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

June 11, 2019

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 4

2 Monitor Consistent Interleaving 5
  2.1 Monitors of lists of monitor pairs 5
  2.2 Properties of consistent interleaving 9

3 Acquisition Histories 11
  3.1 Definitions 12
  3.2 Interleavability 12
  3.3 Used monitors 12
  3.4 Ordering 13
  3.5 Acquisition histories of executions 13
  3.6 Acquisition history backward update 17

4 Labeled transition systems 18
  4.1 Definitions 18
  4.2 Basic properties of transitive reflexive closure 18
    4.2.1 Appending of elements to paths 20
    4.2.2 Transitivity reasoning setup 20
    4.2.3 Monotonicity 20
    4.2.4 Special lemmas for reasoning about states that are pairs 21
    4.2.5 Invariants 21

5 Thread Tracking 21
  5.1 Semantic on multiset configuration 21
  5.2 Invariants 22
  5.3 Context preservation assumption 23
  5.4 Explicit local context 24
    5.4.1 Lifted step datatype 25
    5.4.2 Definition of the loc/env-semantics 26
    5.4.3 Relation between multiset- and loc/env-semantics 26
    5.4.4 Invariants 28

6 Flowgraphs 28
  6.1 Definitions 29
  6.2 Basic properties 29
  6.3 Extra assumptions for flowgraphs 30
  6.4 Example Flowgraph 31

7 Operational Semantics 31
  7.1 Configurations and labels 31
  7.2 Monitors 32
  7.3 Valid configurations 34
7.4 Configurations at control points .................................. 35
7.5 Operational semantics .............................................. 37
7.5.1 Semantic reference point ....................................... 37
7.6 Basic properties ..................................................... 38
7.6.1 Validity ............................................................. 38
7.6.2 Equivalence to reference point .................................. 39
7.6.3 Case distinctions .................................................. 39
7.7 Advanced properties ............................................... 42
7.7.1 Stack composition / decomposition ............................. 42
7.7.2 Adding threads ................................................... 48
7.7.3 Conversion between environment and monitor restrictions .... 48

8 Normalized Paths .......................................................... 51
8.1 Semantic properties of restricted flowgraphs ...................... 51
8.2 Definition of normalized paths ..................................... 53
8.3 Representation property for reachable configurations ........... 54
8.4 Properties of normalized path ...................................... 60
8.4.1 Validity ............................................................. 61
8.4.2 Monitors ........................................................... 61
8.4.3 Modifying the context ............................................ 64
8.4.4 Altering the local stack .......................................... 68
8.5 Relation to monitor consistent interleaving ....................... 70
8.5.1 Abstraction function for normalized paths .................... 70
8.5.2 Monitors ........................................................... 72
8.5.3 Interleaving theorem ............................................. 72
8.5.4 Reverse splitting .................................................. 84

9 Constraint Systems .......................................................... 86
9.1 Same-level paths ...................................................... 87
9.1.1 Definition .......................................................... 87
9.1.2 Soundness and Precision ....................................... 88
9.2 Single reaching path .................................................. 93
9.2.1 Constraint system ................................................ 93
9.2.2 Soundness and precision ....................................... 94
9.3 Simultaneously reaching path ...................................... 102
9.3.1 Constraint system ................................................ 102
9.3.2 Soundness and precision ....................................... 103

10 Main Result ............................................................... 113

11 Conclusion ................................................................. 114
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 \( \pi \) and two sets of control points \( U \) and \( V \), the analysis problem is to decide whether there is an execution of \( \pi \) 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 \cite{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 \cite{5}.

