w fg* (*foldl* (@) [] *ww*) = *mon-ww fg ww*
apply (*induct ww*)
apply (*unfold mon-ww-def*)
apply *simp*
apply *simp*
apply (*subst foldl-conc-empty-eq*, *subst foldl-un-empty-eq*)
apply (*simp add*: *mon-w-unconc*)
done

lemma *mon-ww-ileq*: $w \preceq w' \Longrightarrow$ *mon-ww fg w* $\subseteq$ *mon-ww fg* w'
by (*induct rule*: *less-eq-list-induct*) *auto*

lemma *mon-ww-cil*:
$w \in w1 \otimes_\alpha w2 \Longrightarrow$ *mon-ww fg w* = *mon-ww fg w1* $\cup$ *mon-ww fg w2*
by (*induct rule*: *cil-set-induct-fix*α) *auto*
lemma *mon-loc-cil*:
$w \in w1 \otimes_\alpha w2 \Longrightarrow$ *mon-loc fg w* = *mon-loc fg w1* $\cup$ *mon-loc fg w2*
by (*induct rule*: *cil-set-induct-fix*α) *auto*
lemma *mon-env-cil*:
$w \in w1 \otimes_\alpha w2 \Longrightarrow$ *mon-env fg w* = *mon-env fg w1* $\cup$ *mon-env fg w2*
by (*induct rule*: *cil-set-induct-fix*α) *auto*

lemma *mon-ww-of-le-rem*:
mon-ww fg (*map le-rem-s w*) = *mon-loc fg w* $\cup$ *mon-env fg w*
by (*induct w*) (*auto split*: *el-step.split*)

lemma *mon-env-ileq*: $w \preceq w' \Longrightarrow$ *mon-env fg w* $\subseteq$ *mon-env fg w'*
by (*induct rule*: *less-eq-list-induct*) *auto*
lemma *mon-loc-ileq*: $w \preceq w' \Longrightarrow$ *mon-loc fg w* $\subseteq$ *mon-loc fg w'*
by (*induct rule*: *less-eq-list-induct*) *auto*

lemma *mon-loc-map-loc*[*simp*]: *mon-loc fg* (*map LOC w*) = *mon-ww fg w*
by (*unfold mon-loc-def*) *simp*
lemma *mon-env-map-env*[*simp*]: *mon-env fg* (*map ENV w*) = *mon-ww fg w*
by (*unfold mon-env-def*) *simp*
lemma *mon-loc-map-env*[*simp*]: *mon-loc fg* (*map ENV w*) = {}
by (*induct w*) *auto*
lemma *mon-env-map-loc*[*simp*]: *mon-env fg* (*map LOC w*) = {}
by (*induct w*) *auto*

— As monitors are syntactically bound to procedures, and each macrostep starts with a non-returning call, the set of monitors allocated during the execution of a normalized path is monotonically increasing
lemma (**in** *flowgraph*) *ntrs-mon-increasing-s*: $((s,c),e,(s',c')) \in$ *ntrs fg*
$\Longrightarrow$ *mon-s fg s* $\subseteq$ *mon-s fg s'* $\wedge$ *mon-c fg c* = *mon-c fg c'*
apply (*erule ntrs.cases*)
apply (*auto simp add*: *trss-c-no-mon*)
apply (*subgoal-tac mon-n fg u* = *mon-n fg u'*)
apply (*simp*)
apply (*auto elim*!: *trss.cases dest*!: *mon-n-same-proc edges-part*)
done

lemma (**in** *flowgraph*) *ntr-mon-increasing-s*:
$(c,ee,c') \in$ *ntr fg* $\Longrightarrow$ *mon-c fg c* $\subseteq$ *mon-c fg c'*
by (*erule gtrE*) (*auto dest*: *ntrs-mon-increasing-s simp add*: *mon-c-unconc*)

lemma (**in** *flowgraph*) *ntrp-mon-increasing-s*: $((s,c),e,(s',c')) \in$ *ntrp fg*
$\Longrightarrow$ *mon-s fg s* $\subseteq$ *mon-s fg s'* $\wedge$ *mon-c fg c* $\subseteq$ *mon-c fg c'*
apply (*erule gtrp.cases*)
apply (*auto dest*: *ntrs-mon-increasing-s simp add*: *mon-c-unconc*)[]
apply (*intro conjI*)
apply (*auto dest*: *ntrs-mon-increasing-s simp add*: *mon-c-unconc*)[]
apply (*auto dest*: *ntrs-mon-increasing-s simp add*: *mon-c-unconc*)[]
apply (*erule ntrs-c-cases-s*)
apply (*auto simp*: *mon-c-unconc*)
done

lemma (**in** *flowgraph*) *ntrp-mon-increasing*: $((s,c),e,(s',c')) \in$ *trcl* (*ntrp fg*)
$\Longrightarrow$ *mon-s fg s* $\subseteq$ *mon-s fg s'* $\wedge$ *mon-c fg c* $\subseteq$ *mon-c fg c'*
by (*induct rule*: *trcl-rev-pair-induct*) (*auto dest*!: *ntrp-mon-increasing-s*)

8.4.3 Modifying the context

lemmas (**in** *flowgraph*) *ntrs-c-no-mon-s* = *trss-c-no-mon*[*OF ntrs-is-trss-s*]
lemmas (**in** *flowgraph*) *ntrs-c-no-mon* = *trss-c-no-mon*[*OF ntrs-is-trss*]

Also like a usual path, a normalized step must not use any monitors that are allocated by other threads

lemmas (**in** *flowgraph*) *ntrs-mon-e-no-ctx* = *trss-mon-w-no-ctx*[*OF ntrs-is-trss-s*]
lemma (**in** *flowgraph*) *ntrs-mon-w-no-ctx*:
  assumes *A*: $((s,c),w,(s',c'))\in trcl\ (ntrs\ fg)$
  shows *mon-ww fg w* $\cap$ *mon-c fg c* = {}
  using *trss-mon-w-no-ctx*[*OF ntrs-is-trss*[*OF A*]] **by** *simp*

lemma (**in** *flowgraph*) *ntrp-mon-env-e-no-ctx*:
  $((s,c),ENV\ e,(s',c'))\in ntrp\ fg \Longrightarrow$ *mon-w fg e* $\cap$ *mon-s fg s* = {}
  by (*auto elim*!: *gtrp.cases dest*!: *ntrs-mon-e-no-ctx simp add*: *mon-c-unconc*)
lemma (**in** *flowgraph*) *ntrp-mon-loc-e-no-ctx*:
  $((s,c),LOC\ e,(s',c'))\in ntrp\ fg \Longrightarrow$ *mon-w fg e* $\cap$ *mon-c fg c* = {}
  by (*auto elim*!: *gtrp.cases dest*!: *ntrs-mon-e-no-ctx*)

lemma (**in** *flowgraph*) *ntrp-mon-env-w-no-ctx*:
  $((s,c),w,(s',c'))\in trcl\ (ntrp\ fg) \Longrightarrow$ *mon-env fg w* $\cap$ *mon-s fg s* = {}
  by (*induct rule*: *trcl-rev-pair-induct*) (*unfold mon-env-def*, *auto split*: *el-step.split dest*!: *ntrp-mon-env-e-no-ctx ntrp-mon-increasing simp add*: *mon-ww-unconc*)

lemma (**in** *flowgraph*) *ntrp-mon-loc-w-no-ctx*:
  $((s,c),w,(s',c'))\in trcl\ (ntrp\ fg) \Longrightarrow$ *mon-loc fg w* $\cap$ *mon-c fg c* = {}
  by (*induct rule*: *trcl-rev-pair-induct*) (*unfold mon-loc-def*, *auto split*: *el-step.split dest*!: *ntrp-mon-loc-e-no-ctx ntrp-mon-increasing simp add*: *mon-ww-unconc*)

The next lemmas are rules how to add or remove threads while preserving the executability of a path

lemma (**in** *flowgraph*) *ntrs-modify-context-s*:
  assumes *A*: $((s,c),ee,(s',c'))\in ntrs\ fg$
  and *B*: *mon-w fg ee* $\cap$ *mon-c fg cn* = {}
  shows $\exists$ *csp*. $c'=csp+c \wedge$ *mon-c fg csp*={} $\wedge$ $((s,cn),ee,(s',csp+cn))\in ntrs\ fg$
proof –
  from *A* **obtain** *p r u u' v w* **where** *S*: $s=u\#r$ $ee=LCall\ p\#w$ $s'=v\#u'\#r$ $((u\#r,c),LCall\ p,(entry\ fg\ p\#u'\#r,c))\in trss\ fg$ $(([entry\ fg\ p],c),w,([v],c'))\in trcl\ (trss\ fg)$ **by** (*blast elim*!: *ntrs.cases*[*simplified*])
  with *trss-modify-context-s*[*OF S*(*4*)] *B* **have** $((u\#r,cn),LCall\ p,(entry\ fg\ p\#u'\#r,cn))\in trss\ fg$ **by** *auto*
  moreover from *S trss-modify-context*[*OF S*(*5*)] *B* **obtain** *csp* **where** $c'=csp+c$ *mon-c fg csp* = {} $(([entry\ fg\ p],cn),w,([v],csp+cn))\in trcl\ (trss\ fg)$ **by** *auto*
  ultimately show *?thesis* **using** *S* **by** (*auto intro*!: *ntrs-step*)
qed

lemma (**in** *flowgraph*) *ntrs-modify-context*[*rule-format*]:

$\llbracket((s,c),w,(s',c'))\in trcl\ (ntrs\ fg)\rrbracket$
$\Longrightarrow \forall\, cn.\ mon\text{-}ww\ fg\ w \cap mon\text{-}c\ fg\ cn = \{\}$
$\longrightarrow (\exists\, csp.\ c'=csp+c \wedge mon\text{-}c\ fg\ csp = \{\} \wedge$
$((s,cn),w,(s',csp+cn))\in trcl\ (ntrs\ fg))$
proof (*induct rule*: *trcl-pair-induct*)
case *empty* **thus** *?case* **by** *simp*
next
case (*cons s c e sh ch w s' c'*) **note** *IHP=this* **show** *?case*
proof (*intro allI impI*)
fix *cn*
assume *MON*: *mon-ww fg* (*e* # *w*) ∩ *mon-c fg cn* = {}
from *ntrs-modify-context-s*[*OF IHP*(*1*)] *MON* **obtain** *csph* **where** *S1*: *ch* = *csph* + *c* *mon-c fg csph*={} ((*s*, *cn*), *e*, *sh*, *csph* + *cn*) ∈ *ntrs fg* **by** *auto*
with *MON* **have** *mon-ww fg w* ∩ *mon-c fg* (*csph*+*cn*) = {} **by** (*auto simp add*: *mon-c-unconc*)
with *IHP*(*3*)[*rule-format*] **obtain** *csp* **where** *S2*: *c'*=*csp*+*ch* *mon-c fg csp*={} ((*sh*,*csph*+*cn*),*w*,(*s'*,*csp*+(*csph*+*cn*)))∈*trcl* (*ntrs fg*) **by** *blast*
from *S1 S2* **have** *c'*=(*csp*+*csph*)+*c* *mon-c fg* (*csp*+*csph*)={} ((*s*,*cn*),*e*#*w*,(*s'*,(*csp*+*csph*)+*cn*))∈*trcl* (*ntrs fg*) **by** (*auto simp add*: *union-assoc mon-c-unconc*)
thus ∃ *csp*. *c'* = *csp* + *c* ∧ *mon-c fg csp* = {} ∧ ((*s*, *cn*), *e* # *w*, *s'*, *csp* + *cn*) ∈ *trcl* (*ntrs fg*) **by** *blast*
qed
qed

lemma *ntrs-xchange-context-s*:
assumes *A*: ((*s*,*c*),*ee*,(*s'*,*csp*+*c*))∈*ntrs fg*
and *B*: *mon-c fg cn* ⊆ *mon-c fg c*
shows ((*s*,*cn*),*ee*,(*s'*,*csp*+*cn*))∈*ntrs fg*
proof −
obtain *p r u u' v w* **where** *S*: *s*=*u*#*r* *ee*=*LCall p*#*w* *s'*=*v*#*u'*#*r* ((*u*#*r*,*c*),*LCall p*,(*entry fg p*#*u'*#*r*,*c*))∈*trss fg* (([*entry fg p*],*c*),*w*,([*v*],*csp*+*c*))∈*trcl* (*trss fg*)
proof −
from *ntrs.cases*[*OF A*, *simplified*] **obtain** *ce ce' p r u u' v w* **where** *s* = *u* # *r* *c* = *ce* *ee* = *LCall p* # *w* *s'* = *v* # *u'* # *r* *csp* + *ce* = *ce'* ((*u* # *r*, *ce*), *LCall p*, *entry fg p* # *u'* # *r*, *ce*) ∈ *trss fg*
(([*entry fg p*], *ce*), *w*, [*v*], *ce'*) ∈ *trcl* (*trss fg*) **.**
hence *s*=*u*#*r* *ee*=*LCall p*#*w* *s'*=*v*#*u'*#*r* ((*u*#*r*,*c*),*LCall p*,(*entry fg p*#*u'*#*r*,*c*))∈*trss fg* (([*entry fg p*],*c*),*w*,([*v*],*csp*+*c*))∈*trcl* (*trss fg*) **by** *auto*
then show *?thesis* **..**
qed
from *ntrs-step*[*simplified*, *OF trss-xchange-context-s*[**where** *csp*={#}, *simplified*, *OF S*(*4*) *B*] *trss-xchange-context*[*OF S*(*5*) *B*]] *S* **show** *?thesis* **by** *simp*
qed

lemma *ntrs-replace-context-s*:
assumes *A*: ((*s*,*c*+*cr*),*ee*,(*s'*,*c'*+*cr*))∈*ntrs fg*
and *B*: *mon-c fg crn* ⊆ *mon-c fg cr*
shows ((*s*,*c*+*crn*),*ee*,(*s'*,*c'*+*crn*))∈*ntrs fg*
proof −

from *ntrs-c-cases-s*[*OF A*] **obtain** *csp* **where** *G*: $c'+cr = csp+(c+cr)$ **. hence** *F*: $c'=csp+c$ **by** (*auto simp add*: *union-assoc*[*symmetric*])
from *B* **have** *MON*: *mon-c fg* $(c+crn) \subseteq$ *mon-c fg* $(c+cr)$ **by** (*auto simp add*: *mon-c-unconc*)
from *ntrs-xchange-context-s*[*OF - MON*] *A G* **have** $((s,c+crn),ee,(s',csp+(c+crn)))\in$ *ntrs fg* **by** *auto*
with *F* **show** *?thesis* **by** (*simp add*: *union-assoc*)
qed

lemma (**in** *flowgraph*) *ntrs-xchange-context*: $!!s\ c\ c'\ cn.$ $\llbracket$
$((s,c),ww,(s',c'))\in$ *trcl* (*ntrs fg*);
mon-c fg cn $\subseteq$ *mon-c fg c*
$\rrbracket \Longrightarrow \exists\, csp.$
$c'=csp+c \wedge ((s,cn),ww,(s',csp+cn))\in$ *trcl* (*ntrs fg*)
proof (*induct ww*)
case *Nil* **note** *CASE=this*
thus *?case* **by** (*auto intro*!: *exI*[*of* - {#}])
next
case (*Cons ee ww*) **note** *IHP=this*
then obtain *sh ch* **where** *SPLIT*: $((s,c),ee,(sh,ch))\in$ *ntrs fg* $((sh,ch),ww,(s',c'))\in$ *trcl* (*ntrs fg*) **by** (*fast dest*: *trcl-uncons*)
from *ntrs-c-cases-s*[*OF SPLIT*(*1*)] **obtain** *csph* **where** *CHFMT*: $ch=csph+c$ $!!s.$ $s \in\#$ *csph* $\Longrightarrow \exists\, p\ u\ v.\ s=$[*entry fg p*] $\wedge$ $(u,$ *Spawn* $p,\ v) \in$ *edges fg* $\wedge$ *initialproc fg p* **by** *blast*
with *ntrs-xchange-context-s SPLIT*(*1*) *IHP*(*3*) **have** $((s,cn),ee,(sh,csph+cn))\in$ *ntrs fg* **by** *blast*
also
from *c-of-initial-no-mon CHFMT*(*2*) **have** *CSPH-NO-MON*: *mon-c fg csph* = {} **by** *auto*
with *IHP*(*3*) *CHFMT* **have** *1*: *mon-c fg* $(csph+cn) \subseteq$ *mon-c fg ch* **by** (*auto simp add*: *mon-c-unconc*)
from *IHP*(*1*)[*OF SPLIT*(*2*) *this*] **obtain** *csp* **where** *C'FMT*: $c'=csp+ch$ **and** *SND*: $((sh,csph+cn),ww,(s',csp+(csph+cn)))\in$ *trcl* (*ntrs fg*) **by** *blast*
note *SND*
finally have $((s,\ cn),\ ee\ \#\ ww,\ s',\ (csp\ +\ csph)\ +\ cn) \in$ *trcl* (*ntrs fg*) **by** (*simp add*: *union-assoc*)
moreover from *CHFMT*(*1*) *C'FMT* **have** $c'=(csp+csph)+c$ **by** (*simp add*: *union-assoc*)
ultimately show *?case* **by** *blast*
qed

lemma (**in** *flowgraph*) *ntrs-replace-context*:
assumes *A*: $((s,c+cr),ww,(s',c'+cr))\in$ *trcl* (*ntrs fg*)
and *B*: *mon-c fg crn* $\subseteq$ *mon-c fg cr*
shows $((s,c+crn),ww,(s',c'+crn))\in$ *trcl* (*ntrs fg*)
proof −
from *ntrs-c-cases*[*OF A*] **obtain** *csp* **where** *G*: $c'+cr = csp+(c+cr)$ **. hence** *F*: $c'=csp+c$ **by** (*auto simp add*: *union-assoc*[*symmetric*])

from B **have** MON: $mon\text{-}c\ fg\ (c+crn) \subseteq mon\text{-}c\ fg\ (c+cr)$ **by** (*auto simp add*: *mon-c-unconc*)
from *ntrs-xchange-context*[*OF A MON*] G **have** $((s,c+crn),ww,(s',csp+(c+crn)))\in trcl$ $(ntrs\ fg)$ **by** *auto*
with F **show** *?thesis* **by** (*simp add*: *union-assoc*)
qed

lemma (**in** *flowgraph*) *ntr-add-context-s*:
assumes A: $(c,e,c')\in ntr\ fg$
and B: $mon\text{-}w\ fg\ e \cap mon\text{-}c\ fg\ cn = \{\}$
shows $(c+cn,e,c'+cn)\in ntr\ fg$
proof −
from *gtrE*[*OF A*] **obtain** $s\ ce\ s'\ ce'$ **where** $NTRS$: $c = add\text{-}mset\ s\ ce\ c' = add\text{-}mset\ s'\ ce'\ ((s,\ ce),\ e,\ s',\ ce') \in ntrs\ fg$ **.**
from *ntrs-mon-e-no-ctx*[*OF NTRS(3)*] B **have** M: $mon\text{-}w\ fg\ e \cap (mon\text{-}c\ fg\ (ce+cn)) = \{\}$ **by** (*auto simp add*: *mon-c-unconc*)
from *ntrs-modify-context-s*[*OF NTRS(3) M*] **have** $((s,ce+cn),e,(s',ce'+cn))\in ntrs\ fg$ **by** (*auto simp add*: *union-assoc*)
with $NTRS$ **show** *?thesis* **by** (*auto simp add*: *union-assoc intro*: *gtrI-s*)
qed

lemma (**in** *flowgraph*) *ntr-add-context*:
$[\![(c,w,c')\in trcl\ (ntr\ fg);\ mon\text{-}ww\ fg\ w \cap mon\text{-}c\ fg\ cn = \{\}]\!]$
$\Longrightarrow (c+cn,w,c'+cn)\in trcl\ (ntr\ fg)$
by (*induct rule*: *trcl.induct*) (*simp*, *force dest*: *ntr-add-context-s*)

lemma (**in** *flowgraph*) *ntrs-add-context-s*:
assumes A: $((s,c),e,(s',c'))\in ntrs\ fg$
and B: $mon\text{-}w\ fg\ e \cap mon\text{-}c\ fg\ cn = \{\}$
shows $((s,c+cn),e,(s',c'+cn))\in ntrs\ fg$
using *ntrs-mon-e-no-ctx*[*OF A*] *ntrs-modify-context-s*[*OF A*, *of c+cn*] B **by** (*force simp add*: *mon-c-unconc union-ac*)

lemma (**in** *flowgraph*) *ntrp-add-context-s*:
$[\![\ ((s,c),e,(s',c'))\in ntrp\ fg;\ mon\text{-}w\ fg\ (le\text{-}rem\text{-}s\ e) \cap mon\text{-}c\ fg\ cn = \{\}\ ]\!]$
$\Longrightarrow ((s,c+cn),e,(s',c'+cn))\in ntrp\ fg$
apply (*erule gtrp.cases*)
by (*auto dest*: *ntrs-add-context-s intro*!: *gtrp.intros*)

lemma (**in** *flowgraph*) *ntrp-add-context*: $[\![$
$((s,c),w,(s',c'))\in trcl\ (ntrp\ fg)$;
$mon\text{-}ww\ fg\ (map\ le\text{-}rem\text{-}s\ w) \cap mon\text{-}c\ fg\ cn = \{\}$
$]\!] \Longrightarrow ((s,c+cn),w,(s',c'+cn))\in trcl\ (ntrp\ fg)$
by (*induct rule*: *trcl-pair-induct*) (*simp*, *force dest*: *ntrp-add-context-s*)

8.4.4 Altering the local stack

lemma *ntrs-stack-comp-s*:

assumes *A*: ((s,c),ee,(s′,c′))∈*ntrs fg*
shows ((s@r,c),ee,(s′@r,c′))∈*ntrs fg*
using *A*
by (*auto dest*: *trss-stack-comp trss-stack-comp-s elim*!: *ntrs.cases intro*!: *ntrs-step*[*simplified*])

lemma *ntrs-stack-comp*: ((s,c),ww,(s′,c′))∈*trcl* (*ntrs fg*)
⟹ ((s@r,c),ww,(s′@r,c′))∈*trcl* (*ntrs fg*)
by (*induct rule*: *trcl-pair-induct*) (*auto intro*!: *trcl.cons*[*OF ntrs-stack-comp-s*])

lemma (**in** *flowgraph*) *ntrp-stack-comp-s*:
assumes *A*: ((s,c),ee,(s′,c′))∈*ntrp fg*
and *B*: *mon-s fg r* ∩ *mon-env fg* [*ee*] = {}
shows ((s@r,c),ee,(s′@r,c′))∈*ntrp fg*
using *A*
proof (*cases rule*: *gtrp.cases*)
case *gtrp-loc* **then obtain** *e* **where** *CASE*: *ee=LOC e* ((s,c),e,(s′,c′))∈*ntrs fg*
by *auto*
hence ((s@r,c),e,(s′@r,c′))∈*ntrs fg* **by** (*blast dest*: *ntrs-stack-comp-s*)
with *CASE*(*1*) **show** *?thesis* **by** (*auto intro*: *gtrp.gtrp-loc*)
next
case *gtrp-env* **then obtain** *sm ce sm′ ce′ e* **where** *CASE*: *s′=s c*={#*sm*#}+*ce*
c′={#*sm′*#}+*ce′ ee=ENV e* ((*sm*,{#*s*#}+*ce*),*e*,(*sm′*,{#*s*#}+*ce′*))∈*ntrs fg* **by** *auto*
from *ntrs-modify-context-s*[*OF CASE*(*5*), **where** *cn*={#*s*@*r*#}+*ce*] *ntrs-mon-e-no-ctx*[*OF CASE*(*5*)] *B CASE*(*4*) **obtain** *csp* **where**
ADD: {#*s*#} + *ce′* = *csp* + ({#*s*#} + *ce*) *mon-c fg csp* = {} ((*sm*, {#*s* @ *r*#} + *ce*), *e*, *sm′*, *csp* + ({#*s* @ *r*#} + *ce*)) ∈ *ntrs fg* **by** (*auto simp add*: *mon-c-unconc mon-s-unconc*)
moreover from *ADD*(*1*) **have** {#*s*#}+*ce′*={#*s*#}+(*csp*+*ce*) **by** (*simp add*: *union-ac*) **hence** *ce′*=*csp*+*ce* **by** *simp*
ultimately have ((*sm*, {#*s* @ *r*#} + *ce*), *e*, *sm′*, ({#*s* @ *r*#} + *ce′*)) ∈ *ntrs fg*
by (*simp add*: *union-ac*)
with *CASE*(*1*,*2*,*3*,*4*) **show** *?thesis* **by** (*auto intro*: *gtrp.gtrp-env*)
qed

lemma (**in** *flowgraph*) *ntrp-stack-comp*:
⟦ ((s,c),ww,(s′,c′))∈*trcl* (*ntrp fg*); *mon-s fg r* ∩ *mon-env fg ww* = {} ⟧
⟹ ((s@r,c),ww,(s′@r,c′))∈*trcl* (*ntrp fg*)
by (*induct rule*: *trcl-pair-induct*) (*auto intro*!: *trcl.cons*[*OF ntrp-stack-comp-s*])

lemma *ntrs-stack-top-decomp-s*:
assumes *A*: ((u#r,c),ee,(s′,c′))∈*ntrs fg*
and *EX*: !!*v u′ p*. ⟦
s′=v#u′#r;
(([u],c),ee,([v,u′],c′))∈*ntrs fg*;
(u,*Call p*,u′)∈*edges fg*
⟧ ⟹ *P*
shows *P*
using *A*
proof (*cases rule*: *ntrs.cases*)

case *ntrs-step* **then obtain** *u′ v p w* **where** *CASE*: *ee=LCall p#w s′=v#u′#r* ((*u#r,c*),*LCall p*,(*entry fg p#u′#r,c*))∈*trss fg* (([*entry fg p*],*c*),*w*,([*v*],*c′*))∈*trcl* (*trss fg*) **by** (*simp*)
from *trss-stack-decomp-s*[**where** *s*=[*u*], *simplified*, *OF CASE*(*3*)] **have** *SDC*: (([*u*], *c*), *LCall p*, ([*entry fg p*, *u′*], *c*)) ∈ *trss fg* **by** *auto*
with *CASE*(*1*,*4*) **have** (([*u*],*c*),*ee*,([*v*,*u′*],*c′*))∈*ntrs fg* **by** (*auto intro*!: *ntrs.ntrs-step*)
moreover from *SDC* **have** (*u*,*Call p*,*u′*)∈*edges fg* **by** (*auto elim*!: *trss.cases*)
ultimately show *?thesis* **using** *CASE*(*2*) **by** (*blast intro*!: *EX*)
qed