Related Work In \cite{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 \cite{1, 2, 3} as well as constraint-system-based techniques \cite{7, 6} to analyze programs with procedures and either parallel calls or thread creation, but without any synchronization. In \cite{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\textsuperscript{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 un-
lock 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 formaliza-
tion of these ideas as a collection of Isabelle/HOL theories. A more abstract
description is in preparation. This document starts with formalization mon-
tor 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 con-
straint systems. Finally, the main result of this development – the correct-
ness 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 arbi-
trary elements, provided an abstraction function $\alpha$ 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: $\text{mon-pl (map } \alpha$ w)$ is the set of monitors used by a word $w$ w.r.t. the abstraction $\alpha$

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

lemma mon-pl-empty[simp]: mon-pl [] = {}
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 ⪯ w' ⇒ mon-pl w ⊆ mon-pl w'
by (induct rule: less-eq-list-induct) auto

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

fun
cil :: 'a list ⇒ ('m set × 'm set) ⇒ 'a list
| ⊗_α w = {w}
| w ⊗_α [] = {w}
— Interleaving with the empty word results in the empty word
— 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 ⊗_α e2#w2 = (if fst (α e1) ∩ mon-pl (map α (e2#w2)) = {} then
e1·(w1 ⊗_α e2#w2)
else {} ) ∪ (if fst (α e2) ∩ mon-pl (map α (e1#w1)) = {} then
e2·(e1#w1 ⊗_α 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 ⊗_α e2#w2;
| ![w', w=e1#w'; w'∈(w1 ⊗_α e2#w2);
|   fst (α e1) ∩ mon-pl (map α (e2#w2)) = {} ] => P;
| ![w', w=e2#w'; w'∈(e1#w1 ⊗_α w2);
|   fst (α e2) ∩ mon-pl (map α (e1#w1)) = {} ] => P
| ] => P
by (auto elim: list-set-cons-cases split: if-split-asmp)

lemma cil-cases[cases set, case-names both-empty left-empty right-empty app-left app-right]:
\[ \text{proof (induct } w a \alpha w b \text{ rule: cil.induct)} \]
\begin{align*}
\text{case 1 thus } & \text{ ?case by simp next } \\
\text{case 2 thus } & \text{ ?case by simp next } \\
\text{case (3 } e a & w a' \alpha e b w b' ) \\
\text{from } 3.\text{prems(1)} & \text{ show } \Theta \text{thesis proof (cases rule: cil-last-case-split)} \\
\text{case (left } w' \text{) from } 3.\text{prems(5)} & (\text{OF left(1) - left(2,3)} \text{) show } \Theta \text{thesis by simp next } \\
\text{case (right } w' \text{) from } 3.\text{prems(6)} & (\text{OF right(1) - right(2,3)} \text{) show } \Theta \text{thesis by simp next } \\
\text{qed} \\
\text{qed} \\
\end{align*}

\textbf{lemma cil-induct'[case-names both-empty left-empty right-empty append]}: \[ w \in w a \otimes a w b; \]
\[ w = [] ; w a = [] ; w b = [] \implies P; \]
\[ w a = [] ; w = w b \implies P; \]
\[ w = w a ; w b = [] \implies P; \]
\[ \text{!! } e a & w a' & w' \implies w = e a \# w' ; w a = e a \# w a' ; w' \in w a ' \otimes a w b; \]
\[ \text{fst } (\alpha e a) \cap \text{mon-pl } (\text{map } \alpha w b) = \{ \} \implies P; \]
\[ \text{!! } e b & w b' & w' \implies w = e b \# w' ; w b = e b \# w b' ; w' \in w a \otimes a w b; \]
\[ \text{fst } (\alpha e b) \cap \text{mon-pl } (\text{map } \alpha w a) = \{ \} \implies P \]
\[ \implies P \]
\text{apply (induct } w a \alpha w b \text{ rule: cil.induct) }
\text{apply (case-tac } w \text{) }
\text{apply auto }
\text{done}

\textbf{lemma cil-induct-fix}\alpha: \[ P \alpha [] [] ; \]
\[ \text{\L } a \text{ ad ae. } P \alpha [] (\text{ad } \# \text{ ae); } \]
\[ \text{\L } a z a a. P \alpha (z \# a a) [] ; \]
\[ \text{\L } e 1 w 1 e 2 w 2. \]
\[ \text{[fst } (\alpha e 1) \cap \text{mon-pl } (\text{map } \alpha (e 2 \# w 2))\} = \{ \} \implies P \alpha (e 1 \# e 2 \# w 2) ; \]
\[ \text{\implies P } \alpha (e 1 \# e 2 \# w 2) \]
\[ \implies P \alpha w a w b \]
\text{apply (induct } w a \alpha w b \text{ rule: cil.induct) }
\text{apply (case-tac } w \text{) }
\text{apply auto }
\text{done}