lemma *ntrs-stack-decomp-s*:
assumes *A*: ((*u#s@r,c*),*ee*,(*s′,c′*))∈*ntrs fg*
and *EX*: !!*v u′ p*. ⟦
s′=v#u′#s@r;
((*u#s,c*),*ee*,(*v#u′#s,c′*))∈*ntrs fg*;
(*u*,*Call p*,*u′*)∈*edges fg*
⟧ ⟹ *P*
shows *P*
apply (*rule ntrs-stack-top-decomp-s*[*OF A*])
apply (*rule EX*)
apply (*auto dest*: *ntrs-stack-comp-s*)
done

lemma *ntrs-stack-decomp*: !!*u s r c P*. ⟦
((*u#s@r,c*),*ww*,(*s′,c′*))∈*trcl* (*ntrs fg*);
!!*v rr*. ⟦*s′=v#rr@r*; ((*u#s,c*),*ww*,(*v#rr,c′*))∈*trcl* (*ntrs fg*)⟧ ⟹ *P*
⟧ ⟹ *P*
proof (*induct ww*)
case *Nil* **thus** *?case* **by** *fastforce*
next
case (*Cons e w*) **from** *Cons.prems* **show** *?case* **proof** (*cases rule*: *trcl-pair-unconsE*)
case (*split sh ch*)
from *ntrs-stack-decomp-s*[*OF split*(*1*)] **obtain** *vh uh p* **where** *F*: *sh = vh#uh#s@r* ((*u#s*, *c*), *e*, *vh#uh#s*, *ch*) ∈ *ntrs fg* (*u*, *Call p*, *uh*) ∈ *edges fg* **by** *blast*
from *F*(*1*) *split*(*2*) *Cons.hyps*[*of vh uh#s r ch*] **obtain** *v′ rr* **where** *S*: *s′=v′#rr@r* ((*vh#uh#s,ch*),*w*,(*v′#rr,c′*))∈*trcl* (*ntrs fg*) **by** *auto*
from *trcl.cons*[*OF F*(*2*) *S*(*2*)] *S*(*1*) *Cons.prems*(*2*) **show** *?thesis* **by** *blast*
qed
qed

lemma *ntrp-stack-decomp-s*:
assumes *A*: ((*u#s@r,c*),*ee*,(*s′,c′*))∈*ntrp fg*
and *EX*: !!*v rr*. ⟦ *s′=v#rr@r*; ((*u#s,c*),*ee*,(*v#rr,c′*))∈*ntrp fg* ⟧ ⟹ *P*
shows *P*
using *A*
proof (*cases rule*: *gtrp.cases*)
case *gtrp-loc* **thus** *?thesis* **using** *EX* **by** (*force elim*!: *ntrs-stack-decomp-s intro*!: *gtrp.intros*)
next

case *gtrp-env* **then obtain** *e ss ss′ ce ce′* **where** *S*: *ee=ENV e s′=u#s@r c={#ss#}+ce c′={#ss′#}+ce′ ((ss,ce+{#u#s@r#}),e,(ss′,ce′+{#u#s@r#}))∈ntrs fg* **by** (*auto simp add*: *union-ac*)
from *ntrs-replace-context-s*[*OF S*(*5*), **where** *crn*={#*u*#*s*#}] **have** ((*ss*, {#*u* # *s*#} + *ce*), *e*, *ss′*, {#*u* # *s*#} + *ce′*) ∈ *ntrs fg* **by** (*auto simp add*: *mon-s-unconc union-ac*)
with *S* **show** *P* **by** (*rule-tac EX*) (*auto intro*: *gtrp.gtrp-env*)
qed

lemma *ntrp-stack-decomp*: !!*u s r c P*. ⟦
((*u*#*s*@*r*,*c*),*ww*,(*s′*,*c′*))∈*trcl* (*ntrp fg*);
!!*v rr*. ⟦*s′*=*v*#*rr*@*r*; ((*u*#*s*,*c*),*ww*,(*v*#*rr*,*c′*))∈*trcl* (*ntrp fg*)⟧ ⟹ *P*
⟧ ⟹ *P*
proof (*induct ww*)
case *Nil* **thus** *?case* **by** *fastforce*
next
case (*Cons e w*) **from** *Cons.prems* **show** *?case* **proof** (*cases rule*: *trcl-pair-unconsE*)
case (*split sh ch*)
from *ntrp-stack-decomp-s*[*OF split*(*1*)] **obtain** *vh rrh* **where** *F*: *sh* = *vh*#*rrh*@*r* ((*u*#*s*, *c*), *e*, *vh*#*rrh*, *ch*) ∈ *ntrp fg* **by** *blast*
from *F*(*1*) *split*(*2*) *Cons.hyps*[*of vh rrh r ch*] **obtain** *v′ rr* **where** *S*: *s′*=*v′*#*rr*@*r* ((*vh*#*rrh*,*ch*),*w*,(*v′*#*rr*,*c′*))∈*trcl* (*ntrp fg*) **by** *auto*
from *trcl.cons*[*OF F*(*2*) *S*(*2*)] *S*(*1*) *Cons.prems*(*2*) **show** *?thesis* **by** *blast*
qed
qed

8.5 Relation to monitor consistent interleaving

In this section, we describe the relation of the consistent interleaving operator (cf. Section 2) and the macrostep-semantics.

8.5.1 Abstraction function for normalized paths

We first need to define an abstraction function that maps a macrostep on a pair of entered and passed monitors, as required by the $\otimes_\alpha$-operator:

A step on a normalized paths enters the monitors of the first called procedure and passes the monitors that occur in the following same-level path.

definition
αn fg e == *if e*=[] *then* ({},{}) *else* (*mon-e fg* (*hd e*), *mon-w fg* (*tl e*))

lemma *αn-simps*[*simp*]:
αn fg [] = ({},{})
αn fg (*e*#*w*) = (*mon-e fg e*, *mon-w fg w*)
by (*unfold αn-def*, *auto*)

— We also need an abstraction function for normalized loc/env-paths
definition

$\alpha nl\ fg\ e == \alpha n\ fg\ (le\text{-}rem\text{-}s\ e)$

lemma *αnl-def′*: $\alpha nl\ fg == \alpha n\ fg \circ le\text{-}rem\text{-}s$
by (*rule eq-reflection*[*OF ext*]) (*auto simp add*: *αnl-def*)

— These are some ad-hoc simplifications, with the aim at converting *αnl* back to *αn*
lemma *αnl-simps*[*simp*]:
$\alpha nl\ fg\ (ENV\ x) = \alpha n\ fg\ x$
$\alpha nl\ fg\ (LOC\ x) = \alpha n\ fg\ x$
by (*unfold αnl-def*, *auto*)
lemma *αnl-simps1*[*simp*]:
$(\alpha nl\ fg) \circ ENV = \alpha n\ fg$
$(\alpha nl\ fg) \circ LOC = \alpha n\ fg$
by (*unfold αnl-def′ comp-def*) (*simp-all*)
lemma *αn-αnl*: $(\alpha n\ fg) \circ le\text{-}rem\text{-}s = \alpha nl\ fg$
unfolding *αnl-def′*[*symmetric*] **..**
lemma *αn-fst-snd*[*simp*]: $fst\ (\alpha n\ fg\ w) \cup snd\ (\alpha n\ fg\ w) = mon\text{-}w\ fg\ w$
by (*induct w*) *auto*

lemma *mon-pl-of-αnl*: $mon\text{-}pl\ (map\ (\alpha nl\ fg)\ w) = mon\text{-}loc\ fg\ w \cup mon\text{-}env\ fg\ w$
by (*induct w*) (*auto split*: *el-step.split*)

We now derive specialized introduction lemmas for $\otimes_{\alpha n\ fg}$

lemma *cil-αn-cons-helper*: $mon\text{-}pl\ (map\ (\alpha n\ fg)\ wb) = mon\text{-}ww\ fg\ wb$
apply (*unfold mon-pl-def*)
apply (*induct wb*)
apply *simp-all*
apply (*unfold mon-ww-def*)
apply (*subst foldl-un-empty-eq*)
apply (*case-tac a*)
apply *simp-all*
done

lemma *cil-αnl-cons-helper*:
$mon\text{-}pl\ (map\ (\alpha nl\ fg)\ wb) = mon\text{-}ww\ fg\ (map\ le\text{-}rem\text{-}s\ wb)$
by (*simp add*: *αn-αnl cil-αn-cons-helper*[*symmetric*])

lemma *cil-αn-cons1*: $[\![w \in wa \otimes_{\alpha n\ fg} wb;\ fst\ (\alpha n\ fg\ e) \cap mon\text{-}ww\ fg\ wb = \{\}]\!]$
$\Longrightarrow e\#w \in e\#wa \otimes_{\alpha n\ fg} wb$
apply (*rule cil-cons1*)
apply *assumption*
apply (*subst cil-αn-cons-helper*)
apply *assumption*
done

lemma *cil-αn-cons2*: $[\![w \in wa \otimes_{\alpha n\ fg} wb;\ fst\ (\alpha n\ fg\ e) \cap mon\text{-}ww\ fg\ wa = \{\}]\!]$
$\Longrightarrow e\#w \in wa \otimes_{\alpha n\ fg} e\#wb$
apply (*rule cil-cons2*)

apply *assumption*
apply (*subst cil-αn-cons-helper*)
apply *assumption*
done

8.5.2 Monitors

lemma (**in** *flowgraph*) *ntrs-mon-s*:
assumes *A*: $((s,c),e,(s',c'))\in ntrs\ fg$
shows $mon\text{-}s\ fg\ s' = mon\text{-}s\ fg\ s \cup fst\ (\alpha n\ fg\ e)$
proof –
from *A* **obtain** $u\ r\ p\ u'\ w\ v$ **where** *DET*: $s{=}u\#r$ $e{=}LCall\ p\#w$ $((u\#r,c),LCall\ p,(entry\ fg\ p\#u'\#r,c))\in trss\ fg$ $(([entry\ fg\ p],c),w,([v],c'))\in trcl\ (trss\ fg)$ $s'{=}v\#u'\#r$
by (*blast elim*!: *ntrs.cases*[*simplified*])
hence $mon\text{-}n\ fg\ u = mon\text{-}n\ fg\ u'$ **by** (*auto elim*!: *trss.cases dest*: *mon-n-same-proc edges-part*)
with *trss-bot-proc-const*[**where** $s{=}[]$ **and** $s'{=}[]$, *simplified*, *OF DET*(*4*)] *DET*(*1*,*2*,*5*)
show *?thesis* **by** (*auto simp add*: *mon-n-def αn-def*)
qed

corollary (**in** *flowgraph*) *ntrs-called-mon*:
assumes *A*: $((s,c),e,(s',c'))\in ntrs\ fg$
shows $fst\ (\alpha n\ fg\ e) \subseteq mon\text{-}s\ fg\ s'$
using *ntrs-mon-s*[*OF A*] **by** *auto*

lemma (**in** *flowgraph*) *ntr-mon-s*:
$(c,e,c')\in ntr\ fg \Longrightarrow mon\text{-}c\ fg\ c' = mon\text{-}c\ fg\ c \cup fst\ (\alpha n\ fg\ e)$
by (*erule gtrE*) (*auto simp add*: *mon-c-unconc ntrs-c-no-mon-s ntrs-mon-s*)

lemma (**in** *flowgraph*) *ntrp-mon-s*:
assumes *A*: $((s,c),e,(s',c'))\in ntrp\ fg$
shows $mon\text{-}c\ fg\ (add\text{-}mset\ s'\ c') = mon\text{-}c\ fg\ (add\text{-}mset\ s\ c) \cup fst\ (\alpha nl\ fg\ e)$
using *ntr-mon-s*[*OF gtrp2gtr-s*[*OF A*]] **by** (*unfold αnl-def*)

8.5.3 Interleaving theorem

In this section, we show that the consistent interleaving operator describes the intuition behind interleavability of normalized paths. We show: *Two paths are simultaneously executable if and only if they are consistently interleavable and the monitors of the initial configurations are compatible*

The split lemma splits an execution from a context of the form $ca + cb$ into two interleavable executions from ca and cb respectively. While further down we prove this lemma for loc/env-path, which is more general but also more complicated, we start with the proof for paths of the multiset-semantics for illustrating the idea.

lemma (**in** *flowgraph*) *ntr-split*:
$!!ca\ cb.\ [\![(ca{+}cb,w,c')\in trcl\ (ntr\ fg);\ valid\ fg\ (ca{+}cb)]\!] \Longrightarrow$

$\exists$ *ca′ cb′ wa wb.*
c′=ca′+cb′ $\wedge$
w$\in$(*wa*$\otimes_{\alpha n\ fg}$*wb*) $\wedge$
mon-c fg ca $\cap$ (*mon-c fg cb* $\cup$ *mon-ww fg wb*) = {} $\wedge$
mon-c fg cb $\cap$ (*mon-c fg ca* $\cup$ *mon-ww fg wa*) = {} $\wedge$
(*ca*,*wa*,*ca′*)$\in$*trcl* (*ntr fg*) $\wedge$ (*cb*,*wb*,*cb′*)$\in$*trcl* (*ntr fg*)

proof (*induct w*) — The proof is done by induction on the path

— If the path is empty, the lemma is trivial

case *Nil* **thus** *?case* **by** − (*rule exI*[*of* - *ca*], *rule exI*[*of* - *cb*], *intro exI*[*of* - []], *auto simp add*: *valid-unconc*)

next

case (*Cons e w*) **note** *IHP=this*

— We split a non-empty paths after the first (macro) step

then obtain *ch* **where** *SPLIT*: (*ca+cb*,*e*,*ch*)$\in$*ntr fg* (*ch*,*w*,*c′*)$\in$*trcl* (*ntr fg*) **by** (*fast dest*: *trcl-uncons*)

— Pick the stack that made the first step

from *gtrE*[*OF SPLIT*(*1*)] **obtain** *s ce sh ceh* **where** *NTRS*: *ca+cb=add-mset s ce ch=add-mset sh ceh* ((*s*,*ce*),*e*,(*sh*,*ceh*))$\in$*ntrs fg* **.**

— And separate the threads that where spawned during the first step from the ones that where already there

then obtain *csp* **where** *CEHFMT*: *ceh=csp+ce mon-c fg csp*={} **by** (*auto elim*!: *ntrs-c-cases-s intro*!: *c-of-initial-no-mon*)

— Needed later: The first macrostep uses no monitors already owned by threads that where already there

from *ntrs-mon-e-no-ctx*[*OF NTRS*(*3*)] **have** *MONED*: *mon-w fg e* $\cap$ *mon-c fg ce* = {} **by** (*auto simp add*: *mon-c-unconc*)

— Needed later: The intermediate configuration is valid

from *ntr-valid-preserve-s*[*OF SPLIT*(*1*) *IHP*(*3*)] **have** *CHVALID*: *valid fg ch* **.**

— We make a case distinction whether the thread that made the first step was in the left or right part of the initial configuration

from *NTRS*(*1*)[*symmetric*] **show** *?case* **proof** (*cases rule*: *mset-unplusm-dist-cases*)

— The first step was on a thread in the left part of the initial configuration

case *left* **note** *CASE=this*

— We can write the intermediate configuration so that it is suited for the induction hypothesis

with *CEHFMT NTRS* **have** *CHFMT*: *ch*=({#*sh*#}+*csp*+(*ca*−{#*s*#}))+*cb* **by** (*simp add*: *union-ac*)

— and by the induction hypothesis, we split the path from the intermediate configuration

with *IHP*(*1*) *SPLIT*(*2*) *CHVALID* **obtain** *ca′ cb′ wa wb* **where** *IHAPP*:
c′=ca′+cb′
w$\in$*wa*$\otimes_{\alpha n\ fg}$*wb*
mon-c fg ({#*sh*#}+*csp*+(*ca*−{#*s*#})) $\cap$ (*mon-c fg cb* $\cup$ *mon-ww fg wb*)={}
mon-c fg cb $\cap$ (*mon-c fg* ({#*sh*#}+*csp*+(*ca*−{#*s*#})) $\cup$ *mon-ww fg wa*)={}
({#*sh*#}+*csp*+(*ca*−{#*s*#}),*wa*,*ca′*)$\in$*trcl* (*ntr fg*)
(*cb*,*wb*,*cb′*)$\in$*trcl* (*ntr fg*)

by *blast*
moreover
— It remains to show that we can execute the first step with the right part of the configuration removed
have *FIRSTSTEP*: (*ca*,*e*,{#*sh*#}+*csp*+(*ca*−{#*s*#}))∈*ntr fg*
proof −
from *CASE*(*2*) **have** *mon-c fg* (*ca*−{#*s*#}) ⊆ *mon-c fg ce* **by** (*auto simp add*: *mon-c-unconc*)
with *ntrs-xchange-context-s NTRS*(*3*) *CEHFMT CASE*(*2*) **have** ((*s*,*ca*−{#*s*#}),*e*,(*sh*,*csp*+(*ca*−{#*s*#})))∈ *fg* **by** *blast*
from *gtrI-s*[*OF this*] *CASE*(*1*) **show** *?thesis* **by** (*auto simp add*: *union-assoc*)
qed
with *IHAPP*(*5*) **have** (*ca*,*e*#*wa*,*ca*′)∈*trcl* (*ntr fg*) **by** *simp*
moreover
— and that we can prepend the first step to the interleaving
have $e\#w \in e\#wa \otimes_{\alpha n\ fg} wb$
proof −
from *ntrs-called-mon*[*OF NTRS*(*3*)] **have** *fst* (α*n fg e*) ⊆ *mon-s fg sh* **.**
with *IHAPP*(*3*) **have** *fst* (α*n fg e*) ∩ *mon-ww fg wb* = {} **by** (*auto simp add*: *mon-c-unconc*)
from *cil-*α*n-cons1*[*OF IHAPP*(*2*) *this*] **show** *?thesis* **.**
qed
moreover
— and that the monitors of the initial context does not interfere
have *mon-c fg ca* ∩ (*mon-c fg cb* ∪ *mon-ww fg wb*) = {} *mon-c fg cb* ∩ (*mon-c fg ca* ∪ *mon-ww fg* (*e*#*wa*)) = {}
proof −
from *ntr-mon-increasing-s*[*OF FIRSTSTEP*] *IHAPP*(*3*) **show** *mon-c fg ca* ∩ (*mon-c fg cb* ∪ *mon-ww fg wb*) = {} **by** *auto*
from *MONED CASE* **have** *mon-c fg cb* ∩ *mon-w fg e* = {} **by** (*auto simp add*: *mon-c-unconc*)
with *ntr-mon-increasing-s*[*OF FIRSTSTEP*] *IHAPP*(*4*) **show** *mon-c fg cb* ∩ (*mon-c fg ca* ∪ *mon-ww fg* (*e*#*wa*)) = {} **by** *auto*
qed
ultimately show *?thesis* **by** *blast*
next
— The other case, that is if the first step was made on a thread in the right part of the configuration, is shown completely analogously
case *right* **note** *CASE*=*this*
with *CEHFMT NTRS* **have** *CHFMT*: *ch*=*ca*+({#*sh*#}+*csp*+(*cb*−{#*s*#})) **by** (*simp add*: *union-ac*)
with *IHP*(*1*) *SPLIT*(*2*) *CHVALID* **obtain** *ca*′ *cb*′ *wa wb* **where** *IHAPP*: *c*′=*ca*′+*cb*′ $w \in wa \otimes_{\alpha n\ fg} wb$ *mon-c fg ca* ∩ (*mon-c fg* ({#*sh*#}+*csp*+(*cb*−{#*s*#})) ∪ *mon-ww fg wb*)={}
mon-c fg ({#*sh*#}+*csp*+(*cb*−{#*s*#})) ∩ (*mon-c fg ca* ∪ *mon-ww fg wa*)={} (*ca*,*wa*,*ca*′)∈*trcl* (*ntr fg*) ({#*sh*#}+*csp*+(*cb*−{#*s*#}),*wb*,*cb*′)∈*trcl* (*ntr fg*)
by *blast*
moreover
have *FIRSTSTEP*: (*cb*,*e*,{#*sh*#}+*csp*+(*cb*−{#*s*#}))∈*ntr fg* **proof** −

from *CASE*(*2*) **have** *mon-c fg* (*cb*$-${#*s*#}) $\subseteq$ *mon-c fg ce* **by** (*auto simp add*: *mon-c-unconc*)
with *ntrs-xchange-context-s NTRS*(*3*) *CEHFMT CASE*(*2*) **have** ((*s*,*cb*$-${#*s*#}),*e*,(*sh*,*csp*+(*cb*$-${#*s*#})))$\in$
fg **by** *blast*
from *gtrI-s*[*OF this*] *CASE*(*1*) **show** *?thesis* **by** (*auto simp add*: *union-assoc*)
qed
with *IHAPP*(*6*) **have** *PA*: (*cb*,*e*#*wb*,*cb*$'$)$\in$*trcl* (*ntr fg*) **by** *simp*
moreover
have *e*#*w* $\in$ *wa* $\otimes_{\alpha n\ fg}$ *e*#*wb*
proof $-$
from *ntrs-called-mon*[*OF NTRS*(*3*)] **have** *fst* (αn *fg e*) $\subseteq$ *mon-s fg sh* **.**
with *IHAPP*(*4*) **have** *fst* (αn *fg e*) $\cap$ *mon-ww fg wa* = {} **by** (*auto simp add*: *mon-c-unconc*)
from *cil-*α*n-cons2*[*OF IHAPP*(*2*) *this*] **show** *?thesis* **.**
qed
moreover
have *mon-c fg cb* $\cap$ (*mon-c fg ca* $\cup$ *mon-ww fg wa*) = {} *mon-c fg ca* $\cap$ (*mon-c fg cb* $\cup$ *mon-ww fg* (*e*#*wb*)) = {}
proof $-$
from *ntr-mon-increasing-s*[*OF FIRSTSTEP*] *IHAPP*(*4*) **show** *mon-c fg cb* $\cap$ (*mon-c fg ca* $\cup$ *mon-ww fg wa*) = {} **by** *auto*
from *MONED CASE* **have** *mon-c fg ca* $\cap$ *mon-w fg e* = {} **by** (*auto simp add*: *mon-c-unconc*)
with *ntr-mon-increasing-s*[*OF FIRSTSTEP*] *IHAPP*(*3*) **show** *mon-c fg ca* $\cap$ (*mon-c fg cb* $\cup$ *mon-ww fg* (*e*#*wb*)) = {} **by** *auto*
qed
ultimately show *?thesis* **by** *blast*
qed
qed

The next lemma is a more general version of *flowgraph.ntr-split* for the semantics with a distinguished local thread. The proof follows exactly the same ideas, but is more complex.

lemma (**in** *flowgraph*) *ntrp-split*:
!!*s c1 c2 s*$'$ *c*$'$.
$[\![$((*s*,*c1*+*c2*),*w*,(*s*$'$,*c*$'$))$\in$*trcl* (*ntrp fg*); *valid fg* ({#*s*#}+*c1*+*c2*)$]\!]$
$\Longrightarrow$ $\exists$ *w1 w2 c1*$'$ *c2*$'$.
w $\in$ *w1* $\otimes_{\alpha nl\ fg}$ (*map ENV w2*) $\wedge$
c$'$=*c1*$'$+*c2*$'$ $\wedge$
((*s*,*c1*),*w1*,(*s*$'$,*c1*$'$))$\in$*trcl* (*ntrp fg*) $\wedge$
(*c2*,*w2*,*c2*$'$)$\in$*trcl* (*ntr fg*) $\wedge$
mon-ww fg (*map le-rem-s w1*) $\cap$ *mon-c fg c2* = {} $\wedge$
mon-ww fg w2 $\cap$ *mon-c fg* ({#*s*#}+*c1*) = {}
proof (*induct w*)
case *Nil* **thus** *?case* **by** (*auto intro*: *exI*[*of* - []] *exI*[*of* - {#}])
next
case (*Cons ee w*) **then obtain** *sh ch* **where** *SPLIT*: ((*s*,*c1*+*c2*),*ee*,(*sh*,*ch*))$\in$*ntrp fg* ((*sh*,*ch*),*w*,(*s*$'$,*c*$'$))$\in$*trcl* (*ntrp fg*) **by** (*fast dest*: *trcl-uncons*)
from *SPLIT*(*1*) **show** *?case* **proof** (*cases rule*: *gtrp.cases*)

case *gtrp-loc* **then obtain** *e* **where** *CASE*: $ee=LOC\ e\ ((s,c1+c2),e,(sh,ch))\in ntrs$ *fg* **by** *auto*
from *ntrs-c-cases-s*[*OF CASE*(*2*)] **obtain** *csp* **where** *CHFMT*: $ch=(csp+c1)+c2$ $\bigwedge s.\ s \in\#\ csp \Longrightarrow \exists\, p\ u\ v.\ s = [entry\ fg\ p] \wedge (u,\ Spawn\ p,\ v) \in edges\ fg \wedge initialproc$ $fg\ p$ **by** (*simp add*: *union-assoc*, *blast*)
with *c-of-initial-no-mon* **have** *CSPNOMON*: $mon\text{-}c\ fg\ csp = \{\}$ **by** *auto*
from *ntr-valid-preserve-s*[*OF gtrI-s*, *OF CASE*(*2*)] *Cons.prems*(*2*) *CHFMT* **have** *VALID*: $valid\ fg\ (\{\#sh\#\}+(csp+c1)+c2)$ **by** (*simp add*: *union-ac*)
from *Cons.hyps*[*OF - VALID*, *of s′ c′*] *CHFMT*(*1*) *SPLIT*(*2*) **obtain** *w1 w2* *c1′ c2′* **where** *IHAPP*: $w \in w1 \otimes_{\alpha nl\ fg} (map\ ENV\ w2)$ $c' = c1' + c2'$ $((sh,\ csp + c1),\ w1,\ s',\ c1') \in trcl\ (ntrp\ fg)$
$(c2,\ w2,\ c2') \in trcl\ (ntr\ fg)$ $mon\text{-}ww\ fg\ (map\ le\text{-}rem\text{-}s\ w1) \cap mon\text{-}c\ fg\ c2 = \{\}$ $mon\text{-}ww\ fg\ w2 \cap mon\text{-}c\ fg\ (\{\#sh\#\} + (csp + c1)) = \{\}$ **by** *blast*
have $ee\#w \in ee\#w1 \otimes_{\alpha nl\ fg} (map\ ENV\ w2)$ **proof** (*rule cil-cons1*)
from *ntrp-mon-env-w-no-ctx*[*OF SPLIT*(*2*), *unfolded mon-env-def*] **have** $mon\text{-}ww\ fg\ (map\ le\text{-}rem\text{-}s\ (env\ w)) \cap mon\text{-}s\ fg\ sh = \{\}$ **.**
moreover have $mon\text{-}ww\ fg\ w2 \subseteq mon\text{-}ww\ fg\ (map\ le\text{-}rem\text{-}s\ (env\ w))$ **proof** −
from *cil-subset-il IHAPP*(*1*) *ileq-interleave* **have** $map\ ENV\ w2 \preceq w$ **by** *blast*
from *le-list-filter*[*OF this*] **have** $env\ (map\ ENV\ w2) \preceq env\ w$ **by** (*unfold env-def*) *blast*
hence $map\ ENV\ w2 \preceq env\ w$ **by** (*unfold env-def*) *simp*
from *le-list-map*[*OF this*, *of le-rem-s*] **have** $w2 \preceq map\ le\text{-}rem\text{-}s\ (env\ w)$ **by** *simp*
thus *?thesis* **by** (*rule mon-ww-ileq*)
qed
ultimately have $mon\text{-}ww\ fg\ w2 \cap mon\text{-}s\ fg\ sh = \{\}$ **by** *blast*
with *ntrs-mon-s*[*OF CASE*(*2*)] *CASE*(*1*) **show** $fst\ (\alpha nl\ fg\ ee) \cap mon\text{-}pl\ (map\ (\alpha nl\ fg)\ (map\ ENV\ w2)) = \{\}$ **by** (*auto simp add*: *cil-αn-cons-helper*)
qed (*rule IHAPP*(*1*))
moreover
have $((s,c1),ee\#w1,(s',c1'))\in trcl\ (ntrp\ fg)$ **proof** −
from *ntrs-xchange-context-s*[*of s c1+c2 e sh csp fg c1*] *CASE*(*2*) *CHFMT*(*1*) **have** $((s,\ c1),\ e,\ sh,\ csp + c1) \in ntrs\ fg$ **by** (*auto simp add*: *mon-c-unconc union-ac*)
with *CASE*(*1*) **have** $((s,\ c1),\ ee,\ sh,\ csp + c1) \in ntrp\ fg$ **by** (*auto intro*: *gtrp.gtrp-loc*)
also note *IHAPP*(*3*)
finally show *?thesis* **.**
qed
moreover from *CASE*(*1*) *ntrs-mon-e-no-ctx*[*OF CASE*(*2*)] *IHAPP*(*5*) **have** $mon\text{-}ww\ fg\ (map\ le\text{-}rem\text{-}s\ (ee\#w1)) \cap mon\text{-}c\ fg\ c2 = \{\}$ **by** (*auto simp add*: *mon-c-unconc*)
moreover from *ntrs-mon-increasing-s*[*OF CASE*(*2*)] *CHFMT*(*1*) *IHAPP*(*6*) **have** $mon\text{-}ww\ fg\ w2 \cap mon\text{-}c\ fg\ (\{\#s\#\} + c1) = \{\}$ **by** (*auto simp add*: *mon-c-unconc*)
moreover note *IHAPP*(*2*,*4*)
ultimately show *?thesis* **by** *blast*
next
case *gtrp-env* **then obtain** *e ss ce ssh ceh* **where** *CASE*: $ee=ENV\ e$ $c1+c2=add\text{-}mset\ ss\ ce$ $sh=s$ $ch=add\text{-}mset\ ssh\ ceh$ $((ss,add\text{-}mset\ s\ ce),e,(ssh,add\text{-}mset\ s\ ceh))\in ntrs$

fg **by** *auto*

from *ntrs-c-cases-s*[*OF CASE*(*5*)] **obtain** *csp* **where** *HFMT*: *add-mset s ceh* = *csp* + (*add-mset s ce*) ⋀*s*. *s* ∈# *csp* ⟹ ∃ *p u v*. *s* = [*entry fg p*] ∧ (*u*, *Spawn p*, *v*) ∈ *edges fg* ∧ *initialproc fg p* **by** (*blast*)

from *union-left-cancel*[*of* {#*s*#} *ceh csp*+*ce*] *HFMT*(*1*) **have** *CEHFMT*: *ceh*=*csp*+*ce* **by** (*auto simp add*: *union-ac*)

from *HFMT*(*2*) **have** *CHNOMON*: *mon-c fg csp* = {} **by** (*blast intro*!: *c-of-initial-no-mon*)

from *CASE*(*2*)[*symmetric*] **show** *?thesis* **proof** (*cases rule*: *mset-unplusm-dist-cases*)