\text{apply (induct } v \alpha w \text{ rule: cil.induct) }
\text{apply (case-tac } w \text{) }
\text{apply auto }
\text{done}
 lemma cil-induct-fixα''[case-names both-empty left-empty right-empty append]: [
\[
\begin{align*}
\forall \alpha: P \alpha \emptyset = (\alpha \# \alpha)
\wedge \forall \alpha: P \alpha \emptyset = (\alpha \# \alpha)
\wedge w_1 w_2. \mathcal{F}(\alpha e) \cap \mathcal{M}(\map{\map{\alpha}{w_2}}{\alpha e}{w_2}) = \emptyset \implies P \alpha w_1 (e \# w_2)
\wedge w_2. \mathcal{F}(\alpha e) \cap \mathcal{M}(\map{\map{\alpha}{w_1}}{\alpha e}{w_1}) = \emptyset \implies P \alpha (e \# w_1) w_2
\end{align*}
\]
\] 
apply (induct wa α wb rule: cil.induct)
apply (case-tac w)
done

 lemma [simp]: \[ w \otimes \alpha \emptyset = \{ w \} \]
by (cases w, auto)

 lemma cil-contains-empty[rule-format, simp]: (\emptyset \in wa \otimes \alpha wb) = (wa=\emptyset \land wb=\emptyset)
by (induct wa α wb rule: cil.induct) auto

 lemma cil-cons-cases[cases set, case-names left right]: [ e \# w \in w_1 \otimes \alpha w_2; \]
\[ w_1 \# e \# w_2; \mathcal{F}(\alpha e) \cap \mathcal{M}(\map{\map{\alpha}{w_1}}{\alpha e}{w_1}) = \emptyset \implies P; \]
\[ w_2 \# e \# w_2; \mathcal{F}(\alpha e) \cap \mathcal{M}(\map{\map{\alpha}{w_1}}{\alpha e}{w_1}) = \emptyset \implies P \]
by (cases rule: cil-cases) auto

 lemma cil-set-induct[induct set, case-names empty left right]: [\![\alpha \in w_1 w_2, \]
\[ w \in w_1 \otimes \alpha w_2; \]
\[ \![\alpha e \in w_1 w_2; \mathcal{F}(\alpha e) \cap \mathcal{M}(\map{\map{\alpha}{w_1}}{\alpha e}{w_1}) = \emptyset \implies P; \]
\] \[ w_2 \in w_1 \otimes \alpha w_2; \]
\[ P \alpha \emptyset \emptyset \]
by (induct w) (auto intro!: cil-contains-empty elim: cil-cons-cases)

 lemma cil-set-induct-fixα[induct set, case-names empty left right]: [\![\alpha \in w_1 w_2, \]
\[ w \in w_1 \otimes \alpha w_2; \]
\[ P \emptyset \emptyset \]
by (induct w) (auto intro!: cil-contains-empty elim: cil-cons-cases)

 lemma cil-cons1: [\[ w \in wa \otimes \alpha wb; \mathcal{F}(\alpha e) \cap \mathcal{M}(\map{\map{\alpha}{wb}}{\alpha e}{wb}) = \emptyset \]
\[ \implies e \# w \in e \# wa \otimes \alpha wb \]
by (cases wb) auto
2.2 Properties of consistent interleaving

— Consistent interleaving preserves the length of both operands

**lemma** cil-subset-il': $\forall w1, w2. w \in w1 \otimes_{\alpha} w2 \implies w \subseteq w1 \otimes w2$

**apply** (induct rule: cil.induct)

**apply** simp-all

**apply** safe

**apply** auto

**done**

**lemma** cil-set-induct-fix: $\forall w, w1, w2. w \in w1 \otimes w2 \implies w1 \subseteq w1 \otimes w2$

**by** (induct rule: cil.induct) auto

**corollary** cil-mon-pl: $\forall w1, w2. w \in w1 \otimes_{\alpha} w2 \implies mon-pl (map \alpha w) \subseteq mon-pl (map \alpha w1) \cup mon-pl (map \alpha w2)$

**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: $\forall w. w \in w1 \otimes_{\alpha} w2 \implies \text{length } w = \text{length } w1 + \text{length } w2$

**by** (induct rule: cil.induct) auto

— Consistent interleaving contains all letters of each operand in the original order

**lemma** cil-ileq: $\forall w. w \in w1 \otimes_{\alpha} w2 \implies w1 \subseteq w \land w2 \subseteq w$

**by** (intro conj1 cil-subset-il' ileq-interleave)

— Consistent interleaving is commutative and associative

**lemma** cil-commute: $\forall w1, w2. w1 \otimes_{\alpha} w2 = w2 \otimes_{\alpha} w1$

**by** (induct rule: cil.induct) auto

**lemma** cil-assoc1: $\forall w1, w2, w3. \forall \alpha. w1 \otimes_{\alpha} w2 \otimes_{\alpha} w3$

**proof** (induct rule: length-compl-induct)

**case** Nil thus $\forall case$ **by** auto

**next**

**case** (Cons e w) from Cons.prems(1) show $\forall case$ **proof** (cases rule: cil-cons-cases)

**case** (left w') with Cons.prems(2) **have** $e \# w1 \otimes_{\alpha} w2$ **by** simp

**thus** $\forall thesis$ **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: $w \in w1 \otimes_{\alpha} w2 \otimes_{\alpha} w3$ **by** blast

**have** $e \# w1 \# w1' \otimes_{\alpha} wr$ **proof** (rule cil-cons1[OF IHAPP(1)])

**from** left left' cil-mon-pl[OF IHAPP(2)] **show** $\forall f = (\alpha e) \cap mon-pl (map \alpha$
\[ \text{IHAPP} \] = \{ \} by auto

\text{qed}

thus \( \text{thesis using IHAPP}(2) \) left' by blast

next

case (right' \( w' \)) from Cons.hyps[of - left(2) right'(2)] obtain \( wr \) where
IHAPP: \( w \in w1 \otimes \alpha \) \( wr \in w2' \otimes \alpha \) \( w3 \) 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 \( \text{thesis} \) using right' by blast

qed

next

case (right \( w3' \)) from Cons.hyps[of - right(2) Cons.prems(2)] obtain \( wr \)

where IHAPP: \( w \in w1 \otimes \alpha \) \( wr \in w2 \otimes \alpha \) \( w3' \) 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 \( \text{thesis} \) using right by blast

qed

qed

\text{lemma} cil-assoc2:

assumes \( A: w \in w1 \otimes \alpha \) \( wr \) and \( B: wr \in w2 \otimes \alpha \) \( w3 \)

shows \( \exists w1. w \in w1 \otimes \alpha \) \( w3 \) \( \land \) \( w1 \in w1 \otimes \alpha \) \( w2 \)

\text{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 \( \text{cil-assoc1}[OF A' B'] \) obtain \( w1 \) where \( w \in w3 \otimes \alpha \) \( w1 \) \( \land \) \( w1 \in w2 \otimes \alpha \) \( w1 \)

by blast

thus \( \text{thesis} \) by (auto simp add: cil-commute)

qed

— Parts of the abstraction can be moved to the operands

\text{lemma} cil-map: \( w \in w1 \otimes (\alpha f) \) \( w2 \) \( \Rightarrow \) map \( f \) \( w \in map \) \( f \) \( w1 \otimes \alpha \) map \( f \) \( w2 \)

\text{proof} (induct rule: cil-set-induct-fix\( \alpha \))

case empty thus \( \text{thesis} \) 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 \( \text{fst} \) \( ((\alpha f) e) \cap \text{mon-pl} \) \( \text{map} \) \( \alpha \) \( \text{map} \) \( f \) \( w2 \)\) = \( \{ \} \) by (simp only: map-map[ symmetric])

thus \( \text{fst} \) \( \alpha \) \( (f e) \) \( \cap \text{mon-pl} \) \( \text{map} \) \( \alpha \) \( \text{map} \) \( f \) \( w2 \)\) = \( \{ \} \) by (simp only: o-apply)

qed (rule left(3))

thus \( \text{?case} \) by simp

next

case (right \( e \) \( w' \) \( w2' \) \( w1 \))
have $f \neq map f w' \in map f w1 \otimes\alpha f \neq map f w2$. Proof (rule cil-cons2)