— Made an env-step in *c1*, this is considered the „left” part. Apply induction hypothesis with original(!) local thread and the spawned threads on the left side

case *left*

with *HFMT*(*1*) *CASE*(*4*) *CEHFMT* **have** *CHFMT'*: *ch*=(*csp*+{#*ssh*#}+(*c1*−{#*ss*#})) + *c2* **by** (*simp add*: *union-ac*)

have *VALID*: *valid fg* ({#*s*#} + (*csp*+{#*ssh*#}+(*c1*−{#*ss*#})) + *c2*) **proof** −

from *ntr-valid-preserve-s*[*OF gtrI-s*, *OF CASE*(*5*)] *Cons.prems*(*2*) *CASE*(*2*) **have** *valid fg* ({#*ssh*#} + ({#*s*#} + *ceh*)) **by** (*simp add*: *union-assoc add-mset-commute*)

with *left CEHFMT* **show** *?thesis* **by** (*auto simp add*: *union-ac add-mset-commute*)

qed

from *Cons.hyps*[*OF* - *VALID*,*of s' c'*] *CHFMT' SPLIT*(*2*) *CASE*(*3*) **obtain** *w1 w2 c1' c2'* **where** *IHAPP*: $w \in w1 \otimes_{\alpha nl\ fg}$ *map ENV w2 c'* = *c1'* + *c2'*

((*s*, *csp* + {#*ssh*#} + (*c1* − {#*ss*#})), *w1*, *s'*, *c1'*) ∈ *trcl* (*ntrp fg*) (*c2*, *w2*, *c2'*) ∈ *trcl* (*ntr fg*)

mon-ww fg (*map le-rem-s w1*) ∩ *mon-c fg c2* = {} *mon-ww fg w2* ∩ *mon-c fg* ({#*s*#} + (*csp* + {#*ssh*#} + (*c1* − {#*ss*#}))) = {} **by** *blast*

have *ee* # $w \in (ee\#w1) \otimes_{\alpha nl\ fg}$ *map ENV w2* **proof** (*rule cil-cons1*)

from *IHAPP*(*6*) **have** *mon-ww fg w2* ∩ *mon-s fg ssh* = {} **by** (*auto simp add*: *mon-c-unconc*)

moreover from *ntrs-mon-s*[*OF CASE*(*5*)] *CASE*(*1*) **have** *fst* (*αnl fg ee*) ⊆ *mon-s fg ssh* **by** *auto*

ultimately have *fst* (*αnl fg ee*) ∩ *mon-ww fg w2* = {} **by** *auto*

moreover have *mon-pl* (*map* (*αnl fg*) (*map ENV w2*)) = *mon-ww fg w2* **by** (*simp add*: *cil-αn-cons-helper*)

ultimately show *fst* (*αnl fg ee*) ∩ *mon-pl* (*map* (*αnl fg*) (*map ENV w2*)) = {} **by** *auto*

qed (*rule IHAPP*(*1*))

moreover

have *SS*: ((*s*,*c1*),*ee*,(*s*,*csp* + {#*ssh*#} + (*c1* − {#*ss*#})))∈*ntrp fg* **proof** −

from *left HFMT*(*1*) **have** {#*s*#}+*ce*={#*s*#}+(*c1*−{#*ss*#})+*c2* {#*s*#}+*ceh* = *csp*+({#*s*#}+(*c1*−{#*ss*#})+*c2*) **by** (*simp-all add*: *union-ac*)

with *CASE*(*5*) *ntrs-xchange-context-s*[*of ss* {#*s*#}+(*c1*−{#*ss*#})+*c2 e ssh csp fg* ({#*s*#}+(*c1*−{#*ss*#}))] **have**

((*ss*, *add-mset s* (*c1* − {#*ss*#})), *e*, *ssh*, *add-mset s* (*csp*+(*c1* − {#*ss*#}))) ∈ *ntrs fg* **by** (*auto simp add*: *mon-c-unconc union-ac*)

from *gtrp.gtrp-env*[*OF this*] *left*(*1*)[*symmetric*] *CASE*(*1*) **show** *?thesis* **by** (*simp add*: *union-ac*)

qed

from *trcl.cons*[*OF this IHAPP*(*3*)] **have** ((*s*, *c1*), *ee* # *w1*, *s'*, *c1'*) ∈ *trcl* (*ntrp*

fg) .
moreover
from *ntrs-mon-e-no-ctx*[*OF CASE*(*5*)] *left CASE*(*1*) *IHAPP*(*5*) **have** *mon-ww fg* (*map le-rem-s* (*ee*#*w1*)) ∩ *mon-c fg c2* = {} **by** (*auto simp add*: *mon-c-unconc*)
moreover
from *ntrp-mon-increasing-s*[*OF SS*] *IHAPP*(*6*) **have** *mon-ww fg w2* ∩ *mon-c fg* ({#*s*#} + *c1*) = {} **by** (*auto simp add*: *mon-c-unconc*)
moreover note *IHAPP*(*2*,*4*)
ultimately show *?thesis* **by** *blast*
next
— Made an env-step in *c2*. This is considered the right part. Induction hypothesis is applied with original local thread and the spawned threads on the right side
case *right*
with *HFMT*(*1*) *CASE*(*4*) *CEHFMT* **have** *CHFMT′*: *ch*=*c1* + (*csp*+{#*ssh*#}+(*c2*−{#*ss*#})) **by** (*simp add*: *union-ac*)
have *VALID*: *valid fg* ({#*s*#} + *c1* + ((*csp*+{#*ssh*#}+(*c2*−{#*ss*#}))))
proof −
from *ntr-valid-preserve-s*[*OF gtrI-s*, *OF CASE*(*5*)] *Cons.prems*(*2*) *CASE*(*2*) **have** *valid fg* ({#*ssh*#} + ({#*s*#} + *ceh*)) **by** (*auto simp add*: *union-ac add-mset-commute*)
with *right CEHFMT* **show** *?thesis* **by** (*auto simp add*: *union-ac add-mset-commute*)
qed
from *Cons.hyps*[*OF - VALID*,*of s′ c′*] *CHFMT′ SPLIT*(*2*) *CASE*(*3*) **obtain** *w1 w2 c1′ c2′* **where** *IHAPP*: $w \in w1 \otimes_{\alpha nl\ fg}$ *map ENV w2 c′* = *c1′* + *c2′*
((*s*, *c1*), *w1*, *s′*, *c1′*) ∈ *trcl* (*ntrp fg*) (*csp* + {#*ssh*#} + (*c2* − {#*ss*#}), *w2*, *c2′*) ∈ *trcl* (*ntr fg*)
mon-ww fg (*map le-rem-s w1*) ∩ *mon-c fg* (*csp* + {#*ssh*#} + (*c2* − {#*ss*#})) = {} *mon-ww fg w2* ∩ *mon-c fg* ({#*s*#} + *c1*) = {} **by** *blast*
have $ee \# w \in w1 \otimes_{\alpha nl\ fg}$ *map ENV* (*e*#*w2*) **proof** (*simp add*: *CASE*(*1*), *rule cil-cons2*)
from *IHAPP*(*5*) **have** *mon-ww fg* (*map le-rem-s w1*) ∩ *mon-s fg ssh* = {} **by** (*auto simp add*: *mon-c-unconc*)
moreover from *ntrs-mon-s*[*OF CASE*(*5*)] *CASE*(*1*) **have** *fst* (*αnl fg ee*) ⊆ *mon-s fg ssh* **by** *auto*
ultimately have *fst* (*αnl fg ee*) ∩ *mon-ww fg* (*map le-rem-s w1*) = {} **by** *auto*
moreover have *mon-pl* (*map* (*αnl fg*) *w1*) = *mon-ww fg* (*map le-rem-s w1*) **by** (*unfold αnl-def′*) (*simp add*: *cil-αn-cons-helper*[*symmetric*])
ultimately show *fst* (*αnl fg* (*ENV e*)) ∩ *mon-pl* (*map* (*αnl fg*) *w1*) = {} **using** *CASE*(*1*) **by** *auto*
qed (*rule IHAPP*(*1*))
moreover
have *SS*: (*c2*,*e*,*csp* + {#*ssh*#} + (*c2* − {#*ss*#}))∈*ntr fg* **proof** −
from *right HFMT*(*1*) **have** {#*s*#}+*ce*={#*s*#}+*c1*+(*c2*−{#*ss*#}) {#*s*#}+*ceh* = *csp*+({#*s*#}+*c1*+(*c2*−{#*ss*#})) **by** (*simp-all add*: *union-ac*)
with *CASE*(*5*) *ntrs-xchange-context-s*[*of ss* {#*s*#}+*c1*+(*c2*−{#*ss*#}) *e ssh csp fg c2*−{#*ss*#}] **have**
((*ss*, (*c2* − {#*ss*#})), *e*, *ssh*, *csp*+ (*c2* − {#*ss*#})) ∈ *ntrs fg* **by** (*auto simp add*: *mon-c-unconc union-ac*)

from *gtrI-s*[*OF this*] *right*(*1*)[*symmetric*] **show** *?thesis* **by** (*simp add*: *union-ac*)
qed
from *trcl.cons*[*OF this IHAPP*(*4*)] **have** (*c2*, *e* # *w2*, *c2'*) ∈ *trcl* (*ntr fg*) **.**
moreover
from *ntr-mon-increasing-s*[*OF SS*] *IHAPP*(*5*) **have** *mon-ww fg* (*map le-rem-s w1*) ∩ *mon-c fg c2* = {} **by** (*auto simp add*: *mon-c-unconc*)
moreover
from *ntrs-mon-e-no-ctx*[*OF CASE*(*5*)] *right IHAPP*(*6*) **have** *mon-ww fg* (*e#w2*) ∩ *mon-c fg* ({#*s*#} + *c1*) = {} **by** (*auto simp add*: *mon-c-unconc*)
moreover note *IHAPP*(*2*,*3*)
ultimately show *?thesis* **by** *blast*
qed
qed
qed

— Just a check that *flowgraph.ntrp-split* is really a generalization of *flowgraph.ntr-split*:
lemma (**in** *flowgraph*) *ntr-split'*:
assumes *A*: (*ca*+*cb*,*w*,*c'*)∈*trcl* (*ntr fg*)
and *VALID*: *valid fg* (*ca*+*cb*)
shows ∃ *ca' cb' wa wb*.
c'=*ca'*+*cb'* ∧
w∈(*wa*$\otimes_{\alpha n\ fg}$*wb*) ∧
mon-c fg ca ∩ (*mon-c fg cb* ∪ *mon-ww fg wb*) = {} ∧
mon-c fg cb ∩ (*mon-c fg ca* ∪ *mon-ww fg wa*) = {} ∧
(*ca*,*wa*,*ca'*)∈*trcl* (*ntr fg*) ∧
(*cb*,*wb*,*cb'*)∈*trcl* (*ntr fg*)
using *A VALID* **by**(*rule ntr-split*)

The unsplit lemma combines two interleavable executions. For illustration purposes, we first prove the less general version for multiset-configurations. The general version for loc/env-configurations is shown later.

lemma (**in** *flowgraph*) *ntr-unsplit*:
assumes *A*: *w*∈*wa*$\otimes_{\alpha n\ fg}$*wb* **and**
B: (*ca*,*wa*,*ca'*)∈*trcl* (*ntr fg*)
(*cb*,*wb*,*cb'*)∈*trcl* (*ntr fg*)
mon-c fg ca ∩ (*mon-c fg cb* ∪ *mon-ww fg wb*)={}
mon-c fg cb ∩ (*mon-c fg ca* ∪ *mon-ww fg wa*)={}
shows (*ca*+*cb*,*w*,*ca'*+*cb'*)∈*trcl* (*ntr fg*)
proof −
— We have to generalize and rewrite the goal, in order to apply Isabelle's induction method
from *A* **have** ∀ *ca cb*. (*ca*,*wa*,*ca'*)∈*trcl* (*ntr fg*) ∧ (*cb*,*wb*,*cb'*)∈*trcl* (*ntr fg*) ∧ *mon-c fg ca* ∩ (*mon-c fg cb* ∪ *mon-ww fg wb*)={} ∧ *mon-c fg cb* ∩ (*mon-c fg ca* ∪ *mon-ww fg wa*)={} ⟶
(*ca*+*cb*,*w*,*ca'*+*cb'*)∈*trcl* (*ntr fg*)
— We prove the generalized goal by induction over the structure of consistent interleaving
proof (*induct rule*: *cil-set-induct-fixα*)

— If both words are empty, the proposition is trivial
case *empty* **thus** *?case* **by** *simp*
next
— The first macrostep of the combined path was taken from the left operand of the interleaving
case (*left e w′ w1′ w2*) **thus** *?case*
proof (*intro allI impI*, *goal-cases*)
case (*1 ca cb*)
hence *I*: $w' \in w1' \otimes_{\alpha n\ fg}$ *w2 fst* (αn *fg e*) ∩ *mon-pl* (*map* (αn *fg*) *w2*) = {}
!!*ca cb.*
⟦(*ca*, *w1′*, *ca′*) ∈ *trcl* (*ntr fg*);
(*cb*, *w2*, *cb′*) ∈ *trcl* (*ntr fg*);
mon-c fg ca ∩ (*mon-c fg cb* ∪ *mon-ww fg w2*) = {};
mon-c fg cb ∩ (*mon-c fg ca* ∪ *mon-ww fg w1′*) = {}⟧ ⟹
(*ca* + *cb*, *w′*, *ca′* + *cb′*) ∈ *trcl* (*ntr fg*)
(*ca*, *e* # *w1′*, *ca′*) ∈ *trcl* (*ntr fg*) (*cb*, *w2*, *cb′*) ∈ *trcl* (*ntr fg*)
mon-c fg ca ∩ (*mon-c fg cb* ∪ *mon-ww fg w2*) = {}
mon-c fg cb ∩ (*mon-c fg ca* ∪ *mon-ww fg* (*e* # *w1′*)) = {} **by** *blast+*
— Split the left path after the first step
then obtain *cah* **where** *SPLIT*: (*ca*,*e*,*cah*)∈*ntr fg* (*cah*,*w1′*,*ca′*)∈*trcl* (*ntr fg*) **by** (*fast dest*: *trcl-uncons*)
— and combine the first step of the left path with the initial right context
from *ntr-add-context-s*[*OF SPLIT*(*1*), **where** *cn*=*cb*] *I*(*7*) **have** (*ca* + *cb*, *e*, *cah* + *cb*) ∈ *ntr fg* **by** *auto*
also
— The rest of the path is combined by using the induction hypothesis
have (*cah* + *cb*, *w′*, *ca′* + *cb′*) ∈ *trcl* (*ntr fg*) **proof** −
from *I*(*2*,*6*,*7*) *ntr-mon-s*[*OF SPLIT*(*1*)] **have** *MON-CAH*: *mon-c fg cah* ∩ (*mon-c fg cb* ∪ *mon-ww fg w2*) = {} **by** (*cases e*) (*auto simp add*: *cil-αn-cons-helper*)

with *I*(*7*) **have** *MON-CB*: *mon-c fg cb* ∩ (*mon-c fg cah* ∪ *mon-ww fg w1′*) = {} **by** *auto*
from *I*(*3*)[*OF SPLIT*(*2*) *I*(*5*) *MON-CAH MON-CB*] **show** *?thesis* **.**
qed
finally show *?case* **.**
qed
next
— The first macrostep of the combined path was taken from the right path – this case is done completely analogous
case (*right e w′ w2′ w1*) **thus** *?case*
proof (*intro allI impI*, *goal-cases*)
case (*1 ca cb*)
hence *I*: $w' \in w1 \otimes_{\alpha n\ fg}$ *w2′ fst* (αn *fg e*) ∩ *mon-pl* (*map* (αn *fg*) *w1*) = {}
!!*ca cb.*
⟦(*ca*, *w1*, *ca′*) ∈ *trcl* (*ntr fg*);
(*cb*, *w2′*, *cb′*) ∈ *trcl* (*ntr fg*);
mon-c fg ca ∩ (*mon-c fg cb* ∪ *mon-ww fg w2′*) = {};
mon-c fg cb ∩ (*mon-c fg ca* ∪ *mon-ww fg w1*) = {}⟧ ⟹
(*ca* + *cb*, *w′*, *ca′* + *cb′*) ∈ *trcl* (*ntr fg*)

$(ca,\ w1,\ ca') \in trcl\ (ntr\ fg)$ $(cb,\ e\#w2',\ cb') \in trcl\ (ntr\ fg)$
$mon\text{-}c\ fg\ ca \cap (mon\text{-}c\ fg\ cb \cup mon\text{-}ww\ fg\ (e\#w2')) = \{\}$
$mon\text{-}c\ fg\ cb \cap (mon\text{-}c\ fg\ ca \cup mon\text{-}ww\ fg\ w1) = \{\}$ **by** *blast+*
then obtain *cbh* **where** *SPLIT*: $(cb,e,cbh) \in ntr\ fg$ $(cbh,w2',cb') \in trcl\ (ntr\ fg)$ **by** (*fast dest*: *trcl-uncons*)
from *ntr-add-context-s*[*OF SPLIT*(*1*), **where** *cn*=*ca*] *I*(*6*) **have** $(ca + cb,\ e,\ ca + cbh) \in ntr\ fg$ **by** (*auto simp add*: *union-commute*)
also
have $(ca + cbh,\ w',\ ca' + cb') \in trcl\ (ntr\ fg)$ **proof** −
from *I*(*2*,*6*,*7*) *ntr-mon-s*[*OF SPLIT*(*1*)] **have** *MON-CBH*: $mon\text{-}c\ fg\ cbh \cap (mon\text{-}c\ fg\ ca \cup mon\text{-}ww\ fg\ w1) = \{\}$ **by** (*cases e*) (*auto simp add*: *cil-αn-cons-helper*)
with *I*(*6*) **have** *MON-CA*: $mon\text{-}c\ fg\ ca \cap (mon\text{-}c\ fg\ cbh \cup mon\text{-}ww\ fg\ w2') = \{\}$ **by** *auto*
from *I*(*3*)[*OF I*(*4*) *SPLIT*(*2*) *MON-CA MON-CBH*] **show** *?thesis* **.**
qed
finally show *?case* **.**
qed
qed
with *B* **show** *?thesis* **by** *blast*
qed

lemma (**in** *flowgraph*) *ntrp-unsplit*:
assumes *A*: $w \in wa \otimes_{\alpha nl\ fg} (map\ ENV\ wb)$ **and**
B: $((s,ca),wa,(s',ca')) \in trcl\ (ntrp\ fg)$
$(cb,wb,cb') \in trcl\ (ntr\ fg)$
$mon\text{-}c\ fg\ (\{\#s\#\}+ca) \cap (mon\text{-}c\ fg\ cb \cup mon\text{-}ww\ fg\ wb) = \{\}$
$mon\text{-}c\ fg\ cb \cap (mon\text{-}c\ fg\ (\{\#s\#\}+ca) \cup mon\text{-}ww\ fg\ (map\ le\text{-}rem\text{-}s\ wa)) = \{\}$
shows $((s,ca+cb),w,(s',ca'+cb')) \in trcl\ (ntrp\ fg)$
proof −
{ fix *wb'*
have $w \in wa \otimes_{\alpha nl\ fg} wb' \Longrightarrow$
$\forall\ s\ ca\ cb\ wb.\ wb'=map\ ENV\ wb \wedge$
$((s,ca),wa,(s',ca')) \in trcl\ (ntrp\ fg) \wedge (cb,wb,cb') \in trcl\ (ntr\ fg) \wedge mon\text{-}c\ fg\ (\{\#s\#\}+ca) \cap (mon\text{-}c\ fg\ cb \cup mon\text{-}ww\ fg\ wb) = \{\} \wedge mon\text{-}c\ fg\ cb \cap (mon\text{-}c\ fg\ (\{\#s\#\}+ca) \cup mon\text{-}ww\ fg\ (map\ le\text{-}rem\text{-}s\ wa)) = \{\} \longrightarrow$
$((s,ca+cb),w,(s',ca'+cb')) \in trcl\ (ntrp\ fg)$
proof (*induct rule*: *cil-set-induct-fixα*)
case *empty* **thus** *?case* **by** *simp*
next
case (*left e w' w1' w2*)
thus *?case*
proof (*intro allI impI*, *goal-cases*)
case (*1 s ca cb wb*)
hence *I*: $w' \in w1' \otimes_{\alpha nl\ fg} w2\ fst\ (\alpha nl\ fg\ e) \cap mon\text{-}pl\ (map\ (\alpha nl\ fg)\ w2) = \{\}$
$!!s\ ca\ cb\ wb.\ [\![$
$w2 = map\ ENV\ wb;$
$((s,\ ca),\ w1',\ s',\ ca') \in trcl\ (ntrp\ fg);$