from right(2) have \( \text{fst } ((\alpha \circ f) e) \cap \text{mon-pl } (\text{map } \alpha (\text{map } f w1)) = \{\} \) by (simp only; map-map[symmetric])

thus \( \text{fst } (\alpha (f e)) \cap \text{mon-pl } (\text{map } \alpha (\text{map } f w1)) = \{\} \) 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 interleav-

ability. 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 ob-

servation 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]: \[ h \in ah; h m = \{\} \Rightarrow P ; m \in h m \Rightarrow P \] \Rightarrow 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** \( ah-il :: (\forall m \in 'm set) \Rightarrow bool \) (infix \( [\ast] \) 65)

where

\( h1 \ast h2 == \neg(\exists m1 m2. m1 \in h2 m2 \land m2 \in h1 m1) \)

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: \[ h1 \in ah; h2 \in ah; h1 \ast h2 \] \Rightarrow h1 m = \{\} \lor h2 m = \{\} \)

by (unfold ah-il-def) (auto elim: ah-cases)

Of course, acquisition history interleavability is commutative

**Lemma** \( ah-il-commute: h1 \ast h2 \Rightarrow h2 \ast h1 \)

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** \( mon-ah :: (\forall m \in 'm set) \Rightarrow 'm set \)

where

\( mon-ah h == \bigcup \{ h(m) | m. True\} \)

12
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: \[ h1 \; \text{[} * \; h2; h1' \leq h1; h2' \leq h2 \; \text{]} \implies h1' \; [\; * \; h2' \]
by (unfold ah-il-def le-fun-def \[ \text{where } 'b'=\text{'a set} \]) blast+

lemma ah-leq-il-left: \[ h1 \; [\; * \; h2; h1' \leq h1 \; ] \implies h1' \; [\; * \; h2' \]
and
ah-leq-il-right: \[ h1 \; [\; * \; h2; h2' \leq h2 \; ] \implies h1 \; [\; * \; h2' \]
by (unfold ah-il-def le-fun-def \[ \text{where } 'b'=\text{'a set} \]) 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 :: \( ('m set \times 'm set) \) list \( \Rightarrow \) 'm \( \Rightarrow \) 'm set where
\[ \alphaah \; [] \; m = \{\} \]
| \( \alphaah \; (e \# w) \; m = (if m \in \text{fst } e \text{ then } \text{fst } e \cup \text{snd } e \cup \text{mon-pl } w \text{ else } \alphaah \; w \; m) \]

— \( \alphaah \) generates valid acquisition histories

lemma αah-ah: \( \alphaah \; w \in ah \)
apply (induct w)
apply (unfold ah-def)
apply simp
apply (fastforce split: if-split-asm)
done

lemma αah-hd: \[ m \in \text{fst } e; \; x \in \text{fst } e \cup \text{snd } e \cup \text{mon-pl } w \] \implies \; \alphaah \; (e \# w) \; m
by auto

lemma αah-tl: \[ m \notin \text{fst } e; \; x \in \alphaah \; w \; m \] \implies \; \alphaah \; (e \# w) \; m
by auto

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

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


\[ \Rightarrow P \]

by \( \text{(simp-all split: if-split-asms)} \)

**lemma** mon-ah-subset: mon-ah \( (\alpha \text{ah} w) \subseteq \text{mon-pl} w \)

by \( \text{(induct w) (auto simp add: mon-ah-def)} \)

— Subwords generate smaller acquisition histories

**lemma** \( \alpha \text{ah-ileq}: w1 \preceq w2 \Rightarrow \alpha \text{ah} w1 \preceq \alpha \text{ah} w2 \)

**proof** \( \text{(induct rule: less-eq-list-induct)} \)

\text{case empty thus ?case by \( \text{(unfold le-fun-def [where 'b='a set], simp)} \)}