$(cb,\ wb,\ cb') \in trcl\ (ntr\ fg)$;
$mon\text{-}c\ fg\ (\{\#s\#\} + ca) \cap (mon\text{-}c\ fg\ cb \cup mon\text{-}ww\ fg\ wb) = \{\}$;
$mon\text{-}c\ fg\ cb \cap (mon\text{-}c\ fg\ (\{\#s\#\} + ca) \cup mon\text{-}ww\ fg\ (map\ le\text{-}rem\text{-}s\ w1')) = \{\}$
$\rrbracket \Longrightarrow ((s,\ ca + cb),\ w',\ s',\ ca' + cb') \in trcl\ (ntrp\ fg)$
$w2 = map\ ENV\ wb$
$((s,\ ca),\ e\ \#\ w1',\ s',\ ca') \in trcl\ (ntrp\ fg)$
$(cb,\ wb,\ cb') \in trcl\ (ntr\ fg)$
$mon\text{-}c\ fg\ (\{\#s\#\} + ca) \cap (mon\text{-}c\ fg\ cb \cup mon\text{-}ww\ fg\ wb) = \{\}$
$mon\text{-}c\ fg\ cb \cap (mon\text{-}c\ fg\ (\{\#s\#\} + ca) \cup mon\text{-}ww\ fg\ (map\ le\text{-}rem\text{-}s\ (e\ \#\ w1'))) = \{\}$
by *blast+*
then obtain *sh cah* **where** *SPLIT*: $((s,ca),e,(sh,cah)) \in ntrp\ fg\ ((sh,cah),w1',(s',ca')) \in trcl\ (ntrp\ fg)$ **by** (*fast dest*: *trcl-uncons*)
from *ntrp-add-context-s*[*OF SPLIT*(*1*), *of cb*] *I*(*8*) **have** $((s,\ ca + cb),\ e,\ sh,\ cah + cb) \in ntrp\ fg$ **by** *auto*
also have $((sh,cah+cb),w',(s',ca'+cb')) \in trcl\ (ntrp\ fg)$ **proof** (*rule I*(*3*))
from *ntrp-mon-s*[*OF SPLIT*(*1*)] *I*(*2*,*4*,*7*,*8*) **show** *1*: $mon\text{-}c\ fg\ (\{\#sh\#\} + cah) \cap (mon\text{-}c\ fg\ cb \cup mon\text{-}ww\ fg\ wb) = \{\}$
by (*cases e*) (*rename-tac a*, *case-tac a*, *simp add*: *cil-αn-cons-helper*, *fastforce simp add*: *cil-αn-cons-helper*)+
from *I*(*8*) *1* **show** $mon\text{-}c\ fg\ cb \cap (mon\text{-}c\ fg\ (\{\#sh\#\} + cah) \cup mon\text{-}ww\ fg\ (map\ le\text{-}rem\text{-}s\ w1')) = \{\}$ **by** *auto*
qed (*auto simp add*: *I*(*4*,*6*) *SPLIT*(*2*))
finally show *?case* **.**
qed
next
case (*right ee w' w2' w1*)
thus *?case*
proof (*intro allI impI*, *goal-cases*)
case (*1 s ca cb wb*)
hence *I*: $w' \in w1 \otimes_{\alpha nl\ fg} w2'\ fst\ (\alpha nl\ fg\ ee) \cap mon\text{-}pl\ (map\ (\alpha nl\ fg)\ w1) = \{\}$
!!*s ca cb wb*. $\llbracket$
$w2' = map\ ENV\ wb$;
$((s,\ ca),\ w1,\ s',\ ca') \in trcl\ (ntrp\ fg)$;
$(cb,\ wb,\ cb') \in trcl\ (ntr\ fg)$;
$mon\text{-}c\ fg\ (\{\#s\#\} + ca) \cap (mon\text{-}c\ fg\ cb \cup mon\text{-}ww\ fg\ wb) = \{\}$;
$mon\text{-}c\ fg\ cb \cap (mon\text{-}c\ fg\ (\{\#s\#\} + ca) \cup mon\text{-}ww\ fg\ (map\ le\text{-}rem\text{-}s\ w1)) = \{\}$
$\rrbracket \Longrightarrow ((s,\ ca + cb),\ w',\ s',\ ca' + cb') \in trcl\ (ntrp\ fg)$
$ee\#w2' = map\ ENV\ wb$
$((s,\ ca),\ w1,\ s',\ ca') \in trcl\ (ntrp\ fg)$
$(cb,\ wb,\ cb') \in trcl\ (ntr\ fg)$
$mon\text{-}c\ fg\ (\{\#s\#\} + ca) \cap (mon\text{-}c\ fg\ cb \cup mon\text{-}ww\ fg\ wb) = \{\}$
$mon\text{-}c\ fg\ cb \cap (mon\text{-}c\ fg\ (\{\#s\#\} + ca) \cup mon\text{-}ww\ fg\ (map\ le\text{-}rem\text{-}s\ w1)) = \{\}$
by *fastforce+*
from *I*(*4*) **obtain** *e wb'* **where** *EE*: $wb=e\#wb'\ ee=ENV\ e\ w2'=map\ ENV$

wb′ **by** (*cases wb*, *auto*)
with *I*(*6*) **obtain** *cbh* **where** *SPLIT*: (*cb*,*e*,*cbh*)∈*ntr fg* (*cbh*,*wb′*,*cb′*)∈*trcl* (*ntr fg*) **by** (*fast dest*: *trcl-uncons*)
have ((*s*, *ca* + *cb*), *ee*, (*s*, *ca* + *cbh*)) ∈ *ntrp fg* **proof** −
from *gtrE*[*OF SPLIT*(*1*)] **obtain** *sb ceb sbh cebh* **where** *NTRS*: *cb* = *add-mset sb ceb cbh* = *add-mset sbh cebh* ((*sb*, *ceb*), *e*, *sbh*, *cebh*) ∈ *ntrs fg* **.**
from *ntrs-add-context-s*[*OF NTRS*(*3*), *of* {#*s*#}+*ca*] *EE*(*1*) *I*(*7*) **have** ((*sb*, *add-mset s* (*ca*+*ceb*)), *e*, *sbh*, *add-mset s* (*ca*+*cebh*)) ∈ *ntrs fg* **by** (*auto simp add*: *union-ac*)
from *gtrp-env*[*OF this*] *NTRS*(*1*,*2*) *EE*(*2*) **show** *?thesis* **by** (*simp add*: *union-ac*)
qed
also have ((*s*,*ca*+*cbh*),*w′*,(*s′*,*ca′*+*cb′*))∈*trcl* (*ntrp fg*) **proof** (*rule I*(*3*))
from *ntr-mon-s*[*OF SPLIT*(*1*)] *I*(*2*,*4*,*7*,*8*) *EE*(*2*) **show** *1*: *mon-c fg cbh* ∩ (*mon-c fg* ({#*s*#} + *ca*) ∪ *mon-ww fg* (*map le-rem-s w1*)) = {}
by (*cases e*) (*simp add*: *cil-αnl-cons-helper*, *fastforce simp add*: *cil-αnl-cons-helper*)
from *I*(*7*) *1 EE*(*1*) **show** *mon-c fg* ({#*s*#} + *ca*) ∩ (*mon-c fg cbh* ∪ *mon-ww fg wb′*) = {} **by** *auto*
qed (*auto simp add*: *EE*(*3*) *I*(*5*) *SPLIT*(*2*))
finally show *?case* **.**
qed
qed
}
with *A B* **show** *?thesis* **by** *blast*
qed

And finally we get the desired theorem: *Two paths are simultaneously executable if and only if they are consistently interleavable and the monitors of the initial configurations are compatible.* Note that we have to assume a valid starting configuration.

theorem (**in** *flowgraph*) *ntr-interleave*: *valid fg* (*ca*+*cb*) ⟹
(*ca*+*cb*,*w*,*c′*)∈*trcl* (*ntr fg*) ⟷
(∃ *ca′ cb′ wa wb*.
c′=*ca′*+*cb′* ∧
w∈(*wa*$\otimes_{\alpha n\ fg}$*wb*) ∧
mon-c fg ca ∩ (*mon-c fg cb* ∪ *mon-ww fg wb*) = {} ∧
mon-c fg cb ∩ (*mon-c fg ca* ∪ *mon-ww fg wa*) = {} ∧
(*ca*,*wa*,*ca′*)∈*trcl* (*ntr fg*) ∧ (*cb*,*wb*,*cb′*)∈*trcl* (*ntr fg*))
by (*blast intro*!: *ntr-split ntr-unsplit*)

— Here is the corresponding version for executions with an explicit local thread
theorem (**in** *flowgraph*) *ntrp-interleave*:
valid fg ({#*s*#}+*c1*+*c2*) ⟹
((*s*,*c1*+*c2*),*w*,(*s′*,*c′*))∈*trcl* (*ntrp fg*) ⟷
(∃ *w1 w2 c1′ c2′*.
w ∈ *w1* $\otimes_{\alpha nl\ fg}$ (*map ENV w2*) ∧
c′=*c1′*+*c2′* ∧
((*s*,*c1*),*w1*,(*s′*,*c1′*))∈*trcl* (*ntrp fg*) ∧

$(c2,w2,c2')\in trcl\ (ntr\ fg) \wedge$
$mon\text{-}ww\ fg\ (map\ le\text{-}rem\text{-}s\ w1) \cap$
$mon\text{-}c\ fg\ c2 = \{\} \wedge$
$mon\text{-}ww\ fg\ w2 \cap mon\text{-}c\ fg\ (\{\#s\#\}+c1) = \{\})$
apply (*intro iffI*)
apply (*blast intro*: *ntrp-split*)
apply (*auto intro*!: *ntrp-unsplit simp add*: *valid-unconc mon-c-unconc*)
done

The next is a corollary of *flowgraph.ntrp-unsplit*, allowing us to convert a path to loc/env semantics by adding a local stack that does not make any steps.

corollary (**in** *flowgraph*) *ntr2ntrp*: $[\![$
$(c,w,c')\in trcl\ (ntr\ fg)$;
$mon\text{-}c\ fg\ (add\text{-}mset\ s\ cl) \cap (mon\text{-}c\ fg\ c \cup mon\text{-}ww\ fg\ w)=\{\}$
$]\!] \Longrightarrow ((s,cl+c),map\ ENV\ w,(s,cl+c'))\in trcl\ (ntrp\ fg)$
using *ntrp-unsplit*[**where** *wa*=[], *simplified*] **by** *fast*

8.5.4 Reverse splitting

This section establishes a theorem that allows us to find the thread in the original configuration that created some distinguished thread in the final configuration.

lemma (**in** *flowgraph*) *ntr-reverse-split*: !!*w s' ce'*. $[\![$
$(c,w,\{\#s'\#\}+ce')\in trcl\ (ntr\ fg)$;
$valid\ fg\ c\ ]\!] \Longrightarrow$
$\exists\, s\ ce\ w1\ w2\ ce1'\ ce2'.$
$c=\{\#s\#\}+ce \wedge$
$ce'=ce1'+ce2' \wedge$
$w\in w1\otimes_{\alpha n\ fg}\ w2 \wedge$
$mon\text{-}s\ fg\ s \cap (mon\text{-}c\ fg\ ce \cup mon\text{-}ww\ fg\ w2) = \{\} \wedge$
$mon\text{-}c\ fg\ ce \cap (mon\text{-}s\ fg\ s \cup mon\text{-}ww\ fg\ w1) = \{\} \wedge$
$(\{\#s\#\},w1,\{\#s'\#\}+ce1')\in trcl\ (ntr\ fg) \wedge$
$(ce,w2,ce2')\in trcl\ (ntr\ fg)$
— The proof works by induction on the initial configuration. Note that configurations consist of finitely many threads only
— FIXME: An induction over the size (rather then over the adding of some fixed element) may lead to a smoother proof here
proof (*induct c rule*: *multiset-induct'*)
— If the initial configuration is empty, we immediately get a contradiction
case *empty* **hence** *False* **by** *auto* **thus** *?case* **..**
next
— The initial configuration has the form $\{\#s\#\}+ce$.
case (*add ce s*)
— We split the path by this initial configuration
from *ntr-split*[*OF add.prems*(*1*,*2*)] **obtain** *ce1' ce2' w1 w2* **where**
SPLIT: *add-mset s' ce'=ce1'+ce2'* $w\in w1\otimes_{\alpha n\ fg} w2$

mon-c fg ce ∩ (*mon-s fg s*∪*mon-ww fg w1*) = {}
mon-s fg s ∩ (*mon-c fg ce* ∪ *mon-ww fg w2*) = {}
({#*s*#},*w1*,*ce1′*)∈*trcl* (*ntr fg*)
(*ce*,*w2*,*ce2′*)∈*trcl* (*ntr fg*)
by *auto*
— And then check whether splitting off *s* was the right choice
from *SPLIT*(*1*) **show** *?case* **proof** (*cases rule*: *mset-unplusm-dist-cases*)
— Our choice was correct, *s′* is generated by some descendant of *s*"
case *left*
with *SPLIT* **show** *?thesis* **by** *fastforce*
next
— Our choice was not correct, *s′* is generated by some descendant of *ce*
case *right* **with** *SPLIT*(*6*) **have** *C*: (*ce*,*w2*,{#*s′*#}+(*ce2′*−{#*s′*#}))∈*trcl* (*ntr fg*) **by** *auto*
— In this case we apply the induction hypothesis to the path from *ce*
from *add.prems*(*2*) **have** *VALID*: *valid fg ce mon-s fg s* ∩ *mon-c fg ce* = {} **by** (*simp-all add*: *valid-unconc*)
from *add.hyps*[*OF C VALID*(*1*)] **obtain** *st cet w21 w22 ce21′ ce22′* **where**
IHAPP:
ce={#*st*#}+*cet*
ce2′−{#*s′*#} = *ce21′*+*ce22′*
$w2 \in w21 \otimes_{\alpha n\ fg} w22$
mon-s fg st ∩ (*mon-c fg cet* ∪ *mon-ww fg w22*)={}
mon-c fg cet ∩ (*mon-s fg st* ∪ *mon-ww fg w21*)={}
({#*st*#},*w21*,{#*s′*#}+*ce21′*)∈*trcl* (*ntr fg*)
(*cet*,*w22*,*ce22′*)∈*trcl* (*ntr fg*) **by** *blast*

— And finally we add the path from *s* again. This requires some monitor sorting and the associativity of the consistent interleaving operator.
from *cil-assoc2* [*of w w1 - w2 w22 w21*] *SPLIT*(*2*) *IHAPP*(*3*) **obtain** *wl* **where** *CASSOC*: $w \in w21 \otimes_{\alpha n\ fg} wl$ $wl \in w1 \otimes_{\alpha n\ fg} w22$ **by** (*auto simp add*: *cil-commute*)
from *CASSOC IHAPP*(*1*,*3*,*4*,*5*) *SPLIT*(*3*,*4*) **have** *COMBINE*: (*add-mset s cet*, *wl*, *ce1′* + *ce22′*) ∈ *trcl* (*ntr fg*) **using** *ntr-unsplit*[*OF CASSOC*(*2*) *SPLIT*(*5*) *IHAPP*(*7*)] **by** (*auto simp add*: *mon-c-unconc mon-ww-cil Int-Un-distrib2*)
moreover from *CASSOC IHAPP*(*1*,*3*,*4*,*5*) *SPLIT*(*3*,*4*) **have** *mon-s fg st* ∩ (*mon-c fg* ({#*s*#}+*cet*) ∪ *mon-ww fg wl*) = {} *mon-c fg* ({#*s*#}+*cet*) ∩ (*mon-s fg st* ∪ *mon-ww fg w21*) = {} **by** (*auto simp add*: *mon-c-unconc mon-ww-cil*)
moreover from *right IHAPP*(*1*,*2*) **have** {#*s*#}+*ce*={#*st*#}+({#*s*#}+*cet*) *ce′*=*ce21′*+(*ce1′*+*ce22′*) **by** (*simp-all add*: *union-ac*)
moreover note *IHAPP*(*6*) *CASSOC*(*1*)
ultimately show *?thesis* **by** *fastforce*
qed
qed

end

9 Constraint Systems

theory *ConstraintSystems*

imports *Main AcquisitionHistory Normalization*
begin

In this section we develop a constraint-system-based characterization of our analysis.

Constraint systems are widely used in static program analysis. There least solution describes the desired analysis information. In its generic form, a constraint system R is a set of inequations over a complete lattice $(L, \sqsubseteq)$ and a set of variables V. An inequation has the form $R[v] \sqsupseteq \mathsf{rhs}$, where $R[v] \in V$ and rhs is a monotonic function over the variables. Note that for program analysis, there is usually one variable per control point. The variables are then named $R[v]$, where v is a control point. By standard fixed-point theory, those constraint systems have a least solution. Outside the constraint system definition $R[v]$ usually refers to a component of that least solution.
Usually a constraint system is generated from the program. For example, a constraint generation pattern could be the following:

$$\begin{array}{lll} \multicolumn{3}{l}{\text{for } (u, \mathsf{Call}\ q, v) \in E:} \\ S^k[v] & \supseteq & \{(\mathsf{mon}(q) \cup M \cup M', \tilde{P}) \mid (M, P) \in S^k[u] \wedge (M', P') \in S^k[\mathsf{r}_q] \\ & & \wedge\ \tilde{P} \leq P \uplus P' \wedge |\tilde{P}| \leq 2\} \end{array}$$

For some parameter k and a flowgraph with nodes N and edges E, this generates a constraint system over the variables $\{S^k[v] \mid v \in N\}$. One constraint is generated for each call edge. While we use a powerset lattice here, we can in general use any complete lattice. However, all the constraint systems needed for our conflict analysis are defined over powerset lattices $(\mathcal{P}('a), \subseteq)$ for some type $'a$. This admits a convenient formalization in Isabelle/HOL using inductively defined sets. We inductively define a relation between variables[3] and the elements of their values in the least solution, i.e. the set $\{(v, x) \mid x \in R[v]\}$. For example, the constraint generator pattern from above would become the following introduction rule in the inductive definition of the set *S-cs fg k*:

$$\begin{array}{l} [\![(u,\mathit{Call}\ q,v)\in \mathit{edges}\ \mathit{fg};\ (u,M,P)\in \mathit{S\text{-}cs}\ \mathit{fg}\ k; \\ \quad\quad (\mathit{return}\ \mathit{fg}\ q,\mathit{Ms},\mathit{Ps})\in \mathit{S\text{-}cs}\ \mathit{fg}\ k;\ P'\subseteq\#P+\mathit{Ps};\ \mathit{size}\ P' \leq k\ ]\!] \\ \quad\quad \Longrightarrow (v,\mathit{mon}\ \mathit{fg}\ q \cup M \cup \mathit{Ms},P')\in \mathit{S\text{-}cs}\ \mathit{fg}\ k \end{array}$$

The main advantage of this approach is that one gets a concise formalization by using Isabelle's standard machinery, the main disadvantage is that this approach only works for powerset lattices ordered by $\subseteq$.

[3]Variables are identified by control nodes here

9.1 Same-level paths

9.1.1 Definition

We define a constraint system that collects abstract information about same-level paths. In particular, we collect the set of used monitors and all multi-subsets of spawned threads that are not bigger than k elements, where k is a parameter that can be freely chosen.

An element $(u,M,P) \in S\text{-}cs\ fg\ k$ means that there is a same-level path from the entry node of the procedure of u to u, that uses the monitors M and spawns at least the threads in P.

inductive-set
  $S\text{-}cs :: ('n,'p,'ba,'m,'more)\ flowgraph\text{-}rec\text{-}scheme \Rightarrow nat \Rightarrow$
  $\quad ('n \times 'm\ set \times 'p\ multiset)\ set$
  for $fg\ k$
  where
  $S\text{-}init$: $(entry\ fg\ p,\{\},\{\#\}) \in S\text{-}cs\ fg\ k$
  $|\ S\text{-}base$: $[\![(u,Base\ a,v) \in edges\ fg;\ (u,M,P) \in S\text{-}cs\ fg\ k]\!] \Longrightarrow (v,M,P) \in S\text{-}cs\ fg\ k$
  $|\ S\text{-}call$: $[\![(u,Call\ q,v) \in edges\ fg;\ (u,M,P) \in S\text{-}cs\ fg\ k;$
  $\quad (return\ fg\ q,Ms,Ps) \in S\text{-}cs\ fg\ k;\ P' \subseteq\# P+Ps;\ size\ P' \le k\ ]\!]$
  $\quad \Longrightarrow (v,mon\ fg\ q \cup M \cup Ms,P') \in S\text{-}cs\ fg\ k$
  $|\ S\text{-}spawn$: $[\![(u,Spawn\ q,v) \in edges\ fg;\ (u,M,P) \in S\text{-}cs\ fg\ k;$
  $\quad P' \subseteq\# \{\#q\#\}+P;\ size\ P' \le k]\!]$
  $\quad \Longrightarrow (v,M,P') \in S\text{-}cs\ fg\ k$

The intuition underlying this constraint system is the following: The *S-init*-constraint describes that the procedures entry node can be reached with the empty path, that has no monitors and spawns no procedures. The *S-base*-constraint describes that executing a base edge does not use monitors or spawn threads, so each path reaching the start node of the base edge also induces a path reaching the end node of the base edge with the same set of monitors and the same set of spawned threads. The *S-call*-constraint models the effect of a procedure call. If there is a path to the start node of a call edge and a same-level path through the procedure, this also induces a path to the end node of the call edge. This path uses the monitors of both path and spawns the threads that are spawned on both paths. Since we only record a limited subset of the spawned threads, we have to choose which of the threads are recorded. The *S-spawn*-constraint models the effect of a spawn edge. A path to the start node of the spawn edge induces a path to the end node that uses the same set of monitors and spawns the threads of the initial path plus the one spawned by the spawn edge. We again have to choose which of these threads are recorded.

9.1.2 Soundness and Precision

Soundness of the constraint system *S-cs* means, that every same-level path has a corresponding entry in the constraint system.

As usual the soundness proof works by induction over the length of execution paths. The base case (empty path) trivially follows from the *S-init* constraint. In the inductive case, we consider the edge that induces the last step of the path; for a return step, this is the corresponding call edge (cf. Lemma *flowgraph.trss-find-call'*). With the induction hypothesis, we get the soundness for the (shorter) prefix of the path, and depending on the last step we can choose a constraint that implies soundness for the whole path.

lemma (**in** *flowgraph*) *S-sound*: !!*p v c' P*.
⟦((([*entry fg p*],{#}),*w*,([*v*],*c'*))∈*trcl* (*trss fg*);
size P≤*k*; (λ*p*. [*entry fg p*]) '# *P* ⊆# *c'* ⟧
⟹ (*v*,*mon-w fg w*,*P*)∈*S-cs fg k*
proof (*induct w rule*: *length-compl-rev-induct*)
case *Nil* **thus** *?case* **by** (*auto intro*: *S-init*)
next
case (*snoc w e*) **then obtain** *sh ch* **where** *SPLIT*: ((([*entry fg p*],{#}),*w*,(*sh*,*ch*))∈*trcl* (*trss fg*) ((*sh*,*ch*),*e*,([*v*],*c'*))∈*trss fg* **by** (*fast dest*: *trcl-rev-uncons*)
from *SPLIT*(*2*) **show** *?case* **proof** (*cases rule*: *trss.cases*)
case *trss-base* **then obtain** *u a* **where** *CASE*: *e*=*LBase a sh*=[*u*] *ch*=*c'* (*u*,*Base a*,*v*)∈*edges fg* **by** *auto*
with *snoc.hyps*[*of w p u c'*, *OF - - snoc.prems*(*2*,*3*)] *SPLIT*(*1*) **have** (*u*,*mon-w fg w*,*P*)∈*S-cs fg k* **by** *blast*
moreover from *CASE*(*1*) **have** *mon-e fg e* = {} **by** *simp*
ultimately show *?thesis* **using** *S-base*[*OF CASE*(*4*)] **by** (*auto simp add*: *mon-w-unconc*)
next
case *trss-ret* **then obtain** *q* **where** *CASE*: *e*=*LRet sh*=*return fg q*#[*v*] *ch*=*c'* **by** *auto*
with *SPLIT*(*1*) **have** (([*entry fg p*], {#}), *w*, [*return fg q*,*v*], *c'*) ∈ *trcl* (*trss fg*) **by** *simp*
from *trss-find-call'*[*OF this*] **obtain** *ut ct w1 w2* **where** *FC*:
w=*w1*@*LCall q*#*w2*
(([*entry fg p*],{#}),*w1*,([*ut*],*ct*))∈*trcl* (*trss fg*)
(([*ut*],*ct*),*LCall q*,([*entry fg q*,*v*],*ct*))∈*trss fg*
(*ut*,*Call q*,*v*)∈*edges fg*
(([*entry fg q*],*ct*),*w2*,([*return fg q*],*c'*))∈*trcl* (*trss fg*) **.**
from *trss-drop-all-context*[*OF FC*(*5*)] **obtain** *csp'* **where** *SLP*: *c'*=*ct*+*csp'* (([*entry fg q*],{#}),*w2*,([*return fg q*],*csp'*))∈*trcl* (*trss fg*) **by** (*auto simp add*: *union-ac*)
from *FC*(*1*) **have** *LEN*: *length w1* ≤ *length w length w2* ≤ *length w* **by** *auto*
from *mset-map-split-orig-le SLP*(*1*) *snoc.prems*(*3*) **obtain** *P1 P2* **where** *PSPLIT*: *P*=*P1*+*P2* (λ*p*. [*entry fg p*]) '# *P1* ⊆# *ct* (λ*p*. [*entry fg p*]) '# *P2* ⊆# *csp'* **by** *blast*
with *snoc.prems*(*2*) **have** *PSIZE*: *size P1* ≤ *k size P2* ≤ *k* **by** *auto*
from *snoc.hyps*[*OF LEN*(*1*) *FC*(*2*) *PSIZE*(*1*) *PSPLIT*(*2*)] *snoc.hyps*[*OF LEN*(*2*) *SLP*(*2*) *PSIZE*(*2*) *PSPLIT*(*3*)] **have** *IHAPP*: (*ut*, *mon-w fg w1*, *P1*) ∈ *S-cs fg k*

(*return fg q*, *mon-w fg w2*, *P2*) ∈ *S-cs fg k* **.**
  from *S-call*[*OF FC*(*4*) *IHAPP subset-mset.eq-refl*[*OF PSPLIT*(*1*)] *snoc.prems*(*2*)] *FC*(*1*) *CASE*(*1*) **show** (*v*, *mon-w fg* (*w*@[*e*]), *P*) ∈ *S-cs fg k* **by** (*auto simp add*: *mon-w-unconc Un-ac*)
  next
  case *trss-spawn* **then obtain** *u q* **where** *CASE*: *e*=*LSpawn q sh*=[*u*] *c′*={#[*entry fg q*]#}+*ch* (*u*,*Spawn q*,*v*)∈*edges fg* **by** *auto*
    from *mset-map-split-orig-le CASE*(*3*) *snoc.prems*(*3*) **obtain** *P1 P2* **where** *PSPLIT*: *P*=*P1*+*P2* (λ*p*. [*entry fg p*]) ‘# *P1* ⊆# {#[*entry fg q*]#} (λ*p*. [*entry fg p*]) ‘# *P2* ⊆# *ch* **by** *blast*
    with *snoc.prems*(*2*) **have** *PSIZE*: *size P2* ≤ *k* **by** *simp*
    from *snoc.hyps*[*OF - - PSIZE PSPLIT*(*3*)] *SPLIT*(*1*) *CASE*(*2*) **have** *IHAPP*: (*u*,*mon-w fg w*,*P2*)∈*S-cs fg k* **by** *blast*
    have *PCOND*: *P* ⊆# {#*q*#}+*P2* **proof** −
      from *PSPLIT*(*2*) **have** *P1*⊆#{#*q*#} **by** (*auto elim*!: *mset-le-single-cases mset-map-single-rightE*)
      with *PSPLIT*(*1*) **show** *?thesis* **by** *simp*
    qed
    from *S-spawn*[*OF CASE*(*4*) *IHAPP PCOND snoc.prems*(*2*)] *CASE*(*1*) **show** (*v*, *mon-w fg* (*w* @ [*e*]), *P*) ∈ *S-cs fg k* **by** (*auto simp add*: *mon-w-unconc*)
  qed
qed

Precision means that all entries appearing in the smallest solution of the constraint system are justified by some path in the operational characterization. For proving precision, one usually shows that a family of sets derived as an abstraction from the operational characterization solves all constraints.