**next**

\text{case (drop l' l a) show ?case}

**proof** \( \text{(unfold le-fun-def [where 'b='a set], intro allI subsetI)} \)

fix \( m \)

assume \( x \in \alpha \text{ah} l \)

thus \( x \in \alpha \text{ah} l \)

with \( \text{drop(2) have x}\in\alpha \text{ah} l \)

moreover hence \( x\in\text{mon-pl} l \)

using \( \text{mon-ah-subset[unfolded mon-ah-def] by fast} \)

ultimately show \( x\in\alpha \text{ah} (a \# l) \)

by \( \text{auto} \)

qed

**next**

\text{case (take a b l' l) show ?case}

**proof** \( \text{(unfold le-fun-def [where 'b='a set], intro allI subsetI)} \)

fix \( m \)

assume \( A: x \in \alpha \text{ah} l \)

thus \( x \in \alpha \text{ah} (b \# l) \)

**proof** \( \text{(cases rule: ah-cons-cases)} \)

\text{case hd}

with \( \text{mon-pl-ileq[OF take.hyps(2)] and } (a = b) \)

show ?thesis by \( \text{auto} \)

**next**

\text{case tl}

with \text{take.hyps(3)[unfolded le-fun-def [where 'b='a set]] and } (a = b)

show ?thesis by \( \text{auto} \)

qed

qed

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

**lemma** ah-interleavable1:

\( w \in w1 \otimes \alpha w2 \Rightarrow \alpha \text{ah} (\text{map } \alpha w1) [\ast] \alpha \text{ah} (\text{map } \alpha w2) \)

— The lemma is shown by induction on the structure of the monitor consistent interleaving operator

**proof** \( \text{(induct w } \alpha w1 w2 \text{ rule: cil-set-induct-fixa)} \)

\text{case empty show ?case by \( \text{(simp add: ah-il-def)} \)}

— The base case is trivial by the definition of \( ([\ast]) \)

**next**

— Case: First step comes from the left word
\textbf{case} (left \(e w' \; w1' \; w2\)) \textbf{show} \ ?case
\begin{proof} (rule \texttt{cccontr}) \text{— We do a proof by contradiction} \\
\quad \text{— Assume there is a conflicting pair in the acquisition histories} \\
\quad \text{assume } \neg \; \alphaah \; (map \; \alpha \; (e \; \# \; w1')) \; [\ast] \; \alphaah \; (map \; \alpha \; w2) \\
\quad \text{then obtain } m1 \; m2 \; \text{where } \text{CPAIR}: \; m1 \in \alphaah \; (map \; \alpha \; (e\#w1')) \; m2 \; m2 \in \alphaah \; (map \; \alpha \; w2) \; \text{by } (\text{unfold ah-il-def, blast}) \\
\quad \text{— It comes either from the first step or not} \\
\quad \text{from } \text{CPAIR(1)} \; \text{have } (m2\in\text{fst } (\alpha \; e) \; \wedge \; m1 \in \text{fst } (\alpha \; e) \; \cup \; \text{snd } (\alpha \; e) \; \cup \; \text{mon-pl} \; (map \; \alpha \; w1')) \; \wedge \; (m2\notin\text{fst } (\alpha \; e) \; \wedge \; m1 \in \alphaah \; (map \; \alpha \; w1') \; m2) \; (\text{is } \text{?CASE1 } \lor \; \text{?CASE2}) \\
\quad \text{by } (\text{auto split: if-split-asm}) \\
\quad \text{moreover } \{
\quad \text{— Case: One monitor of the conflicting pair is entered in the first step of the left path} \\
\quad \quad \text{assume } \text{?CASE1 hence } C: \; m2\in\text{fst } (\alpha \; e) \; .. \\
\quad \quad \text{— Because the paths are consistently interleavable, the monitors entered in the first step must not occur in the other path} \\
\quad \quad \text{from left(2) mon-ah-subset[of map \; \alpha \; w2] have } \text{fst } (\alpha \; e) \; \cap \; \text{mon-ah } (\alphaah \; (\text{map } \alpha \; w2)) = \{\} \; \text{by auto} \\
\quad \quad \text{— But this is a contradiction to being a conflicting pair} \\
\quad \quad \text{with } C \; \text{CPAIR(2) have False by } (\text{unfold mon-ah-def, blast}) \\
\quad \text{) moreover } \{
\quad \text{— Case: The first monitor of the conflicting pair is entered after the first step of the left path} \\
\quad \quad \text{assume } \text{?CASE2 hence } C: \; m1 \in \alphaah \; (\text{map } \alpha \; w1') \; m2 \; .. \\
\quad \quad \text{— 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} \\
\quad \quad \text{with } \text{left(3) CPAIR(2) have False by } (\text{unfold ah-il-def, blast}) \\
\quad \text{) ultimately show False ..} \\
\text{qed}
\end{proof}