In our formalization of constraint systems as inductive sets this amounts to constructing for each constraint a justifying path for the entries described on the conclusion side of the implication – under the assumption that corresponding paths exists for the entries mentioned in the antecedent.

lemma (**in** *flowgraph*) *S-precise*: (*v*,*M*,*P*)∈*S-cs fg k*
  ⟹ ∃ *p c′ w*.
      (([*entry fg p*],{#}),*w*,([*v*],*c′*))∈*trcl* (*trss fg*) ∧
      size P≤*k* ∧
      (λ*p*. [*entry fg p*]) ‘# *P* ⊆# *c′* ∧
      M=*mon-w fg w*
proof (*induct rule*: *S-cs.induct*)
  case (*S-init p*) **have** (([*entry fg p*],{#}),[],([*entry fg p*],{#}))∈*trcl* (*trss fg*) **by** *simp-all*
  thus *?case* **by** *fastforce*
next
  case (*S-base u a v M P*) **then obtain** *p c′ w* **where** *IHAPP*: (([*entry fg p*], {#}), *w*, [*u*], *c′*) ∈ *trcl* (*trss fg*) *size P* ≤ *k* (λ*p*. [*entry fg p*]) ‘# *P* ⊆# *c′ M* = *mon-w fg w* **by** *blast*
  note *IHAPP*(*1*)
  also from *S-base* **have** (([*u*],*c′*),*LBase a*,([*v*],*c′*))∈*trss fg* **by** (*auto intro*: *trss-base*)
  finally have (([*entry fg p*], {#}), *w* @ [*LBase a*], [*v*], *c′*) ∈ *trcl* (*trss fg*) **.**

moreover from *IHAPP(4)* **have** *M=mon-w fg (w @ [LBase a])* **by** *(simp add: mon-w-unconc)*
ultimately show *?case* **using** *IHAPP(2,3,4)* **by** *blast*
next
case *(S-call u q v M P Ms Ps P′)* **then obtain** *p csp1 w1* **where** *REACHING-PATH: (([entry fg p], {#}), w1, [u], csp1) ∈ trcl (trss fg) size P ≤ k (λp. [entry fg p]) ‘# P ⊆# csp1 M = mon-w fg w1* **by** *blast*
from *S-call* **obtain** *csp2 w2* **where** *SL-PATH: (([entry fg q], {#}), w2, [return fg q], csp2) ∈ trcl (trss fg) size Ps ≤ k (λp. [entry fg p]) ‘# Ps ⊆# csp2 Ms = mon-w fg w2*
by *(blast dest: trss-er-path-proc-const)*
from *trss-c-no-mon[OF REACHING-PATH(1)] trss-c-no-mon[OF SL-PATH(1)]* **have** *NOMON: mon-c fg csp1 = {} mon-c fg csp2 = {}* **by** *auto*
have *(([entry fg p], {#}), w1@LCall q#w2@[LRet],([v],csp1+csp2))∈trcl (trss fg)* **proof** −
note *REACHING-PATH(1)*
also from *trss-call[OF S-call(1)] NOMON* **have** *(([u],csp1),LCall q,([entry fg q,v],csp1))∈trss fg* **by** *(auto)*
also from *trss-add-context[OF trss-stack-comp[OF SL-PATH(1)]] NOMON* **have** *(([entry fg q,v],csp1),w2,([return fg q,v],csp1+csp2))∈trcl (trss fg)* **by** *(simp add: union-ac)*
also have *(([return fg q,v],csp1+csp2),LRet,([v],csp1+csp2))∈trss fg* **by** *(rule trss-ret)*
finally show *?thesis* **by** *simp*
qed
moreover from *REACHING-PATH(4) SL-PATH(4)* **have** *mon fg q ∪ M ∪ Ms = mon-w fg (w1@LCall q#w2@[LRet])* **by** *(auto simp add: mon-w-unconc)*
moreover have *(λp. [entry fg p]) ‘# (P′) ⊆# csp1+csp2* **(is** *?f ‘# P′ ⊆# -)* **proof** −
from *image-mset-subseteq-mono[OF S-call(6)]* **have** *?f ‘# P′ ⊆# ?f ‘# P + ?f ‘# Ps* **by** *auto*
also from *mset-subset-eq-mono-add[OF REACHING-PATH(3) SL-PATH(3)]* **have** *. . . ⊆# csp1+csp2* **.**
finally show *?thesis* **.**
qed
moreover note *S-call(7)*
ultimately show *?case* **by** *blast*
next
case *(S-spawn u q v M P P′)* **then obtain** *p c′ w* **where** *IHAPP: (([entry fg p], {#}), w, [u], c′) ∈ trcl (trss fg) size P ≤ k (λp. [entry fg p]) ‘# P ⊆# c′ M = mon-w fg w* **by** *blast*
note *IHAPP(1)*
also from *S-spawn(1)* **have** *(([u],c′),LSpawn q,([v],add-mset [entry fg q] c′))∈trss fg* **by** *(rule trss-spawn)*
finally have *(([entry fg p], {#}), w @ [LSpawn q], [v], add-mset [entry fg q] c′) ∈ trcl (trss fg)* **.**
moreover from *IHAPP(4)* **have** *M=mon-w fg (w @ [LSpawn q])* **by** *(simp add: mon-w-unconc)*
moreover have *(λp. [entry fg p]) ‘# P′ ⊆# {#[entry fg q]#} + c′* **(is** *?f ‘# -*

$\subseteq\#$ -) **proof** –
from *image-mset-subseteq-mono*[*OF S-spawn*(*4*)] **have** *?f* ‘# *P'* $\subseteq\#$ {#[*entry fg q*]#} + *?f* ‘# *P* **by** *auto*
also from *mset-subset-eq-mono-add*[*OF* - *IHAPP*(*3*)] **have** ... $\subseteq\#$ {#[*entry fg q*]#} + *c'* **by** (*auto intro*: *IHAPP*(*3*))
finally show *?thesis* **.**
qed
moreover note *S-spawn*(*5*)
ultimately show *?case* **by** *auto*
qed

— Finally we can state the soundness and precision as a single theorem
theorem (**in** *flowgraph*) *S-sound-precise*:
$(v,M,P)\in S\text{-}cs\ fg\ k \longleftrightarrow$
$(\exists\, p\ c'\ w.\ (([entry\ fg\ p],\{\#\}),w,([v],c'))\in trcl\ (trss\ fg) \wedge$
$size\ P\leq k \wedge (\lambda p.\ [entry\ fg\ p])\ `\#\ P \subseteq\#\ c' \wedge M{=}mon\text{-}w\ fg\ w)$
using *S-sound S-precise* **by** *blast*

Next, we present specialized soundness and precision lemmas, that reason over a macrostep (*ntrp fg*) rather than a same-level path (*trcl* (*trss fg*)). They are tailored for the use in the soundness and precision proofs of the other constraint systems.

lemma (**in** *flowgraph*) *S-sound-ntrp*:
assumes *A*: $(([u],\{\#\}),eel,(sh,ch))\in ntrp\ fg$ **and**
CASE: $!!p\ u'\ v\ w.\ [\![$
$eel{=}LOC\ (LCall\ p\#w);$
$(u,Call\ p,u')\in edges\ fg;$
$sh{=}[v,u'];$
$proc\text{-}of\ fg\ v = p;$
$mon\text{-}c\ fg\ ch = \{\};$
$!!s.\ s \in\#\ ch \Longrightarrow \exists\, p\ u\ v.\ s{=}[entry\ fg\ p] \wedge$
$(u,Spawn\ p,v)\in edges\ fg \wedge$
$initialproc\ fg\ p;$
$!!P.\ (\lambda p.\ [entry\ fg\ p])\ `\#\ P \subseteq\#\ ch \Longrightarrow$
$(v,mon\text{-}w\ fg\ w,P)\in S\text{-}cs\ fg\ (size\ P)$
$]\!] \Longrightarrow Q$
shows *Q*
proof –
from *A* **obtain** *ee* **where** *EE*: $eel{=}LOC\ ee\ (([u],\{\#\}),ee,(sh,ch))\in ntrs\ fg$ **by** (*auto elim*: *gtrp.cases*)
have *CHFMT*: $!!s.\ s \in\#\ ch \Longrightarrow \exists\, p\ u\ v.\ s{=}[entry\ fg\ p] \wedge (u,Spawn\ p,v)\in edges\ fg \wedge initialproc\ fg\ p$ **by** (*auto intro*: *ntrs-c-cases-s*[*OF EE*(*2*)])
with *c-of-initial-no-mon* **have** *CHNOMON*: $mon\text{-}c\ fg\ ch = \{\}$ **by** *blast*
from *EE*(*2*) **obtain** *p u' v w* **where** *FIRSTSPLIT*: $ee{=}LCall\ p\#w\ (([u],\{\#\}),LCall\ p,([entry\ fg\ p,u'],\{\#\}))\in trss\ fg\ sh{=}[v,u']\ (([entry\ fg\ p],\{\#\}),w,([v],ch))\in trcl\ (trss\ fg)$ **by** (*auto elim*!: *ntrs.cases*[*simplified*])
from *FIRSTSPLIT* **have** *EDGE*: $(u,Call\ p,u')\in edges\ fg$ **by** (*auto elim*!: *trss.cases*)
from *trss-bot-proc-const*[**where** $s{=}[]$ **and** $s'{=}[]$, *simplified*, *OF FIRSTSPLIT*(*4*)] **have** *PROC-OF-V*: $proc\text{-}of\ fg\ v = p$ **by** *simp*

have !!P. (λp. [*entry fg p*]) '# $P \subseteq$# *ch* $\Longrightarrow$ (v,*mon-w fg w*,P)$\in$*S-cs fg* (*size P*)
proof –
  fix *P* **assume** (λp. [*entry fg p*]) '# $P \subseteq$# *ch*
  from *S-sound*[*OF FIRSTSPLIT*(*4*) - *this*, *of size P*] **show** *?thesis P* **by** *simp*
qed
with *EE*(*1*) *FIRSTSPLIT*(*1*,*3*) *EDGE PROC-OF-V CHNOMON CHFMT* **show** *Q* **by** (*rule-tac CASE*) *auto*
qed

lemma (**in** *flowgraph*) *S-precise-ntrp*:
  assumes *ENTRY*: (v,M,P)$\in$*S-cs fg k* **and**
    P: *proc-of fg v* = *p* **and**
    EDGE: (u,*Call p*,u')$\in$*edges fg*
  shows $\exists$ *w ch*.
    (([*u*],{#}),*LOC* (*LCall p#w*),([*v*,*u'*],*ch*))$\in$*ntrp fg* $\wedge$
    size P $\leq$ *k* $\wedge$
    M=*mon-w fg w* $\wedge$
    mon-n fg v = *mon fg p* $\wedge$
    (λp. [*entry fg p*]) '# $P \subseteq$# *ch* $\wedge$
    mon-c fg ch={}
proof –
  from *P S-precise*[*OF ENTRY*, *simplified*] *trss-bot-proc-const*[**where** *s*=[] **and** *s'*=[], *simplified*] **obtain** *wsl ch* **where**
    SLPATH: (([*entry fg p*], {#}), *wsl*, [*v*], *ch*) $\in$ *trcl* (*trss fg*) *size P* $\leq$ *k* (λp. [*entry fg p*]) '# $P \subseteq$# *ch M* = *mon-w fg wsl* **by** *fastforce*
  from *mon-n-same-proc*[*OF trss-bot-proc-const*[**where** *s*=[] **and** *s'*=[], *simplified*, *OF SLPATH*(*1*)]] **have** *MON-V*: *mon-n fg v* = *mon fg p* **by** (*simp*)
  from *trss-c-cases*[*OF SLPATH*(*1*), *simplified*] **have** *CHFMT*: $\bigwedge$*s*. *s* $\in$# *ch* $\Longrightarrow$ $\exists$ *p*. *s* = [*entry fg p*] $\wedge$ ($\exists$ *u v*. (*u*, *Spawn p*, *v*) $\in$ *edges fg*) $\wedge$ *initialproc fg p* **by** *blast*
  with *c-of-initial-no-mon* **have** *CHNOMON*: *mon-c fg ch* = {} **by** *blast*
  — From the constraints prerequisites, we can construct the first step
  have *FS*: (([*u*],{#}),*LCall p#wsl*,([*v*,*u'*],*ch*))$\in$*ntrs fg* **proof** (*rule ntrs-step*[**where** *r*=[], *simplified*])
    from *EDGE* **show** (([*u*], {#}), *LCall p*, [*entry fg p*, *u'*], {#}) $\in$ *trss fg* **by** (*auto intro*: *trss-call*)
  qed (*rule SLPATH*(*1*))
  hence *FSP*: (([*u*],{#}),*LOC* (*LCall p#wsl*),([*v*,*u'*],*ch*))$\in$*ntrp fg* **by** (*blast intro*: *gtrp-loc*)
  from *FSP SLPATH*(*2*,*3*,*4*) *CHNOMON MON-V* **show** *?thesis* **by** *blast*
qed

9.2 Single reaching path

In this section we define a constraint system that collects abstract information of paths reaching a control node at *U*. The path starts with a single initial thread. The collected information are the monitors used by the steps of the initial thread, the monitors used by steps of other threads and the acquisition history of the path. To distinguish the steps of the initial thread from steps of other threads, we use the loc/env-semantics (cf. Section 5.4).

9.2.1 Constraint system

An element $(u, Ml, Me, h) \in \mathit{RU\text{-}cs}\ fg\ U$ corresponds to a path from $\{\#[u]\#\}$ to some configuration at U, that uses monitors from Ml in the steps of the initial thread, monitors from Me in the steps of other threads and has acquisition history h.

Here, the correspondence between paths and entries included into the inductively defined set is not perfect but strong enough for our purposes: While each constraint system entry corresponds to a path, not each path corresponds to a constraint system entry. But for each path reaching a configuration at U, we find an entry with less or equal monitors and an acquisition history less or equal to the acquisition history of the path.

inductive-set
  RU-cs :: $('n,'p,'ba,'m,'more)$ *flowgraph-rec-scheme* $\Rightarrow$ $'n$ *set* $\Rightarrow$
    $('n \times 'm\ set \times 'm\ set \times ('m \Rightarrow 'm\ set))$ *set*
  for *fg U*
  where
    RU-init: $u{\in}U \Longrightarrow (u,\{\},\{\},\lambda x.\{\}){\in}\mathit{RU\text{-}cs}\ fg\ U$
  | *RU-call*: $[\![\ (u,\mathit{Call}\ p,u'){\in}\mathit{edges}\ fg;\ \mathit{proc\text{-}of}\ fg\ v = p;\ (v,M,P){\in}\mathit{S\text{-}cs}\ fg\ 0;$
      $(v,Ml,Me,h){\in}\mathit{RU\text{-}cs}\ fg\ U;\ \mathit{mon\text{-}n}\ fg\ u \cap Me = \{\}\ ]\!]$
    $\Longrightarrow (\ u,\ \mathit{mon}\ fg\ p \cup M \cup Ml,\ Me,\ \mathit{ah\text{-}update}\ h\ (\mathit{mon}\ fg\ p,M)\ (Ml{\cup}Me))$
      $\in \mathit{RU\text{-}cs}\ fg\ U$
  | *RU-spawn*: $[\![\ (u,\mathit{Call}\ p,u'){\in}\mathit{edges}\ fg;\ \mathit{proc\text{-}of}\ fg\ v = p;\ (v,M,P){\in}\mathit{S\text{-}cs}\ fg\ 1;$
      $q \in\#\ P;\ (\mathit{entry}\ fg\ q,Ml,Me,h){\in}\mathit{RU\text{-}cs}\ fg\ U;$
      $(\mathit{mon\text{-}n}\ fg\ u \cup \mathit{mon}\ fg\ p) \cap (Ml \cup Me){=}\{\}\ ]\!]$
    $\Longrightarrow (u,\mathit{mon}\ fg\ p \cup M,\ Ml \cup Me,\ \mathit{ah\text{-}update}\ h\ (\mathit{mon}\ fg\ p,M)\ (Ml{\cup}Me))$
      $\in \mathit{RU\text{-}cs}\ fg\ U$

The constraint system works by tracking only a single thread. Initially, there is just one thread, and from this thread we reach a configuration at U. After a macrostep, we have the transformed initial thread and some spawned threads. The key idea is, that the actual node U is reached by just one of these threads. The steps of the other threads are useless for reaching U. Because of the nice properties of normalized paths, we can simply prune those steps from the path.

The *RU-init*-constraint reflects that we can reach a control node from itself with the empty path. The *RU-call*-constraint describes the case that U is reached from the initial thread, and the *RU-spawn*-constraint describes the case that U is reached from one of the spawned threads. In the two latter cases, we have to check whether prepending the macrostep to the reaching path is allowed or not due to monitor restrictions. In the call case, the procedure of the initial node must not own monitors that are used in the environment steps of the appended reaching path ($\mathit{mon\text{-}n}\ fg\ u \cap Me$

$= \{\}$). As we only test disjointness with the set of monitors used by the environment, reentrant monitors can be handled. In the spawn case, we have to check disjointness with both, the monitors of local and environment steps of the reaching path from the spawned thread, because from the perspective of the initial thread, all these steps are environment steps (($mon\text{-}n\ fg\ u \cup mon\ fg\ p) \cap (Ml \cup Me)=\{\}$). Note that in the call case, we do not need to explicitly check that the monitors used by the environment are disjoint from the monitors acquired by the called procedure because this already follows from the existence of a reaching path, as the starting point of this path already holds all these monitors.

However, in the spawn case, we have to check for both the monitors of the start node and of the called procedure to be compatible with the already known reaching path from the entry node of the spawned thread.

9.2.2 Soundness and precision

The following lemma intuitively states: *If we can reach a configuration that is at U from some start configuration, then there is a single thread in the start configuration that can reach a configuration at U with a subword of the original path.*

The proof follows from Lemma *flowgraph.ntr-reverse-split* rather directly.

lemma (**in** *flowgraph*) *ntr-reverse-split-atU*:
  assumes *V*: *valid fg c* **and**
    A: *atU U c′* **and**
    B: $(c,w,c') \in trcl\ (ntr\ fg)$
  shows $\exists\, s\ w'\ c1'.$
    $s \in\#\ c \wedge w' \preceq w \wedge c1' \subseteq\#\ c' \wedge$
    $atU\ U\ c1' \wedge (\{\#s\#\},w',c1') \in trcl\ (ntr\ fg)$
proof –
  obtain *ui r ce′* **where** *C′FMT*: $c'=\{\#ui\#r\#\}+ce'\ ui \in U$ **by** (*rule atU-fmt*[*OF A*], *simp only*: *mset-contains-eq*) (*blast dest*: *sym*)
  with *ntr-reverse-split*[*OF - V*] *B* **obtain** *s ce w1 w2 ce1′ ce2′* **where** *RSPLIT*: $c=\{\#s\#\}+ce\ ce'=ce1'+ce2'\ w \in w1 \otimes_{\alpha n\ fg} w2\ (\{\#s\#\},\ w1,\ \{\#ui\#r\#\} + ce1') \in trcl\ (ntr\ fg)$ **by** *blast*
  with *C′FMT* **have** $s \in\#\ c\ w1 \preceq w\ \{\#ui\#r\#\}+ce1' \subseteq\#\ c'\ atU\ U\ (\{\#ui\#r\#\}+ce1')$ **by** (*auto dest*: *cil-ileq*)
  with *RSPLIT*(*4*) **show** *?thesis* **by** *blast*
qed

The next lemma shows the soundness of the RU constraint system.

The proof works by induction over the length of the reaching path. For the empty path, the proposition follows by the *RU-init*-constraint. For a non-empty path, we consider the first step. It has transformed the initial thread and may have spawned some other threads. From the resulting configuration, *U* is reached. Due to *flowgraph.ntr-split* we get two interleavable paths from the rest of the original path, one from the transformed initial thread and one from the spawned threads. We then distinguish two cases: if the first path reaches U, the proposition follows by the induction hypothesis and the *RU-call* constraint.

Otherwise, we use *flowgraph.ntr-reverse-split-atU* to identify the thread that actually reaches U among all the spawned threads. Then we apply the induction hypothesis to the path of that thread and prepend the first step using the *RU-spawn*-constraint.

The main complexity of the proof script below results from fiddling with the monitors and converting between the multiset-and loc/env-semantics. Also the arguments to show that the acquisition histories are sound approximations require some space.

lemma (**in** *flowgraph*) *RU-sound*:
  $!!u\ s'\ c'.\ [\![(([u],\{\#\}),w,(s',c'))\in trcl\ (ntrp\ fg);\ atU\ U\ (add\text{-}mset\ s'\ c')]\!]$
  $\Longrightarrow \exists\, Ml\ Me\ h.$
    $(u,Ml,Me,h)\in RU\text{-}cs\ fg\ U\ \wedge$
    $Ml \subseteq mon\text{-}loc\ fg\ w\ \wedge$
    $Me \subseteq mon\text{-}env\ fg\ w\ \wedge$
    $h \leq \alpha ah\ (map\ (\alpha nl\ fg)\ w)$
— The proof works by induction over the length of the reaching path
proof (*induct w rule*: *length-compl-induct*)
  — For a reaching path of length zero, the proposition follows immediately by the constraint *RU-init*
  case *Nil* **thus** *?case* **by** *auto* (*auto intro*!: *RU-init*)
next
  case (*Cons eel wwl*)
  — For a non-empty path, we regard the first step and the rest of the path
  then obtain *sh ch* **where** *SPLIT*:
    $(([u],\{\#\}),eel,(sh,ch))\in ntrp\ fg$
    $((sh,ch),wwl,(s',c'))\in trcl\ (ntrp\ fg)$
    by (*fast dest*: *trcl-uncons*)
  obtain $p\ u'\ v\ w$ **where**
    — The first step consists of an initial call and a same-level path
     FS-FMT: $eel = LOC\ (LCall\ p\ \#\ w)\ (u,\ Call\ p,\ u') \in edges\ fg\ sh = [v,\ u']\ proc\text{-}of\ fg\ v = p\ mon\text{-}c\ fg\ ch = \{\}$
    — The only environment threads after the first step are the threads that where spawned by the first step
    and *CHFMT*: $\bigwedge s.\ s \in\#\ ch \Longrightarrow \exists\, p\ u\ v.\ s{=}[entry\ fg\ p] \wedge (u,Spawn\ p,v)\in edges\ fg \wedge initialproc\ fg\ p$
    — For the same-level path, we find a corresponding entry in the *S-cs*-constraint system
    and *S-ENTRY-PAT*: $\bigwedge P.\ (\lambda p.\ [entry\ fg\ p])\ \text{`}\#\ P \subseteq\#\ ch \Longrightarrow (v,\ mon\text{-}w\ fg\ w,\ P) \in S\text{-}cs\ fg\ (size\ P)$
    by (*rule S-sound-ntrp*[*OF SPLIT*(*1*)]) *blast*
  from *ntrp-valid-preserve-s*[*OF SPLIT*(*1*)] **have** *HVALID*: $valid\ fg\ (\{\#sh\#\} + ch)$ **by** *simp*
  — We split the remaining path by the local thread and the spawned threads,

getting two interleavable paths, one from the local thread and one from the spawned threads

from *ntrp-split*[**where** *?c1.0={#}*, *simplified*, *OF SPLIT(2) ntrp-valid-preserve-s*[*OF SPLIT(1)*], *simplified*] **obtain** *w1 w2 c1′ c2′* **where**

LESPLIT:

wwl∈*w1*$\otimes_{\alpha nl\ fg}$ *map ENV w2*

c′ = c1′ + c2′

((*sh*, {#}), *w1*, *s′*, *c1′*) ∈ *trcl* (*ntrp fg*)

(*ch*, *w2*, *c2′*) ∈ *trcl* (*ntr fg*)

mon-ww fg (*map le-rem-s w1*) ∩ *mon-c fg ch* = {}

mon-ww fg w2 ∩ *mon-s fg sh* = {}

by *blast*

— We make a case distinction whether *U* was reached from the local thread or from the spawned threads

from *Cons.prems*(*2*) *LESPLIT*(*2*) **have** *atU U* (({#*s′*#}+*c1′*) + *c2′*) **by** (*auto simp add*: *union-ac*)

thus *?case* **proof** (*cases rule*: *atU-union-cases*)

case *left* — *U* was reached from the local thread

from *cil-ileq*[*OF LESPLIT*(*1*)] **have** *ILEQ*: *w1*⪯*wwl* **and** *LEN*: *length w1* ≤ *length wwl* **by** (*auto simp add*: *le-list-length*)

— We can cut off the bottom stack symbol from the reaching path (as always possible for normalized paths)

from *FS-FMT*(*3*) *LESPLIT*(*3*) *ntrp-stack-decomp*[*of v* [] [*u′*] {#} *w1 s′ c1′ fg*, *simplified*] **obtain** *v′ rr* **where** *DECOMP*: *s′*=*v′*#*rr*@[*u′*] ((([*v*],{#}),*w1*,(*v′*#*rr*,*c1′*))∈*trcl* (*ntrp fg*) **by** *auto*

— This does not affect the configuration being at *U*

from *atU-xchange-stack left DECOMP*(*1*) **have** *ATU*: *atU U* (*add-mset* (*v′*#*rr*) *c1′*) **by** *fastforce*

— Then we can apply the induction hypothesis to get a constraint system entry for the path

from *Cons.hyps*[*OF LEN DECOMP*(*2*) *ATU*] **obtain** *Ml Me h* **where** *IHAPP*: (*v*,*Ml*,*Me*,*h*)∈*RU-cs fg U Ml* ⊆ *mon-loc fg w1 Me* ⊆ *mon-env fg w1 h* ≤ *αah* (*map* (*αnl fg*) *w1*) **by** *blast*

— Next, we have to apply the constraint *RU-call*

from *S-ENTRY-PAT*[*of* {#}, *simplified*] **have** *S-ENTRY*: (*v*, *mon-w fg w*, {#}) ∈ *S-cs fg 0* **.**

have *MON-U-ME*: *mon-n fg u* ∩ *Me* = {} **proof** −

from *ntrp-mon-env-w-no-ctx*[*OF Cons.prems*(*1*)] **have** *mon-env fg wwl* ∩ *mon-n fg u* = {} **by** (*auto*)

with *mon-env-ileq*[*OF ILEQ*] *IHAPP*(*3*) **show** *?thesis* **by** *fast*

qed

from *RU-call*[*OF FS-FMT*(*2*,*4*) *S-ENTRY IHAPP*(*1*) *MON-U-ME*] **have** (*u*, *mon fg p* ∪ *mon-w fg w* ∪ *Ml*, *Me*, *ah-update h* (*mon fg p*, *mon-w fg w*) (*Ml* ∪ *Me*)) ∈ *RU-cs fg U* **.**

— Then we assemble the rest of the proposition, that are the monitor restrictions and the acquisition history restriction

moreover have *mon fg p* ∪ *mon-w fg w* ∪ *Ml* ⊆ *mon-loc fg* (*eel*#*wwl*) **using** *mon-loc-ileq*[*OF ILEQ*] *IHAPP*(*2*) *FS-FMT*(*1*) **by** *fastforce*

moreover have *Me* ⊆ *mon-env fg* (*eel*#*wwl*) **using** *mon-env-ileq*[*OF ILEQ*, *of fg*] *IHAPP*(*3*) **by** *auto*

moreover have *ah-update h* (*mon fg p*, *mon-w fg w*) (*Ml* ∪ *Me*) ≤ *αah* (*map* (*αnl fg*) (*eel#wwl*)) **proof** (*simp add*: *ah-update-cons*)

show *ah-update h* (*mon fg p*, *mon-w fg w*) (*Ml* ∪ *Me*) ≤ *ah-update* (*αah* (*map* (*αnl fg*) *wwl*)) (*αnl fg eel*) (*mon-pl* (*map* (*αnl fg*) *wwl*)) **proof** (*rule ah-update-mono*)

from *IHAPP*(*4*) **have** *h* ≤ *αah* (*map* (*αnl fg*) *w1*) **.**

also from *αah-ileq*[*OF le-list-map*[*OF ILEQ*]] **have** *αah* (*map* (*αnl fg*) *w1*) ≤ *αah* (*map* (*αnl fg*) *wwl*) **.**

finally show *h* ≤ *αah* (*map* (*αnl fg*) *wwl*) **.**

next

from *FS-FMT*(*1*) **show** (*mon fg p*, *mon-w fg w*) = *αnl fg eel* **by** *auto*

next

from *IHAPP*(*2*,*3*) **have** (*Ml* ∪ *Me*) ⊆ *mon-pl* (*map* (*αnl fg*) *w1*) **by** (*auto simp add*: *mon-pl-of-αnl*)

also from *mon-pl-ileq*[*OF le-list-map*[*OF ILEQ*]] **have** ... ⊆ *mon-pl* (*map* (*αnl fg*) *wwl*) **.**

finally show (*Ml* ∪ *Me*) ⊆ *mon-pl* (*map* (*αnl fg*) *wwl*) **.**

qed

qed

ultimately show *?thesis* **by** *blast*

next

case *right* — *U* was reached from the spawned threads

from *cil-ileq*[*OF LESPLIT*(*1*)] *le-list-length*[*of map ENV w2 wwl*] **have** *ILEQ*: *map ENV w2*⪯*wwl* **and** *LEN*: *length w2* ≤ *length wwl* **by** (*auto*)

from *HVALID* **have** *CHVALID*: *valid fg ch mon-s fg sh* ∩ *mon-c fg ch* = {} **by** (*auto simp add*: *valid-unconc*)

— We first identify the actual thread from that *U* was reached

from *ntr-reverse-split-atU*[*OF CHVALID*(*1*) *right LESPLIT*(*4*)] **obtain** *q wr cr′* **where** *RI*: [*entry fg q*] ∈# *ch wr*⪯*w2 cr′*⊆#*c2′ atU U cr′* ({#[*entry fg q*]#},*wr*,*cr′*)∈*trcl* (*ntr fg*) **by** (*blast dest*: *CHFMT*)

— In order to apply the induction hypothesis, we have to convert the reaching path to loc/env semantics

from *ntrs.gtr2gtrp*[**where** *c*={#}, *simplified*, *OF RI*(*5*)] **obtain** *sr′ cre′ wwr* **where** *RI-NTRP*: *cr′*=*add-mset sr′ cre′ wr*=*map le-rem-s wwr* (([*entry fg q*],{#}),*wwr*,(*sr′*,*cre′*))∈*trcl* (*ntrp fg*) **by** *blast*

from *LEN le-list-length*[*OF RI*(*2*)] *RI-NTRP*(*2*) **have** *LEN′*: *length wwr* ≤ *length wwl* **by** *simp*

— The induction hypothesis yields a constraint system entry

from *Cons.hyps*[*OF LEN′ RI-NTRP*(*3*)] *RI-NTRP*(*1*) *RI*(*4*) **obtain** *Ml Me h* **where** *IHAPP*: (*entry fg q*, *Ml*, *Me*, *h*)∈*RU-cs fg U Ml* ⊆ *mon-loc fg wwr Me* ⊆ *mon-env fg wwr h* ≤ *αah* (*map* (*αnl fg*) *wwr*) **by** *auto*

— We also have an entry in the same-level path constraint system that contains the thread from that *U* was reached

from *S-ENTRY-PAT*[*of* {#*q*#}, *simplified*] *RI*(*1*) **have** *S-ENTRY*: (*v*, *mon-w fg w*, {#*q*#}) ∈ *S-cs fg 1* **by** *auto*

— Before we can apply the *RU-spawn*-constraint, we have to analyze the monitors

have *MON-MLE-ENV*: *Ml* ∪ *Me* ⊆ *mon-env fg wwl* **proof** −

from *IHAPP*(*2*,*3*) **have** *Ml* ∪ *Me* ⊆ *mon-loc fg wwr* ∪ *mon-env fg wwr* **by** *auto*

also from *mon-ww-of-le-rem*[*symmetric*] *RI-NTRP*(*2*) **have** ... = *mon-ww fg wr* **by** *fastforce*

also from *mon-env-ileq*[*OF ILEQ*] *mon-ww-ileq*[*OF RI*(*2*)] **have** ... ⊆ *mon-env fg wwl* **by** *fastforce*

finally show *?thesis* **.**

qed

have *MON-UP-MLE*: (*mon-n fg u* ∪ *mon fg p*) ∩ (*Ml* ∪ *Me*) = {} **proof** −

from *ntrp-mon-env-w-no-ctx*[*OF SPLIT*(*2*)] *FS-FMT*(*3*,*4*) *edges-part*[*OF FS-FMT*(*2*)] **have** (*mon-n fg u* ∪ *mon fg p*) ∩ *mon-env fg wwl* = {} **by** (*auto simp add*: *mon-n-def*)

with *MON-MLE-ENV* **show** *?thesis* **by** *auto*

qed

— Finally we can apply the *RU-spawn*-constraint that yields us an entry for the reaching path from *u*

from *RU-spawn*[*OF FS-FMT*(*2*,*4*) *S-ENTRY* - *IHAPP*(*1*) *MON-UP-MLE*] **have** (*u*, *mon fg p* ∪ *mon-w fg w*, *Ml* ∪ *Me*, *ah-update h* (*mon fg p*, *mon-w fg w*) (*Ml* ∪ *Me*)) ∈ *RU-cs fg U* **by** *simp*

— Next we have to assemble the rest of the proposition

moreover have *mon fg p* ∪ *mon-w fg w* ⊆ *mon-loc fg* (*eel*#*wwl*) **using** *FS-FMT*(*1*) **by** *fastforce*

moreover have *Ml* ∪ *Me* ⊆ *mon-env fg* (*eel*#*wwl*) **using** *MON-MLE-ENV* **by** *auto*

moreover have *ah-update h* (*mon fg p*, *mon-w fg w*) (*Ml* ∪ *Me*) ≤ α*ah* (*map* (α*nl fg*) (*eel*#*wwl*)) — Only the proposition about the acquisition histories needs some more work

proof (*simp add*: *ah-update-cons*)

have *MAP-HELPER*: *map* (α*nl fg*) *wwr* ⪯ *map* (α*nl fg*) *wwl* **proof** −

from *RI-NTRP*(*2*) **have** *map* (α*nl fg*) *wwr* = *map* (α*n fg*) *wr* **by** (*simp add*: α*n*-α*nl*)

also from *le-list-map*[*OF RI*(*2*)] **have** ... ⪯ *map* (α*n fg*) *w2* **.**

also have ... = *map* (α*nl fg*) (*map ENV w2*) **by** *simp*

also from *le-list-map*[*OF ILEQ*] **have** ... ⪯ *map* (α*nl fg*) *wwl* **.**

finally show *?thesis* **.**

qed

show *ah-update h* (*mon fg p*, *mon-w fg w*) (*Ml* ∪ *Me*) ≤ *ah-update* (α*ah* (*map* (α*nl fg*) *wwl*)) (α*nl fg eel*) (*mon-pl* (*map* (α*nl fg*) *wwl*)) **proof** (*rule ah-update-mono*)

from *IHAPP*(*4*) **have** *h* ≤ α*ah* (*map* (α*nl fg*) *wwr*) **.**

also have ... ≤ α*ah* (*map* (α*nl fg*) *wwl*) **by** (*rule* α*ah-ileq*[*OF MAP-HELPER*])

finally show *h* ≤ α*ah* (*map* (α*nl fg*) *wwl*) **.**

next

from *FS-FMT*(*1*) **show** (*mon fg p*, *mon-w fg w*) = α*nl fg eel* **by** *simp*

next

from *IHAPP*(*2*,*3*) *mon-pl-ileq*[*OF MAP-HELPER*] **show** *Ml* ∪ *Me* ⊆ *mon-pl* (*map* (α*nl fg*) *wwl*) **by** (*auto simp add*: *mon-pl-of-*α*nl*)

qed

qed

ultimately show *?thesis* **by** *blast*

qed

qed

Now we prove a statement about the precision of the least solution. As in the precision proof of the *S-cs* constraint system, we construct a path for the entry on the conclusion side of each constraint, assuming that there already exists paths for the entries mentioned in the antecedent.

We show that each entry in the least solution corresponds exactly to some executable path, and is not just an under-approximation of a path; while for the soundness direction, we could only show that every executable path is under-approximated. The reason for this is that in effect, the constraint system prunes the steps of threads that are not needed to reach the control point. However, each pruned path is executable.

lemma (**in** *flowgraph*) *RU-precise*: $(u,Ml,Me,h) \in RU\text{-}cs\ fg\ U$
$\Longrightarrow \exists\, w\ s'\ c'.$
$(([u],\{\#\}),w,(s',c')) \in trcl\ (ntrp\ fg) \wedge$
$atU\ U\ (\{\#s'\#\}+c') \wedge$
$mon\text{-}loc\ fg\ w = Ml \wedge$
$mon\text{-}env\ fg\ w = Me \wedge$
$\alpha ah\ (map\ (\alpha nl\ fg)\ w) = h$
proof (*induct rule*: *RU-cs.induct*)
— The *RU-init* constraint is trivially covered by the empty path
case (*RU-init u*) **thus** *?case* **by** (*auto intro*: *exI*[*of* - []])
next
— Call constraint
case (*RU-call u p u′ v M P Ml Me h*)
then obtain $w\ s'\ c'$ **where** *IHAPP*: $(([v],\ \{\#\}),\ w,\ s',\ c') \in trcl\ (ntrp\ fg)\ atU\ U\ (\{\#s'\#\} + c')\ mon\text{-}loc\ fg\ w = Ml\ mon\text{-}env\ fg\ w = Me\ \alpha ah\ (map\ (\alpha nl\ fg)\ w) = h$ **by** *blast*
from *RU-call.hyps*(*2*) *S-precise*[*OF RU-call.hyps*(*3*), *simplified*] *trss-bot-proc-const*[**where** s=[] **and** s'=[], *simplified*] **obtain** *wsl ch* **where**
SLPATH: $(([entry\ fg\ p],\ \{\#\}),\ wsl,\ [v],\ ch) \in trcl\ (trss\ fg)\ M = mon\text{-}w\ fg\ wsl$ **by** *fastforce*
from *trss-c-cases*[*OF SLPATH*(*1*), *simplified*] **have** *CHFMT*: $\bigwedge s.\ s \in\#\ ch \Longrightarrow \exists\, p.\ s = [entry\ fg\ p] \wedge (\exists\, u\ v.\ (u,\ Spawn\ p,\ v) \in edges\ fg) \wedge initialproc\ fg\ p$ **by** *blast*
with *c-of-initial-no-mon* **have** *CHNOMON*: $mon\text{-}c\ fg\ ch = \{\}$ **by** *blast*
— From the constraints prerequisites, we can construct the first step
have *FS*: $(([u],\{\#\}),LCall\ p\#wsl,([v,u'],ch)) \in ntrs\ fg$ **proof** (*rule ntrs-step*[**where** r=[], *simplified*])
from *RU-call.hyps*(*1*) **show** $(([u],\ \{\#\}),\ LCall\ p,\ [entry\ fg\ p,\ u'],\ \{\#\}) \in trss\ fg$ **by** (*auto intro*: *trss-call*)
qed (*rule SLPATH*(*1*))
hence *FSP*: $(([u],\{\#\}),LOC\ (LCall\ p\#wsl),([v,u'],ch)) \in ntrp\ fg$ **by** (*blast intro*: *gtrp-loc*)
also
— The rest of the path comes from the induction hypothesis, after adding the rest of the threads to the context
have $(([v,\ u'],\ ch),\ w,\ s'\ @\ [u'],\ c' + ch) \in trcl\ (ntrp\ fg)$ **proof** (*rule ntrp-add-context*[*OF ntrp-stack-comp*[*OF IHAPP*(*1*), **where** r=[u']], **where** cn=ch, *simplified*])
from *RU-call.hyps*(*1,6*) *IHAPP*(*4*) **show** $mon\text{-}n\ fg\ u' \cap mon\text{-}env\ fg\ w = \{\}$ **by** (*auto simp add*: *mon-n-def edges-part*)

from *CHNOMON* **show** *mon-ww fg* (*map le-rem-s w*) ∩ *mon-c fg ch* = {} **by** *auto*
qed
finally have (([*u*], {#}), *LOC* (*LCall p* # *wsl*) # *w*, *s′* @ [*u′*], *c′* + *ch*) ∈ *trcl* (*ntrp fg*) **.**
— It is straightforward to show that the new path satisfies the required properties for its monitors and acquisition history
moreover from *IHAPP*(*2*) **have** *atU U* ({# *s′*@[*u′*] #}+(*c′*+*ch*)) **by** *auto*
moreover have *mon-loc fg* (*LOC* (*LCall p* # *wsl*) # *w*) = *mon fg p* ∪ *M* ∪ *Ml* **using** *SLPATH*(*2*) *IHAPP*(*3*) **by** *auto*
moreover have *mon-env fg* (*LOC* (*LCall p* # *wsl*) # *w*) = *Me* **using** *IHAPP*(*4*) **by** *auto*
moreover have *αah* (*map* (*αnl fg*) (*LOC* (*LCall p* # *wsl*) # *w*)) = *ah-update h* (*mon fg p*, *M*) (*Ml* ∪ *Me*) **proof** −
have *αah* (*map* (*αnl fg*) (*LOC* (*LCall p* # *wsl*) # *w*)) = *ah-update* (*αah* (*map* (*αnl fg*) *w*)) (*mon fg p*, *mon-w fg wsl*) (*mon-pl* (*map* (*αnl fg*) *w*)) **by** (*auto simp add*: *ah-update-cons*)
also have … = *ah-update h* (*mon fg p*, *M*) (*Ml* ∪ *Me*) **proof** −
from *IHAPP*(*5*) **have** *αah* (*map* (*αnl fg*) *w*) = *h* **.**
moreover from *SLPATH*(*2*) **have** (*mon fg p*, *mon-w fg wsl*) = (*mon fg p*, *M*) **by** (*simp add*: *mon-pl-of-αnl*)
moreover from *IHAPP*(*3*,*4*) **have** *mon-pl* (*map* (*αnl fg*) *w*) = *Ml* ∪ *Me* **by** (*auto simp add*: *mon-pl-of-αnl*)
ultimately show *?thesis* **by** *simp*
qed
finally show *?thesis* **.**
qed
ultimately show *?case* **by** *blast*
next
— Spawn constraint
case (*RU-spawn u p u′ v M P q Ml Me h*) **then obtain** *w s′ c′* **where** *IHAPP*: (([*entry fg q*], {#}), *w*, *s′*, *c′*) ∈ *trcl* (*ntrp fg*) *atU U* ({#*s′*#} + *c′*) *mon-loc fg w* = *Ml mon-env fg w* = *Me αah* (*map* (*αnl fg*) *w*) = *h* **by** *blast*
from *RU-spawn.hyps*(*2*) *S-precise*[*OF RU-spawn.hyps*(*3*), *simplified*] *trss-bot-proc-const*[**where** *s*=[] **and** *s′*=[], *simplified*] **obtain** *wsl ch* **where**
SLPATH: (([*entry fg p*], {#}), *wsl*, [*v*], *ch*) ∈ *trcl* (*trss fg*) *M* = *mon-w fg wsl* *size P* ≤ *1* (λ*p*. [*entry fg p*]) ‘# *P* ⊆# *ch* **by** *fastforce*
with *RU-spawn.hyps*(*4*) **obtain** *che* **where** *PFMT*: *P*={#*q*#} *ch* = {#[*entry fg q*]#} + *che* **by** (*auto elim*!: *mset-size-le1-cases mset-le-addE*)
from *trss-c-cases*[*OF SLPATH*(*1*), *simplified*] **have** *CHFMT*: ⋀*s*. *s* ∈# *ch* ⟹ ∃ *p*. *s* = [*entry fg p*] ∧ (∃ *u v*. (*u*, *Spawn p*, *v*) ∈ *edges fg*) ∧ *initialproc fg p* **by** *blast*
with *c-of-initial-no-mon* **have** *CHNOMON*: *mon-c fg ch* = {} **by** *blast*
have *FS*: (([*u*],{#}),*LCall p*#*wsl*,([*v*,*u′*],*ch*))∈*ntrs fg* **proof** (*rule ntrs-step*[**where** *r*=[], *simplified*])
from *RU-spawn.hyps*(*1*) **show** (([*u*], {#}), *LCall p*, [*entry fg p*, *u′*], {#}) ∈ *trss fg* **by** (*auto intro*: *trss-call*)
qed (*rule SLPATH*(*1*))
hence *FSP*: (([*u*],{#}),*LOC* (*LCall p*#*wsl*),([*v*,*u′*],*ch*))∈*ntrp fg* **by** (*blast intro*: *gtrp-loc*)

also have (([v, u'], ch), map ENV (map le-rem-s w), [v,u'], che+({#s'#}+c')) ∈ trcl (ntrp fg) **proof** −
from IHAPP(3,4) **have** mon-ww fg (map le-rem-s w) ⊆ Ml ∪ Me **by** (auto simp add: mon-ww-of-le-rem)
with RU-spawn.hyps(1,2,7) **have** (mon-n fg v ∪ mon-n fg u') ∩ mon-ww fg (map le-rem-s w) = {} **by** (auto simp add: mon-n-def edges-part)
with ntr2ntrp[OF gtrp2gtr[OF IHAPP(1)], of [v,u'] che] PFMT(2) CHNOMON **show** ?thesis **by** (auto simp add: union-ac mon-c-unconc)
qed
finally have (([u], {#}), LOC (LCall p # wsl) # map ENV (map le-rem-s w), [v, u'], che + ({#s'#} + c')) ∈ trcl (ntrp fg) **.**
moreover from IHAPP(2) **have** atU U ({#[v,u']#} + (che+({#s'#} + c'))) **by** auto
moreover have mon-loc fg (LOC (LCall p # wsl) # map ENV (map le-rem-s w)) = mon fg p ∪ M **using** SLPATH(2) **by** (auto simp del: map-map)
moreover have mon-env fg (LOC (LCall p # wsl) # map ENV (map le-rem-s w)) = Ml ∪ Me **using** IHAPP(3,4) **by** (auto simp add: mon-ww-of-le-rem simp del: map-map)
moreover have αah (map (αnl fg) (LOC (LCall p # wsl) # map ENV (map le-rem-s w))) = ah-update h (mon fg p, M) (Ml ∪ Me) **proof** −
have αah (map (αnl fg) (LOC (LCall p # wsl) # map ENV (map le-rem-s w))) = ah-update (αah (map (αn fg) (map le-rem-s w))) (mon fg p, mon-w fg wsl) (mon-pl (map (αn fg) (map le-rem-s w))) **by** (simp add: ah-update-cons o-assoc)
also have ... = ah-update h (mon fg p, M) (Ml ∪ Me) **proof** −
from IHAPP(5) **have** αah (map (αn fg) (map le-rem-s w)) = h **by** (simp add: αn-αnl)
moreover from SLPATH(2) **have** (mon fg p, mon-w fg wsl) = (mon fg p, M) **by** simp
moreover from IHAPP(3,4) **have** mon-pl (map (αn fg) (map le-rem-s w)) = Ml ∪ Me **by** (auto simp add: mon-pl-of-αnl αn-αnl)
ultimately show ?thesis **by** simp
qed
finally show ?thesis **.**
qed
ultimately show ?case **by** blast
qed

9.3 Simultaneously reaching path

In this section, we define a constraint system that collects abstract information for paths starting at a single control node and reaching two program points simultaneously, one from a set *U* and one from a set *V*.

9.3.1 Constraint system

An element (*u*, *Ml*, *Me*) ∈ *RUV-cs fg U V* means, that there is a path from {#[*u*]#} to some configuration that is simultaneously at *U* and at *V*. That path uses monitors from *Ml* in the first thread and monitors from *Me* in the other threads.

inductive-set
  $RUV\text{-}cs$ :: ('n,'p,'ba,'m,'more) *flowgraph-rec-scheme* ⇒
      'n set ⇒ 'n set ⇒ ('n × 'm set × 'm set) set
  for *fg U V*
where
   RUV-call:
    ⟦ (u,Call p,u')∈edges fg; proc-of fg v = p; (v,M,P)∈S-cs fg 0;
     (v,Ml,Me)∈RUV-cs fg U V; mon-n fg u ∩ Me = {} ⟧
    ⟹ (u,mon fg p ∪ M ∪ Ml,Me)∈RUV-cs fg U V
  | *RUV-spawn*:
    ⟦ (u,Call p,u')∈edges fg; proc-of fg v = p; (v,M,P)∈S-cs fg 1; q ∈# P;
     (entry fg q,Ml,Me)∈RUV-cs fg U V;
     (mon-n fg u ∪ mon fg p) ∩ (Ml ∪ Me) = {} ⟧
    ⟹ (u, mon fg p ∪ M, Ml∪Me)∈RUV-cs fg U V
  | *RUV-split-le*:
    ⟦ (u,Call p,u')∈edges fg; proc-of fg v = p; (v,M,P)∈S-cs fg 1; q ∈# P;
     (v,Ml,Me,h)∈RU-cs fg U; (entry fg q,Ml',Me',h')∈RU-cs fg V;
     (mon-n fg u ∪ mon fg p) ∩ (Me∪Ml'∪Me')={}; h [*] h' ⟧
    ⟹ (u, mon fg p ∪ M ∪ Ml, Me ∪ Ml' ∪ Me')∈RUV-cs fg U V
  | *RUV-split-el*:
    ⟦ (u,Call p,u')∈edges fg; proc-of fg v = p; (v,M,P)∈S-cs fg 1; q ∈# P;
     (v,Ml,Me,h)∈RU-cs fg V; (entry fg q,Ml',Me',h')∈RU-cs fg U;
     (mon-n fg u ∪ mon fg p) ∩ (Me∪Ml'∪Me')={}; h [*] h' ⟧
    ⟹ (u, mon fg p ∪ M ∪ Ml, Me ∪ Ml' ∪ Me')∈RUV-cs fg U V
  | *RUV-split-ee*:
    ⟦ (u,Call p,u')∈edges fg; proc-of fg v = p; (v,M,P)∈S-cs fg 2;
     {#q#}+{#q'#} ⊆# P;
     (entry fg q,Ml,Me,h)∈RU-cs fg U; (entry fg q',Ml',Me',h')∈RU-cs fg V;
     (mon-n fg u ∪ mon fg p) ∩ (Ml∪Me∪Ml'∪Me') = {}; h [*] h' ⟧
    ⟹ (u, mon fg p ∪ M, Ml∪Me∪Ml'∪Me')∈RUV-cs fg U V

The idea underlying this constraint system is similar to the *RU-cs*-constraint system for reaching a single node set. Initially, we just track one thread. After a macrostep, we have a configuration consisting of the transformed initial thread and the spawned threads. From this configuration, we reach two nodes simultaneously, one in U and one in V. Each of these nodes is reached by just a single thread. The constraint system contains one constraint for each case how these threads are related to the initial and the spawned threads:

RUV_call Both, U and V are reached from the initial thread.

RUV_spawn Both, U and V are reached from a single spawned thread.

RUV_split_le U is reached from the initial thread, V is reached from a spawned thread.

RUV_split_el V is reached from the initial thread, U is reached from a spawned thread.

RUV_split_ee Both, U and V are reached from different spawned threads.

In the latter three cases, we have to analyze the interleaving of two paths each reaching a single control node. This is done via the acquisition history information that we collected in the *RU-cs*-constraint system.

Note that we do not need an initializing constraint for the empty path, as a single configuration cannot simultaneously be at two control nodes.

9.3.2 Soundness and precision

lemma (**in** *flowgraph*) *RUV-sound*: !!*u s′ c′*.
⟦ $(([u],\{\#\}),w,(s',c'))\in trcl\ (ntrp\ fg)$; $atUV\ U\ V\ (\{\#s'\#\}+c')$ ⟧
$\Longrightarrow \exists\ Ml\ Me.$
$(u,Ml,Me)\in RUV\text{-}cs\ fg\ U\ V \wedge$
$Ml \subseteq mon\text{-}loc\ fg\ w \wedge$
$Me \subseteq mon\text{-}env\ fg\ w$
— The soundness proof is done by induction over the length of the reaching path
proof (*induct w rule*: *length-compl-induct*)
— In case of the empty path, a contradiction follows because a single-thread configuration cannot simultaneously be at two control nodes
case *Nil* **hence** *False* **by** *simp* **thus** *?case* **..**
next
case (*Cons ee ww*) **then obtain** *sh ch* **where** *SPLIT*: $(([u],\{\#\}),ee,(sh,ch))\in ntrp\ fg$ $((sh,ch),ww,(s',c'))\in trcl\ (ntrp\ fg)$ **by** (*fast dest*: *trcl-uncons*)
from *ntrp-split*[**where** *?c1.0*=$\{\#\}$, *simplified*, *OF SPLIT*(*2*) *ntrp-valid-preserve-s*[*OF SPLIT*(*1*)], *simplified*] **obtain** *w1 w2 c1′ c2′* **where**
LESPLIT: $ww \in w1 \otimes_{\alpha nl\ fg}$ *map ENV w2* $c' = c1' + c2'$ $((sh, \{\#\}), w1, s', c1') \in trcl\ (ntrp\ fg)$ $(ch, w2, c2') \in trcl\ (ntr\ fg)$ *mon-ww fg* (*map le-rem-s w1*) $\cap$ *mon-c fg ch* $= \{\}$ *mon-ww fg w2* $\cap$ *mon-s fg sh* $= \{\}$
by *blast*
obtain *p u′ v w* **where**
FS-FMT: $ee = LOC\ (LCall\ p\ \#\ w)$ $(u,\ Call\ p,\ u') \in edges\ fg\ sh = [v,\ u']$ *proc-of fg v = p mon-c fg ch* $= \{\}$
and *CHFMT*: $\bigwedge s.\ s \in\#\ ch \Longrightarrow \exists\ p\ u\ v.\ s = [entry\ fg\ p] \wedge (u,\ Spawn\ p,\ v) \in edges\ fg \wedge initialproc\ fg\ p$
and *S-ENTRY-PAT*: $\bigwedge P.\ (\lambda p.\ [entry\ fg\ p])$ ‘# $P \subseteq\#\ ch \Longrightarrow (v,\ mon\text{-}w\ fg\ w,\ P) \in S\text{-}cs\ fg\ (size\ P)$
by (*rule S-sound-ntrp*[*OF SPLIT*(*1*)]) *blast*
from *ntrp-mon-env-w-no-ctx*[*OF SPLIT*(*2*)] *FS-FMT*(*3*,*4*) *edges-part*[*OF FS-FMT*(*2*)] **have** *MON-PU*: *mon-env fg ww* $\cap$ (*mon fg p* $\cup$ *mon-n fg u*) $= \{\}$ **by** (*auto simp add*: *mon-n-def*)
from *cil-ileq*[*OF LESPLIT*(*1*)] *mon-loc-ileq*[*of w1 ww fg*] *mon-env-ileq*[*of w1 ww fg*] **have** *MON1-LEQ*: *mon-loc fg w1* $\subseteq$ *mon-loc fg ww mon-env fg w1* $\subseteq$ *mon-env fg ww* **by** *auto*
from *cil-ileq*[*OF LESPLIT*(*1*)] *mon-env-ileq*[*of map ENV w2 ww fg*] **have** *MON2-LEQ*: *mon-ww fg w2* $\subseteq$ *mon-env fg ww* **by** *simp*

from *LESPLIT*(*3*) *FS-FMT*(*3*) *ntrp-stack-decomp*[*of v* [] [*u'*] {#} *w1 s' c1'*, *simplified*] **obtain** *v' rr* **where** *DECOMP-LOC*: *s'=v'#rr@*[*u'*] ((([*v*],{#}),*w1*,(*v'#rr*,*c1'*))∈*trcl* (*ntrp fg*) **by** (*simp*, *blast*)

from *Cons.prems*(*2*) *LESPLIT*(*2*) **have** *atUV U V* (({#*s'*#}+*c1'*) + *c2'*) **by** (*simp add*: *union-ac*)

thus *?case* **proof** (*cases rule*: *atUV-union-cases*)

case *left* **with** *DECOMP-LOC*(*1*) **have** *ATUV*: *atUV U V* ({# *v'#rr* #}+*c1'*) **by** *simp*

from *Cons.hyps*[*OF - DECOMP-LOC*(*2*) *ATUV*] *cil-length*[*OF LESPLIT*(*1*)] **obtain** *Ml Me* **where** *IHAPP*: (*v*, *Ml*, *Me*) ∈ *RUV-cs fg U V Ml* ⊆ *mon-loc fg w1 Me* ⊆ *mon-env fg w1* **by** *auto*

from *RUV-call*[*OF FS-FMT*(*2*,*4*) *S-ENTRY-PAT*[*of* {#}, *simplified*] *IHAPP*(*1*)] **have** (*u*, *mon fg p* ∪ *mon-w fg w* ∪ *Ml*, *Me*) ∈ *RUV-cs fg U V* **using** *IHAPP*(*3*) *MON-PU MON1-LEQ* **by** *fastforce*

moreover have *mon fg p* ∪ *mon-w fg w* ∪ *Ml* ⊆ *mon-loc fg* (*ee#ww*) **using** *FS-FMT*(*1*) *IHAPP*(*2*) *MON1-LEQ* **by** *auto*

moreover have *Me* ⊆ *mon-env fg* (*ee#ww*) **using** *IHAPP*(*3*) *MON1-LEQ* **by** *auto*

ultimately show *?thesis* **by** *blast*

next

case *right* — Both nodes are reached from the spawned threads, we have to further distinguish whether both nodes are reached from the same thread or from different threads

then obtain *s1' s2'* **where** *R-STACKS*: {#*s1'*#}+{#*s2'*#} ⊆# *c2' atU-s U s1' atU-s V s2'* **by** (*unfold atUV-def*) *auto*

then obtain *ce2'* **where** *C2'FMT*: *c2'*={#*s1'*#}+({#*s2'*#}+*ce2'*) **by** (*auto simp add*: *mset-subset-eq-exists-conv union-ac*)

obtain *q ceh w21 w22 ce21' ce22'* **where**

REVSPLIT: *ch*={#[*entry fg q*]#}+*ceh add-mset s2' ce2'* = *ce21'*+*ce22'* *w2*∈*w21*$\otimes_{\alpha n\ fg}$*w22 mon fg q* ∩ (*mon-c fg ceh* ∪ *mon-ww fg w22*)={} *mon-c fg ceh* ∩ (*mon fg q* ∪ *mon-ww fg w21*) = {}

({#[*entry fg q*]#},*w21*,{#*s1'*#}+*ce21'*)∈*trcl* (*ntr fg*) (*ceh*,*w22*,*ce22'*)∈*trcl* (*ntr fg*)

proof −

from *ntr-reverse-split*[*of ch w2 s1'* {#*s2'*#}+*ce2'*] *ntrp-valid-preserve-s*[*OF SPLIT*(*1*), *simplified*] *C2'FMT LESPLIT*(*4*)

obtain *seh ceh w21 w22 ce21' ce22'* **where**

∗: *ch*={#*seh*#}+*ceh* {#*s2'*#}+*ce2'* = *ce21'*+*ce22'* *w2*∈*w21*$\otimes_{\alpha n\ fg}$*w22 mon-s fg seh* ∩ (*mon-c fg ceh* ∪ *mon-ww fg w22*)={} *mon-c fg ceh* ∩ (*mon-s fg seh* ∪ *mon-ww fg w21*) = {}

({#*seh*#},*w21*,{#*s1'*#}+*ce21'*)∈*trcl* (*ntr fg*) (*ceh*,*w22*,*ce22'*)∈*trcl* (*ntr fg*) **by** (*auto simp add*: *valid-unconc*)

from *this*(*1*) *CHFMT*[*of seh*] **obtain** *q* **where** *seh*=[*entry fg q*] **by** *auto*

with ∗ **have** *ch*={#[*entry fg q*]#}+*ceh add-mset s2' ce2'* = *ce21'*+*ce22'* *w2*∈*w21*$\otimes_{\alpha n\ fg}$*w22 mon fg q* ∩ (*mon-c fg ceh* ∪ *mon-ww fg w22*)={} *mon-c fg ceh* ∩ (*mon fg q* ∪ *mon-ww fg w21*) = {}

({#[*entry fg q*]#},*w21*,{#*s1'*#}+*ce21'*)∈*trcl* (*ntr fg*) (*ceh*,*w22*,*ce22'*)∈*trcl* (*ntr fg*) **by** *auto*

thus *thesis* **using** *that* **by** (*blast*)

qed
— For applying the induction hypothesis, it will be handy to have the reaching path in loc/env format:
from *ntrs.gtr2gtrp*[**where** *c*={#}, *simplified*, *OF REVSPLIT*(*6*)] **obtain** *sq′ csp-q ww21* **where**
R-CONV: *add-mset s1′ ce21′* = *add-mset sq′ csp-q w21* = *map le-rem-s ww21* (([*entry fg q*], {#}), *ww21*, *sq′*, *csp-q*) ∈ *trcl* (*ntrp fg*) **by** *auto*
from *cil-ileq*[*OF REVSPLIT*(*3*)] *mon-ww-ileq*[*of w21 w2 fg*] *mon-ww-ileq*[*of w22 w2 fg*] **have** *MON2N-LEQ*: *mon-ww fg w21* ⊆ *mon-ww fg w2 mon-ww fg w22* ⊆ *mon-ww fg w2* **by** *auto*
from *REVSPLIT*(*2*) **show** *?thesis* **proof** (*cases rule*: *mset-unplusm-dist-cases*[*case-names left′ right′*])
case *left′* — Both nodes are reached from the same thread
have *ATUV*: *atUV U V* ({#*sq′*#}+*csp-q*) **using** *right C2′FMT R-STACKS*(*2*,*3*) *left′*(*1*)
by (*metis R-CONV*(*1*) *add-mset-add-single atUV-union atU-add-mset union-commute*)
from *Cons.hyps*[*OF - R-CONV*(*3*) *ATUV*] *cil-length*[*OF REVSPLIT*(*3*)] *cil-length*[*OF LESPLIT*(*1*)] *R-CONV*(*2*) **obtain** *Ml Me* **where** *IHAPP*: (*entry fg q*, *Ml*, *Me*) ∈ *RUV-cs fg U V Ml* ⊆ *mon-loc fg ww21 Me* ⊆ *mon-env fg ww21* **by** *auto*
from *REVSPLIT*(*1*) *S-ENTRY-PAT*[*of* {#*q*#}, *simplified*] **have** *S-ENTRY*: (*v*, *mon-w fg w*, {#*q*#}) ∈ *S-cs fg 1* **by** *simp*
have *MON-COND*: (*mon-n fg u* ∪ *mon fg p*) ∩ (*Ml* ∪ *Me*) = {} **proof** −
from *R-CONV*(*2*) **have** *mon-ww fg w21* = *mon-loc fg ww21* ∪ *mon-env fg ww21* **by** (*simp add*: *mon-ww-of-le-rem*)
with *IHAPP*(*2*,*3*) *MON2N-LEQ*(*1*) *MON-PU MON2-LEQ* **show** *?thesis* **by** *blast*
qed
from *RUV-spawn*[*OF FS-FMT*(*2*) *FS-FMT*(*4*) *S-ENTRY - IHAPP*(*1*) *MON-COND*] **have** (*u*, *mon fg p* ∪ *mon-w fg w*, *Ml* ∪ *Me*) ∈ *RUV-cs fg U V* **by** *simp*
moreover have *mon fg p* ∪ *mon-w fg w* ⊆ *mon-loc fg* (*ee*#*ww*) **using** *FS-FMT*(*1*) **by** *auto*
moreover have *Ml* ∪ *Me* ⊆ *mon-env fg* (*ee*#*ww*) **using** *IHAPP*(*2*,*3*) *R-CONV*(*2*) *MON2N-LEQ*(*1*) *MON2-LEQ* **by** (*auto simp add*: *mon-ww-of-le-rem*)
ultimately show *?thesis* **by** *blast*
next

case *right′* — The nodes are reached from different threads
from *R-STACKS*(*2*,*3*) **have** *ATUV*: *atU U* (*add-mset sq′ csp-q*) *atU V ce22′* **by** (−) (*subst R-CONV*(*1*)[*symmetric*], *simp*, *subst right′*(*1*), *simp*)
— We have to reverse-split the second path again, to extract the second interesting thread
obtain *q′ w22′ ce22e′* **where** *REVSPLIT′*: [*entry fg q′*] ∈# *ceh w22′*⪯*w22 ce22e′* ⊆# *ce22′ atU V ce22e′* ({#[*entry fg q′*]#},*w22′*,*ce22e′*)∈*trcl* (*ntr fg*)
proof −
from *ntr-reverse-split-atU*[*OF - ATUV*(*2*) *REVSPLIT*(*7*)] *ntrp-valid-preserve-s*[*OF SPLIT*(*1*), *simplified*] *REVSPLIT*(*1*) **obtain** *sq″ w22′ ce22e′* **where**

$*$: $sq'' \in\# \ ceh\ w22' \preceq w22\ ce22e' \subseteq\# \ ce22'\ atU\ V\ ce22e'\ (\{\#sq''\#\},w22',ce22e') \in trcl$ $(ntr\ fg)$ **by** $(auto\ simp\ add: valid\text{-}unconc)$

from $CHFMT[of\ sq'']$ $REVSPLIT(1)$ $this(1)$ **obtain** q' **where** $sq''=[entry\ fg\ q']$ **by** $auto$

with $*$ **show** $thesis$ **using** $that$ **by** $blast$

qed

from $ntrs.gtr2gtrp$[**where** $c=\{\#\}$, $simplified$, $OF\ REVSPLIT'(5)$] **obtain** $sq''\ ce22ee'\ ww22'$ **where** $R\text{-}CONV'$: $ce22e' = add\text{-}mset\ sq''\ ce22ee'\ w22'=map\ le\text{-}rem\text{-}s\ ww22'\ (([entry\ fg\ q'],\{\#\}),ww22',(sq'',ce22ee')) \in trcl\ (ntrp\ fg)$ **by** $blast$

— From the soundness of the RU-constraint system, we get the corresponding entries

from $RU\text{-}sound[OF\ R\text{-}CONV(3)\ ATUV(1)]$ **obtain** $Ml\ Me\ h$ **where** RU: $(entry\ fg\ q,\ Ml,\ Me,\ h) \in RU\text{-}cs\ fg\ U\ Ml \subseteq mon\text{-}loc\ fg\ ww21\ Me \subseteq mon\text{-}env\ fg\ ww21\ h \leq \alpha ah\ (map\ (\alpha nl\ fg)\ ww21)$ **by** $blast$

from $RU\text{-}sound[OF\ R\text{-}CONV'(3),\ of\ V]$ $REVSPLIT'(4)$ $R\text{-}CONV'(1)$ **obtain** $Ml'\ Me'\ h'$ **where** RV: $(entry\ fg\ q',\ Ml',\ Me',\ h') \in RU\text{-}cs\ fg\ V\ Ml' \subseteq mon\text{-}loc\ fg\ ww22'\ Me' \subseteq mon\text{-}env\ fg\ ww22'\ h' \leq \alpha ah\ (map\ (\alpha nl\ fg)\ ww22')$ **by** $auto$

from $S\text{-}ENTRY\text{-}PAT[of\ \{\#q\#\}+\{\#q'\#\},\ simplified]$ $REVSPLIT(1)$ $REVSPLIT'(1)$ **have** $S\text{-}ENTRY$: $(v,\ mon\text{-}w\ fg\ w,\ \{\#q\#\} + \{\#q'\#\}) \in S\text{-}cs\ fg\ (2::nat)$ **by** $(simp\ add: numerals)$

have $(u,\ mon\ fg\ p \cup mon\text{-}w\ fg\ w,\ Ml \cup Me \cup Ml' \cup Me') \in RUV\text{-}cs\ fg\ U\ V$ **proof** $(rule\ RUV\text{-}split\text{-}ee[OF\ FS\text{-}FMT(2,4)\ S\text{-}ENTRY - RU(1)\ RV(1)])$

from $MON\text{-}PU\ MON2\text{-}LEQ\ MON2N\text{-}LEQ\ R\text{-}CONV(2)\ R\text{-}CONV'(2)$ $mon\text{-}ww\text{-}ileq[OF\ REVSPLIT'(2),\ of\ fg]$ $RU(2,3)$ $RV(2,3)$ **show** $(mon\text{-}n\ fg\ u \cup mon\ fg\ p) \cap (Ml \cup Me \cup Ml' \cup Me') = \{\}$ **by** $(simp\ add: mon\text{-}ww\text{-}of\text{-}le\text{-}rem)$ $blast$

next

from $ah\text{-}interleavable1[OF\ REVSPLIT(3)]$ **have** $\alpha ah\ (map\ (\alpha n\ fg)\ w21)\ [*]\ \alpha ah\ (map\ (\alpha n\ fg)\ w22)$ **.**

thus $h\ [*]\ h'$

proof $(erule\text{-}tac\ ah\text{-}leq\text{-}il)$

note $RU(4)$

also have $map\ (\alpha nl\ fg)\ ww21 \preceq map\ (\alpha n\ fg)\ w21$ **using** $R\text{-}CONV(2)$ **by** $(simp\ add: \alpha n\text{-}\alpha nl)$

hence $\alpha ah\ (map\ (\alpha nl\ fg)\ ww21) \leq \alpha ah\ (map\ (\alpha n\ fg)\ w21)$ **by** $(rule\ \alpha ah\text{-}ileq)$

finally show $h \leq \alpha ah\ (map\ (\alpha n\ fg)\ w21)$ **.**

next

note $RV(4)$

also have $map\ (\alpha nl\ fg)\ ww22' \preceq map\ (\alpha n\ fg)\ w22$ **using** $R\text{-}CONV'(2)$ $REVSPLIT'(2)$ **by** $(simp\ add: \alpha n\text{-}\alpha nl[symmetric]\ le\text{-}list\text{-}map\ map\text{-}map[symmetric]\ del: map\text{-}map)$

hence $\alpha ah\ (map\ (\alpha nl\ fg)\ ww22') \leq \alpha ah\ (map\ (\alpha n\ fg)\ w22)$ **by** $(rule\ \alpha ah\text{-}ileq)$

finally show $h' \leq \alpha ah\ (map\ (\alpha n\ fg)\ w22)$ **.**

qed

qed $(simp)$

moreover have $mon\ fg\ p \cup mon\text{-}w\ fg\ w \subseteq mon\text{-}loc\ fg\ (ee\#ww)$ **using** $FS\text{-}FMT(1)$ **by** $auto$

moreover have $Ml \cup Me \cup Ml' \cup Me' \subseteq mon\text{-}env\ fg\ (ee\#ww)$ **using** $RV(2,3)$

RU(*2*,*3*) *mon-ww-ileq*[*OF REVSPLIT′*(*2*), *of fg*] *MON2N-LEQ R-CONV*(*2*) *R-CONV′*(*2*) *MON2-LEQ* **by** (*simp add*: *mon-ww-of-le-rem*) *blast*
ultimately show *?thesis* **by** *blast*
qed
next
case *lr* — The first node is reached from the local thread, the second one from a spawned thread
from *RU-sound*[*OF DECOMP-LOC*(*2*), *of U*] *lr*(*1*) *DECOMP-LOC*(*1*) **obtain** *Ml Me h* **where** *RU*: (*v*, *Ml*, *Me*, *h*) ∈ *RU-cs fg U Ml* ⊆ *mon-loc fg w1 Me* ⊆ *mon-env fg w1 h* ≤ *αah* (*map* (*αnl fg*) *w1*) **by** *auto*
obtain *Ml′ Me′ h′ q′* **where** *RV*: [*entry fg q′*] ∈# *ch* (*entry fg q′*, *Ml′*, *Me′*, *h′*) ∈ *RU-cs fg V Ml′* ⊆ *mon-ww fg w2 Me′* ⊆ *mon-ww fg w2 h′* ≤ *αah* (*map* (*αn fg*) *w2*)
proof −
— We have to extract the interesting thread from the spawned threads in order to get an entry in *RU fg V*
obtain *q′ w2′ c2i′* **where** *REVSPLIT*: [*entry fg q′*] ∈# *ch w2′*⪯*w2 c2i′* ⊆# *c2′ atU V c2i′* ({#[*entry fg q′*]#},*w2′*,*c2i′*)∈*trcl* (*ntr fg*)
using *ntr-reverse-split-atU*[*OF* - *lr*(*2*) *LESPLIT*(*4*)] *ntrp-valid-preserve-s*[*OF SPLIT*(*1*), *simplified*] *CHFMT* **by** (*simp add*: *valid-unconc*) *blast*
from *ntrs.gtr2gtrp*[**where** *c*={#}, *simplified*, *OF REVSPLIT*(*5*)] **obtain** *s2i′ c2ie′ ww2′* **where** *R-CONV*: *c2i′*=*add-mset s2i′ c2ie′ w2′*=*map le-rem-s ww2′* (([*entry fg q′*], {#}), *ww2′*, *s2i′*, *c2ie′*) ∈ *trcl* (*ntrp fg*) **.**
from *RU-sound*[*OF R-CONV*(*3*), *of V*] *REVSPLIT*(*4*) *R-CONV*(*1*) **obtain** *Ml′ Me′ h′* **where** *RV*: (*entry fg q′*, *Ml′*, *Me′*, *h′*) ∈ *RU-cs fg V Ml′* ⊆ *mon-loc fg ww2′ Me′* ⊆ *mon-env fg ww2′ h′* ≤ *αah* (*map* (*αnl fg*) *ww2′*) **by** *auto*
moreover have *mon-loc fg ww2′* ⊆ *mon-ww fg w2 mon-env fg ww2′* ⊆ *mon-ww fg w2* **using** *mon-ww-ileq*[*OF REVSPLIT*(*2*), *of fg*] *R-CONV*(*2*) **by** (*auto simp add*: *mon-ww-of-le-rem*)
moreover have *αah* (*map* (*αnl fg*) *ww2′*) ≤ *αah* (*map* (*αn fg*) *w2*) **using** *REVSPLIT*(*2*) *R-CONV*(*2*) **by** (*auto simp add*: *αn-αnl*[*symmetric*] *le-list-map map-map*[*symmetric*] *simp del*: *map-map intro*: *αah-ileq del*: *predicate2I*)
ultimately show *thesis* **using** *that REVSPLIT*(*1*) **by** (*blast intro*: *order-trans*)
qed
from *S-ENTRY-PAT*[*of* {#*q′*#}, *simplified*] *RV*(*1*) **have** *S-ENTRY*: (*v*, *mon-w fg w*, {#*q′*#}) ∈ *S-cs fg 1* **by** *simp*
have (*u*, *mon fg p* ∪ *mon-w fg w* ∪ *Ml*, *Me* ∪ *Ml′* ∪ *Me′*) ∈ *RUV-cs fg U V*
proof (*rule RUV-split-le*[*OF FS-FMT*(*2*,*4*) *S-ENTRY* - *RU*(*1*) *RV*(*2*)])
from *MON-PU MON1-LEQ MON2-LEQ RU*(*3*) *RV*(*3*,*4*) **show** (*mon-n fg u* ∪ *mon fg p*) ∩ (*Me* ∪ *Ml′* ∪ *Me′*) = {} **by** *blast*
next
from *ah-interleavable1*[*OF LESPLIT*(*1*)] **have** *αah* (*map* (*αnl fg*) *w1*) [∗] *αah* (*map* (*αn fg*) *w2*) **by** *simp*
thus *h* [∗] *h′* **using** *RU*(*4*) *RV*(*5*) **by** (*auto elim*: *ah-leq-il*)
qed (*simp*)
moreover have *mon fg p* ∪ *mon-w fg w* ∪ *Ml* ⊆ *mon-loc fg* (*ee* # *ww*) **using** *FS-FMT*(*1*) *MON1-LEQ RU*(*2*) **by** (*simp*) *blast*
moreover have *Me* ∪ *Ml′* ∪ *Me′* ⊆ *mon-env fg* (*ee* # *ww*) **using** *MON1-LEQ MON2-LEQ RU*(*3*) *RV*(*3*,*4*) **by** (*simp*) *blast*

ultimately show *?thesis* **by** *blast*
next
case *rl* — The second node is reached from the local thread, the first one from a spawned thread. This case is symmetric to the previous one
from *RU-sound*[*OF DECOMP-LOC*(*2*), *of V*] *rl*(*1*) *DECOMP-LOC*(*1*) **obtain** *Ml Me h* **where** *RV*: (*v*, *Ml*, *Me*, *h*) ∈ *RU-cs fg V Ml* ⊆ *mon-loc fg w1 Me* ⊆ *mon-env fg w1 h* ≤ *αah* (*map* (*αnl fg*) *w1*) **by** *auto*
obtain *Ml′ Me′ h′ q′* **where** *RU*: [*entry fg q′*] ∈# *ch* (*entry fg q′*, *Ml′*, *Me′*, *h′*) ∈ *RU-cs fg U Ml′* ⊆ *mon-ww fg w2 Me′* ⊆ *mon-ww fg w2 h′* ≤ *αah* (*map* (*αn fg*) *w2*)
proof –
— We have to extract the interesting thread from the spawned threads in order to get an entry in *RU fg V*
obtain *q′ w2′ c2i′* **where** *REVSPLIT*: [*entry fg q′*] ∈# *ch w2′*⪯*w2 c2i′* ⊆# *c2′ atU U c2i′* ({#[*entry fg q′*]#},*w2′*,*c2i′*)∈*trcl* (*ntr fg*)
using *ntr-reverse-split-atU*[*OF - rl*(*2*) *LESPLIT*(*4*)] *ntrp-valid-preserve-s*[*OF SPLIT*(*1*), *simplified*] *CHFMT* **by** (*simp add*: *valid-unconc*) *blast*
from *ntrs.gtr2gtrp*[**where** *c*={#}, *simplified*, *OF REVSPLIT*(*5*)] **obtain** *s2i′ c2ie′ ww2′* **where** *R-CONV*: *c2i′*=*add-mset s2i′ c2ie′ w2′*=*map le-rem-s ww2′* (([*entry fg q′*], {#}), *ww2′*, *s2i′*, *c2ie′*) ∈ *trcl* (*ntrp fg*) **.**
from *RU-sound*[*OF R-CONV*(*3*), *of U*] *REVSPLIT*(*4*) *R-CONV*(*1*) **obtain** *Ml′ Me′ h′* **where** *RU*: (*entry fg q′*, *Ml′*, *Me′*, *h′*) ∈ *RU-cs fg U Ml′* ⊆ *mon-loc fg ww2′ Me′* ⊆ *mon-env fg ww2′ h′* ≤ *αah* (*map* (*αnl fg*) *ww2′*) **by** *auto*
moreover have *mon-loc fg ww2′* ⊆ *mon-ww fg w2 mon-env fg ww2′* ⊆ *mon-ww fg w2* **using** *mon-ww-ileq*[*OF REVSPLIT*(*2*), *of fg*] *R-CONV*(*2*) **by** (*auto simp add*: *mon-ww-of-le-rem*)
moreover have *αah* (*map* (*αnl fg*) *ww2′*) ≤ *αah* (*map* (*αn fg*) *w2*) **using** *REVSPLIT*(*2*) *R-CONV*(*2*) **by** (*auto simp add*: *αn-αnl*[*symmetric*] *le-list-map map-map*[*symmetric*] *simp del*: *map-map intro*: *αah-ileq del*: *predicate2I*)
ultimately show *thesis* **using** *that REVSPLIT*(*1*) **by** (*blast intro*: *order-trans*)
qed
from *S-ENTRY-PAT*[*of* {#*q′*#}, *simplified*] *RU*(*1*) **have** *S-ENTRY*: (*v*, *mon-w fg w*, {#*q′*#}) ∈ *S-cs fg 1* **by** *simp*
have (*u*, *mon fg p* ∪ *mon-w fg w* ∪ *Ml*, *Me* ∪ *Ml′* ∪ *Me′*) ∈ *RUV-cs fg U V* **proof** (*rule RUV-split-el*[*OF FS-FMT*(*2,4*) *S-ENTRY - RV*(*1*) *RU*(*2*)])
from *MON-PU MON1-LEQ MON2-LEQ RV*(*3*) *RU*(*3,4*) **show** (*mon-n fg u* ∪ *mon fg p*) ∩ (*Me* ∪ *Ml′* ∪ *Me′*) = {} **by** *blast*
next
from *ah-interleavable1*[*OF LESPLIT*(*1*)] **have** *αah* (*map* (*αnl fg*) *w1*) [∗] *αah* (*map* (*αn fg*) *w2*) **by** *simp*
thus *h* [∗] *h′* **using** *RV*(*4*) *RU*(*5*) **by** (*auto elim*: *ah-leq-il*)
qed (*simp*)
moreover have *mon fg p* ∪ *mon-w fg w* ∪ *Ml* ⊆ *mon-loc fg* (*ee* # *ww*) **using** *FS-FMT*(*1*) *MON1-LEQ RV*(*2*) **by** (*simp*) *blast*
moreover have *Me* ∪ *Ml′* ∪ *Me′* ⊆ *mon-env fg* (*ee* # *ww*) **using** *MON1-LEQ MON2-LEQ RV*(*3*) *RU*(*3,4*) **by** (*simp*) *blast*
ultimately show *?thesis* **by** *blast*
qed
qed

lemma (**in** *flowgraph*) *RUV-precise*: $(u,Ml,Me)\in RUV\text{-}cs\ fg\ U\ V$
$\Longrightarrow \exists\, w\ s'\ c'.$
$(([u],\{\#\}),w,(s',c'))\in trcl\ (ntrp\ fg) \wedge$
$atUV\ U\ V\ (\{\#s'\#\}+c') \wedge$
$mon\text{-}loc\ fg\ w = Ml \wedge$
$mon\text{-}env\ fg\ w = Me$
proof (*induct rule*: *RUV-cs.induct*)
case (*RUV-call u p u' v M P Ml Me*) **then obtain** *ww s' c'* **where** *IH*: $(([v], \{\#\}), ww, s', c') \in trcl\ (ntrp\ fg)$ $atUV\ U\ V\ (\{\#s'\#\} + c')$ $mon\text{-}loc\ fg\ ww = Ml$ $mon\text{-}env\ fg\ ww = Me$ **by** *blast*
from *S-precise-ntrp*[*OF RUV-call*(*3*,*2*,*1*), *simplified*] **obtain** *w ch* **where** *FS*: $(([u], \{\#\}), LOC\ (LCall\ p\ \#\ w), [v, u'], ch) \in ntrp\ fg$ $P = \{\#\}$ $M = mon\text{-}w\ fg\ w$ $mon\text{-}n\ fg\ v = mon\ fg\ p$ $mon\text{-}c\ fg\ ch = \{\}$ **by** *blast*
note *FS*(*1*)
also have $(([v, u'], ch), ww, s' @ [u'], c' + ch) \in trcl\ (ntrp\ fg)$
using *ntrp-add-context*[*OF ntrp-stack-comp*[*OF IH*(*1*), *of* [*u'*]], *of ch*, *simplified*] *FS*(*5*) *IH*(*4*) *RUV-call.hyps*(*6*) *mon-n-same-proc*[*OF edges-part*[*OF RUV-call.hyps*(*1*)]] **by** *simp*
finally have $(([u], \{\#\}), LOC\ (LCall\ p\ \#\ w)\ \#\ ww, s' @ [u'], c' + ch) \in trcl\ (ntrp\ fg)$ **.**
moreover from *IH*(*2*) **have** $atUV\ U\ V\ (\{\#s' @ [u']\#\}+(c'+ch))$ **by** *auto*
moreover have $mon\text{-}loc\ fg\ (LOC\ (LCall\ p\ \#\ w)\ \#\ ww) = mon\ fg\ p \cup M \cup Ml$ **using** *IH*(*3*) *FS*(*3*) **by** *auto*
moreover have $mon\text{-}env\ fg\ (LOC\ (LCall\ p\ \#\ w)\ \#\ ww) = Me$ **using** *IH*(*4*) **by** *auto*
ultimately show *?case* **by** *blast*
next
case (*RUV-spawn u p u' v M P q Ml Me*) **then obtain** *ww s' c'* **where** *IH*: $(([entry\ fg\ q], \{\#\}), ww, s', c') \in trcl\ (ntrp\ fg)$ $atUV\ U\ V\ (\{\#s'\#\} + c')$ $mon\text{-}loc\ fg\ ww = Ml$ $mon\text{-}env\ fg\ ww = Me$ **by** *blast*
from *S-precise-ntrp*[*OF RUV-spawn*(*3*,*2*,*1*), *simplified*] *mset-size1elem*[*OF - RUV-spawn*(*4*)] **obtain** *w che* **where**
FS: $(([u], \{\#\}), LOC\ (LCall\ p\ \#\ w), [v, u'], \{\#[entry\ fg\ q]\#\} + che) \in ntrp\ fg$ $P=\{\#q\#\}$ $M = mon\text{-}w\ fg\ w$ $mon\text{-}n\ fg\ v = mon\ fg\ p$ $mon\text{-}c\ fg\ (\{\#[entry\ fg\ q]\#\}+che) = \{\}$ **by** (*auto elim*: *mset-le-addE*)
moreover
have $(([v, u'], che + \{\#[entry\ fg\ q]\#\}), map\ ENV\ (map\ le\text{-}rem\text{-}s\ ww), ([v,u'],che+(\{\#s'\#\} + c')))\in trcl\ (ntrp\ fg)$
using *ntr2ntrp*[*OF gtrp2gtr*[*OF IH*(*1*)], *of* [*v*,*u'*] *che*] *IH*(*3*,*4*) *RUV-spawn*(*7*) *FS*(*4*,*5*) *mon-n-same-proc*[*OF edges-part*[*OF RUV-spawn*(*1*)]]
by (*auto simp add*: *mon-c-unconc mon-ww-of-le-rem*)
ultimately have $(([u], \{\#\}), LOC\ (LCall\ p\ \#\ w)\ \#\ map\ ENV\ (map\ le\text{-}rem\text{-}s\ ww), ([v,u'],che+(\{\#s'\#\} + c'))) \in trcl\ (ntrp\ fg)$ **by** (*auto simp add*: *union-ac*)
moreover have $atUV\ U\ V\ (\{\#[v,u']\#\} + (che+(\{\#s'\#\} + c')))$ **using** *IH*(*2*) **by** *auto*
moreover have $mon\text{-}loc\ fg\ (LOC\ (LCall\ p\ \#\ w)\ \#\ map\ ENV\ (map\ le\text{-}rem\text{-}s\ ww)) = mon\ fg\ p \cup M$ **using** *FS*(*3*) **by** (*simp del*: *map-map*)
moreover have $mon\text{-}env\ fg\ (LOC\ (LCall\ p\ \#\ w)\ \#\ map\ ENV\ (map\ le\text{-}rem\text{-}s$

ww)) = *Ml* ∪ *Me* **using** *IH*(*3*,*4*) **by** (*auto simp add*: *mon-ww-of-le-rem simp del*: *map-map*)
ultimately show *?case* **by** *blast*
next
case (*RUV-split-le u p u′ v M P q Ml Me h Ml′ Me′ h′*)
— Get paths from precision results
from *S-precise-ntrp*[*OF RUV-split-le*(*3*,*2*,*1*), *simplified*] *mset-size1elem*[*OF - RUV-split-le*(*4*)]
obtain *w che* **where**
FS: ((\[*u*\], {#}), *LOC* (*LCall p # w*), \[*v*, *u′*\], {#\[*entry fg q*\]#} + *che*) ∈ *ntrp fg P*={#*q*#} *M* = *mon-w fg w mon-n fg v* = *mon fg p mon-c fg* ({#\[*entry fg q*\]#}+*che*) = {} **by** (*auto elim*: *mset-le-addE*)
from *RU-precise*[*OF RUV-split-le*(*5*)] **obtain** *ww1 s1′ c1′* **where** *P1*: ((\[*v*\], {#}), *ww1*, *s1′*, *c1′*) ∈ *trcl* (*ntrp fg*) *atU U* ({#*s1′*#} + *c1′*) *mon-loc fg ww1* = *Ml mon-env fg ww1* = *Me αah* (*map* (*αnl fg*) *ww1*) = *h* **by** *blast*
from *RU-precise*[*OF RUV-split-le*(*6*)] **obtain** *ww2 s2′ c2′* **where** *P2*: ((\[*entry fg q*\], {#}), *ww2*, *s2′*, *c2′*) ∈ *trcl* (*ntrp fg*) *atU V* ({#*s2′*#} + *c2′*) *mon-loc fg ww2* = *Ml′ mon-env fg ww2* = *Me′ αah* (*map* (*αnl fg*) *ww2*) = *h′* **by** *blast*
— Get combined path from the acquisition history interleavability, need to remap loc/env-steps in second path
from *P2*(*5*) **have** *αah* (*map* (*αnl fg*) (*map ENV* (*map le-rem-s ww2*))) = *h′* **by** (*simp add*: *αn-αnl o-assoc*)
with *P1*(*5*) *RUV-split-le*(*8*) **obtain** *ww* **where** *IL*: $ww \in ww1 \otimes_{\alpha nl\ fg}$(*map ENV* (*map le-rem-s ww2*)) **using** *ah-interleavable2* **by** (*force*)
— Use the *ntrp-unsplit*-theorem to combine the executions
from *ntrp-unsplit*[**where** *ca*={#},*OF IL P1*(*1*) *gtrp2gtr*[*OF P2*(*1*)], *simplified*]
have ((\[*v*\], {#\[*entry fg q*\]#}), *ww*, *s1′*, *c1′* + ({#*s2′*#} + *c2′*)) ∈ *trcl* (*ntrp fg*)
using *FS*(*4*,*5*) *RUV-split-le*(*7*)
by (*auto simp add*: *mon-c-unconc mon-ww-of-le-rem P2*(*3*,*4*))
from *ntrp-add-context*[*OF ntrp-stack-comp*[*OF this*, *of* \[*u′*\]], *of che*] **have** ((\[*v*\] @ \[*u′*\], {#\[*entry fg q*\]#} + *che*), *ww*, *s1′* @ \[*u′*\], *c1′* + ({#*s2′*#} + *c2′*) + *che*) ∈ *trcl* (*ntrp fg*)
using *mon-n-same-proc*[*OF edges-part*[*OF RUV-split-le*(*1*)]] *mon-loc-cil*[*OF IL*, *of fg*] *mon-env-cil*[*OF IL*, *of fg*] *FS*(*4*,*5*) *RUV-split-le*(*7*) **by** (*auto simp add*: *mon-c-unconc P1*(*3*,*4*) *P2*(*3*,*4*) *mon-ww-of-le-rem simp del*: *map-map*)
with *FS*(*1*) **have** ((\[*u*\], {#}), *LOC* (*LCall p # w*) # *ww*, (*s1′* @ \[*u′*\], *c1′* + ({#*s2′*#} + *c2′*) + *che*))∈*trcl* (*ntrp fg*) **by** *simp*
moreover have *atUV U V* ({#*s1′* @ \[*u′*\]#}+(*c1′* + ({#*s2′*#} + *c2′*) + *che*)) **using** *P1*(*2*) *P2*(*2*) **by** *auto*
moreover have *mon-loc fg* (*LOC* (*LCall p # w*) # *ww*) = *mon fg p* ∪ *M* ∪ *Ml* **using** *FS*(*3*) *P1*(*3*) *mon-loc-cil*[*OF IL*, *of fg*] **by** (*auto simp del*: *map-map*)
moreover have *mon-env fg* (*LOC* (*LCall p # w*) # *ww*) = *Me* ∪ *Ml′* ∪ *Me′* **using** *P1*(*4*) *P2*(*3*,*4*) *mon-env-cil*[*OF IL*, *of fg*] **by** (*auto simp add*: *mon-ww-of-le-rem simp del*: *map-map*)
ultimately show *?case* **by** *blast*
next
case (*RUV-split-el u p u′ v M P q Ml Me h Ml′ Me′ h′*) — This is the symmetric case to *RUV-split-le*, it is proved completely analogously, just need to swap *U* and *V*.
— Get paths from precision results

from *S-precise-ntrp*[*OF RUV-split-el*(*3*,*2*,*1*), *simplified*] *mset-size1elem*[*OF - RUV-split-el*(*4*)]
obtain *w che* **where**
FS: (([*u*], {#}), *LOC* (*LCall p* # *w*), [*v*, *u'*], {#[*entry fg q*]#} + *che*) ∈ *ntrp fg* *P*={#*q*#} *M* = *mon-w fg w* *mon-n fg v* = *mon fg p* *mon-c fg* ({#[*entry fg q*]#}+*che*) = {} **by** (*auto elim*: *mset-le-addE*)
from *RU-precise*[*OF RUV-split-el*(*5*)] **obtain** *ww1 s1' c1'* **where** *P1*: (([*v*], {#}), *ww1*, *s1'*, *c1'*) ∈ *trcl* (*ntrp fg*) *atU V* ({#*s1'*#} + *c1'*) *mon-loc fg ww1* = *Ml* *mon-env fg ww1* = *Me* *αah* (*map* (*αnl fg*) *ww1*) = *h* **by** *blast*
from *RU-precise*[*OF RUV-split-el*(*6*)] **obtain** *ww2 s2' c2'* **where** *P2*: (([*entry fg q*], {#}), *ww2*, *s2'*, *c2'*) ∈ *trcl* (*ntrp fg*) *atU U* ({#*s2'*#} + *c2'*) *mon-loc fg ww2* = *Ml'* *mon-env fg ww2* = *Me'* *αah* (*map* (*αnl fg*) *ww2*) = *h'* **by** *blast*
— Get combined path from the acquisition history interleavability, need to remap loc/env-steps in second path
from *P2*(*5*) **have** *αah* (*map* (*αnl fg*) (*map ENV* (*map le-rem-s ww2*))) = *h'* **by** (*simp add*: *αn-αnl o-assoc*)
with *P1*(*5*) *RUV-split-el*(*8*) **obtain** *ww* **where** *IL*: $ww \in ww1 \otimes_{\alpha nl\ fg}$(*map ENV* (*map le-rem-s ww2*)) **using** *ah-interleavable2* **by** (*force*)
— Use the *ntrp-unsplit*-theorem to combine the executions
from *ntrp-unsplit*[**where** *ca*={#},*OF IL P1*(*1*) *gtrp2gtr*[*OF P2*(*1*)], *simplified*] **have** (([*v*], {#[*entry fg q*]#}), *ww*, *s1'*, *c1'* + ({#*s2'*#} + *c2'*)) ∈ *trcl* (*ntrp fg*)
using *FS*(*4*,*5*) *RUV-split-el*(*7*)
by (*auto simp add*: *mon-c-unconc mon-ww-of-le-rem P2*(*3*,*4*))
from *ntrp-add-context*[*OF ntrp-stack-comp*[*OF this*, *of* [*u'*]], *of che*] **have** (([*v*] @ [*u'*], {#[*entry fg q*]#} + *che*), *ww*, *s1'* @ [*u'*], *c1'* + ({#*s2'*#} + *c2'*) + *che*) ∈ *trcl* (*ntrp fg*)
using *mon-n-same-proc*[*OF edges-part*[*OF RUV-split-el*(*1*)]] *mon-loc-cil*[*OF IL*, *of fg*] *mon-env-cil*[*OF IL*, *of fg*] *FS*(*4*,*5*) *RUV-split-el*(*7*) **by** (*auto simp add*: *mon-c-unconc P1*(*3*,*4*) *P2*(*3*,*4*) *mon-ww-of-le-rem simp del*: *map-map*)
with *FS*(*1*) **have** (([*u*], {#}), *LOC* (*LCall p* # *w*) # *ww*, (*s1'* @ [*u'*], *c1'* + ({#*s2'*#} + *c2'*) + *che*))∈*trcl* (*ntrp fg*) **by** *simp*
moreover have *atUV U V* ({#*s1'* @ [*u'*]#}+(*c1'* + ({#*s2'*#} + *c2'*) + *che*)) **using** *P1*(*2*) *P2*(*2*) **by** *auto*
moreover have *mon-loc fg* (*LOC* (*LCall p* # *w*) # *ww*) = *mon fg p* ∪ *M* ∪ *Ml* **using** *FS*(*3*) *P1*(*3*) *mon-loc-cil*[*OF IL*, *of fg*] **by** (*auto simp del*: *map-map*)
moreover have *mon-env fg* (*LOC* (*LCall p* # *w*) # *ww*) = *Me* ∪ *Ml'* ∪ *Me'* **using** *P1*(*4*) *P2*(*3*,*4*) *mon-env-cil*[*OF IL*, *of fg*] **by** (*auto simp add*: *mon-ww-of-le-rem simp del*: *map-map*)
ultimately show *?case* **by** *blast*
next
case (*RUV-split-ee u p u' v M P q q' Ml Me h Ml' Me' h'*)
— Get paths from precision results
from *S-precise-ntrp*[*OF RUV-split-ee*(*3*,*2*,*1*), *simplified*] *mset-size2elem*[*OF - RUV-split-ee*(*4*)] **obtain** *w che* **where**
FS: (([*u*], {#}), *LOC* (*LCall p* # *w*), [*v*, *u'*], {#[*entry fg q*]#} + {#[*entry fg q'*]#} + *che*) ∈ *ntrp fg* *P*={#*q*#}+{#*q'*#} *M* = *mon-w fg w* *mon-n fg v* = *mon fg p* *mon-c fg* ({#[*entry fg q*]#}+{#[*entry fg q'*]#}+*che*) = {}
by (*auto elim*: *mset-le-addE*)
from *RU-precise*[*OF RUV-split-ee*(*5*)] **obtain** *ww1 s1' c1'* **where** *P1*: (([*entry fg q*], {#}), *ww1*, *s1'*, *c1'*) ∈ *trcl* (*ntrp fg*) *atU U* ({#*s1'*#} + *c1'*) *mon-loc fg ww1* =

Ml mon-env fg ww1 = *Me αah* (*map* (*αnl fg*) *ww1*) = *h* **by** *blast*
from *RU-precise*[*OF RUV-split-ee*(*6*)] **obtain** *ww2 s2′ c2′* **where** *P2*: (([*entry fg q′*], {#}), *ww2*, *s2′*, *c2′*) ∈ *trcl* (*ntrp fg*) *atU V* ({#*s2′*#} + *c2′*) *mon-loc fg ww2* = *Ml′ mon-env fg ww2* = *Me′ αah* (*map* (*αnl fg*) *ww2*) = *h′* **by** *blast*

— Get interleaved paths, project away loc/env information first
from *P1*(*5*) *P2*(*5*) **have** *αah* (*map* (*αn fg*) (*map le-rem-s ww1*)) = *h αah* (*map* (*αn fg*) (*map le-rem-s ww2*)) = *h′* **by** (*auto simp add*: *αn-αnl o-assoc*)
with *RUV-split-ee*(*8*) **obtain** *ww* **where** *IL*: *ww* ∈ (*map le-rem-s ww1*) $\otimes_{\alpha n\ fg}$ (*map le-rem-s ww2*) **using** *ah-interleavable2* **by** (*force simp del*: *map-map*)
— Use the *ntr-unsplit*-theorem to combine the executions
from *ntr-unsplit*[*OF IL gtrp2gtr*[*OF P1*(*1*)] *gtrp2gtr*[*OF P2*(*1*)], *simplified*] **have** *PC*: ({#[*entry fg q*]#} + {#[*entry fg q′*]#}, *ww*, {#*s1′*#} + *c1′* + ({#*s2′*#} + *c2′*)) ∈ *trcl* (*ntr fg*) **using** *FS*(*5*) **by** (*auto simp add*: *mon-c-unconc*)
— Prepend first step
from *ntr2ntrp*[*OF PC*(*1*), *of* [*v*,*u′*] *che*] **have** (([*v*, *u′*], *che* + ({#[*entry fg q*]#} + {#[*entry fg q′*]#})), *map ENV ww*, [*v*, *u′*], *che* + ({#*s1′*#} + *c1′* + ({#*s2′*#} + *c2′*))) ∈ *trcl* (*ntrp fg*)
using *RUV-split-ee*(*7*) *FS*(*5*) *mon-ww-cil*[*OF IL*, *of fg*] *FS*(*4*) *mon-n-same-proc*[*OF edges-part*[*OF RUV-split-ee*(*1*)]] **by** (*auto simp add*: *mon-c-unconc mon-ww-of-le-rem P1*(*3*,*4*) *P2*(*3*,*4*))
with *FS*(*1*) **have** (([*u*], {#}), *LOC* (*LCall p* # *w*) # *map ENV ww*, ([*v*, *u′*], *che* + ({#*s1′*#} + *c1′* + ({#*s2′*#} + *c2′*)))) ∈ *trcl* (*ntrp fg*) **by** (*auto simp add*: *union-ac*)
moreover have *atUV U V* ({#[*v*, *u′*]#}+(*che* + ({#*s1′*#} + *c1′* + ({#*s2′*#} + *c2′*)))) **using** *P1*(*2*) *P2*(*2*) **by** *auto*
moreover have *mon-loc fg* (*LOC* (*LCall p* # *w*) # *map ENV ww*) = *mon fg p* ∪ *M* **using** *FS*(*3*) **by** *auto*
moreover have *mon-env fg* (*LOC* (*LCall p* # *w*) # *map ENV ww*) = *Ml* ∪ *Me* ∪ *Ml′* ∪ *Me′* **using** *mon-ww-cil*[*OF IL*, *of fg*] **by** (*auto simp add*: *P1*(*3*,*4*) *P2*(*3*,*4*) *mon-ww-of-le-rem*)
ultimately show *?case* **by** *blast*
qed

end

10 Main Result

theory *MainResult*
imports *ConstraintSystems*
begin

At this point everything is available to prove the main result of this project: *The constraint system RUV-cs precisely characterizes simultaneously reachable control nodes w.r.t. to our semantic reference point.*

The „trusted base" of this proof, that are all definitions a reader that trusts the Isabelle prover must additionally trust, is the following:

- The flowgraph and the assumptions made on it in the *flowgraph*- and *eflowgraph*-locales. Note that we show in Section 6.4 that there is at least one non-trivial model of *eflowgraph*.

- The reference point semantics (*refpoint*) and the transitive closure operator (*trcl*).
- The definition of *atUV*.
- All dependencies of the above definitions in the Isabelle standard libraries.

theorem (**in** *eflowgraph*) *RUV-is-sim-reach*:
 $(\exists\, w\ c'.\ (\{\#[entry\ fg\ (main\ fg)]\#\},w,c')\in trcl\ (refpoint\ fg) \wedge atUV\ U\ V\ c')$
 $\longleftrightarrow (\exists\, Ml\ Me.\ (entry\ fg\ (main\ fg),Ml,Me)\in RUV\text{-}cs\ fg\ U\ V)$
— The proof uses the soundness and precision theorems wrt. to normalized paths (*flowgraph.RUV-sound*, *flowgraph.RUV-precise*) as well as the normalization result, i.e. that every reachable configuration is also reachable using a normalized path (*eflowgraph.normalize*) and, vice versa, that every normalized path is also a usual path (*ntr-is-tr*). Finally the conversion between our working semantics and the semantic reference point is exploited (*flowgraph.refpoint-eq*).
 (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
 assume *?lhs*
 then obtain *w c′* **where** *C*: $(\{\#[entry\ fg\ (main\ fg)]\#\},\ w,\ c') \in trcl\ (tr\ fg)$ $atUV\ U\ V\ c'$ **by** (*auto simp add*: *refpoint-eq*)
 from *normalize*[*OF C*(*1*), *of main fg*, *simplified*] **obtain** *ww* **where** $(\{\#[entry\ fg\ (main\ fg)]\#\},\ ww,\ c') \in trcl\ (ntr\ fg)$ **by** *blast*
 from *ntrs.gtr2gtrp*[**where** *c*={#}, *simplified*, *OF this*] **obtain** *s′ ce′ wwl* **where** *1*: $c'=add\text{-}mset\ s'\ ce'\ ww = map\ le\text{-}rem\text{-}s\ wwl\ (([entry\ fg\ (main\ fg)],\ \{\#\}),\ wwl,\ s',\ ce') \in trcl\ (ntrp\ fg)$ **by** *blast*
 with *C*(*2*) **have** *2*: $atUV\ U\ V\ (\{\#s'\#\}+ce')$ **by** *auto*
 from *RUV-sound*[*OF 1*(*3*) *2*] **show** *?rhs* **by** *blast*
next
 assume *?rhs*
 then obtain *Ml Me* **where** *C*: $(entry\ fg\ (main\ fg),\ Ml,\ Me) \in RUV\text{-}cs\ fg\ U\ V$ **by** *blast*
 from *RUV-precise*[*OF C*] **obtain** *wwl s′ c′* **where** *P*: $(([entry\ fg\ (main\ fg)],\ \{\#\}),\ wwl,\ s',\ c') \in trcl\ (ntrp\ fg)\ atUV\ U\ V\ (\{\#s'\#\} + c')$ **by** *blast*
 from *gtrp2gtr*[*OF P*(*1*)] **have** $(\{\#\ [entry\ fg\ (main\ fg)]\ \#\},\ map\ le\text{-}rem\text{-}s\ wwl,\ \{\#s'\#\}+c') \in trcl\ (ntr\ fg)$ **by** (*auto*)
 from *ntr-is-tr*[*OF this*] *P*(*2*) **have** $\exists\, w\ c'.\ (\{\#[entry\ fg\ (main\ fg)]\#\},\ w,\ c') \in trcl\ (tr\ fg) \wedge atUV\ U\ V\ c'$ **by** *blast*
 thus *?lhs* **by** (*simp add*: *refpoint-eq*)
qed

end

11 Conclusion

We have formalized a flowgraph-based model for programs with recursive procedure calls, dynamic thread creation and reentrant monitors and its operational semantics. Based on the operational semantics, we defined a conflict as being able to simultaneously reach two control points from two given sets U and V when starting at the initial program configuration, just consisting of a single thread at the entry point of the main procedure. We then formalized a constraint-system-based analysis for conflicts and proved it sound and precise w.r.t. the operational definition of a conflict. The main idea of the analysis was to restrict the possible schedules of a program. On the one hand, this restriction enabled the constraint system based analysis, on the other hand it did not change the set of reachable configurations (and thus the set of conflicts).

We characterized the constraint systems as inductive sets. While we did not derive an executable algorithm explicitly, the steps from the inductive sets characterization to an algorithm follow the path common in program analysis and pose no particular difficulty. The algorithm would have to construct a constraint system (system of inequalities over a finite height lattice) from a given program corresponding to the inductively defined sets studied here and then determine its least solution, e.g. by a worklist algorithm. In order to make the algorithm executable, we would have to introduce finiteness assumptions for our programs. The derivation of executable algorithms is currently in preparation.

A formal analysis of the algorithmic complexity of the problem will be presented elsewhere. Here we only present some results: Already the problem of deciding the reachability of a single control node is NP-hard, as can be shown by a simple reduction from SAT. On the other hand, we can decide simultaneous reachability in nondeterministic polynomial time in the program size, where the number of random bits depends on the possible nesting depth of the monitors. This can be shown by analyzing the constraint systems.

Acknowledgement We thank Dejvuth Suwimonteerabuth for an interesting discussion about static analysis of programs with locks. We also thank the people on the Isabelle mailing list for quick and useful responses.

References

[1] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. of CONCUR'05*. Springer, 2005.

[2] J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Proc. of FoSSaCS'99*, pages 14–30. Springer, 1999.

[3] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *Proc. of POPL'00*, pages 1–11. Springer, 2000.

[4] V. Kahlon and A. Gupta. An automata-theoretic approach for model checking threads for LTL properties. In *Proc. of LICS 2006*, pages 101–110. IEEE Computer Society, 2006.

[5] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *Proc. of CAV 2005*, pages 505–518. Springer, 2005.

[6] P. Lammich and M. Müller-Olm. Precise fixpoint-based analysis of programs with thread-creation. In *Proc. of CONCUR 2007*, pages 287–302. Springer, 2007.

[7] H. Seidl and B. Steffen. Constraint-based inter-procedural analysis of parallel programs. *Nordic Journal of Computing (NJC)*, 7(4):375–400, 2000.