\textbf{next} \\
\text{— Case: First step comes from the right word. This case is shown completely analogous}
\begin{proof} (rule \texttt{cccontr}) \\
\quad \text{assume } \neg \; \alphaah \; (map \; \alpha \; w1) \; [\ast] \; \alphaah \; (map \; \alpha \; (e\#w2')) \\
\quad \text{then obtain } m1 \; m2 \; \text{where } \text{CPAIR}: \; m1 \in \alphaah \; (map \; \alpha \; w1) \; m2 \; m2 \in \alphaah \; (map \; \alpha \; (e\#w2')) \; m1 \; \text{by } (\text{unfold ah-il-def, blast}) \\
\quad \text{from } \text{CPAIR(2)} \; \text{have } (m1\in\text{fst } (\alpha \; e) \; \wedge \; m2 \in \text{fst } (\alpha \; e) \; \cup \; \text{snd } (\alpha \; e) \; \cup \; \text{mon-pl} \; (map \; \alpha \; w2')) \; \lor \; (m1\notin\text{fst } (\alpha \; e) \; \wedge \; m2 \in \alphaah \; (map \; \alpha \; w2') \; m1) \; (\text{is } \text{?CASE1 } \lor \; \text{?CASE2}) \\
\quad \text{by } (\text{auto split: if-split-asm}) \\
\quad \text{moreover } \{
\quad \text{assume } \text{?CASE1 hence } C: \; m1\in\text{fst } (\alpha \; e) \; .. \\
\quad \text{from right(2) mon-ah-subset[of map \; \alpha \; w1] have } \text{fst } (\alpha \; e) \; \cap \; \text{mon-ah } (\alphaah \; (\text{map } \alpha \; w1)) = \{\} \; \text{by auto} \\
\quad \text{with } C \; \text{CPAIR(1) have False by } (\text{unfold mon-ah-def, blast}) \\
\quad \text{) moreover } \{
\quad \text{assume } \text{?CASE2 hence } C: \; m2 \in \alphaah \; (map \; \alpha \; w2') \; m1 \; ..
\end{proof}
with right(3) CPAIR(1) have False by (unfold ah-il-def, blast)
} ultimately show False ..
qed

lemma ah-interleavable2:
assumes A: αah (map α w1) ∗ αah (map α w2)
shows w1 ⊗α w2 ≠ {}
— 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] ⇒ w1 ⊗α w2 ≠ {}]
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
next
— The interesting case is if both words are not empty
next
— 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) ∩ mon-pl (map α w2) = {})
— 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 ⊗α w2 ≠ {} proof —
from I.prems(1) CONS1 ah-leq-il-left[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 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) ∩ mon-pl (map α w1) = {})
— The first step of the second word can safely be executed
— This case is shown analogously to the latter one
have $w_1 \otimes \alpha w_2' \neq \{\}$ proof
  from $I$ prems(1) CONS2 ah-leq-il-right[OF - $\alpha$ ah-ileq[OF le-list-map, OF less-eq-list-drop[OF order-ref]]] have $\alpha ah (map \alpha w_1) [\ast] \alpha ah (map \alpha w_2')$
  by fast
moreover from CONS2 $I$ prems(2) have length $w_1 + \text{length } w_2' < n$ by simp
ultimately show ?thesis using $I$.hyps by blast
next
case False note C2 = this — Neither first step can safely be executed.
from $C1$ $C2$ obtain $m_1 m_2$ where $m_1 \in \text{fst } (\alpha e_1) m_1 \in \text{mon-pl } (\text{map } \alpha w_2)\ m_2 \in \text{fst } (\alpha e_2) m_2 \in \text{mon-pl } (\text{map } \alpha w_1)$ by blast
  with CONS1 CONS2 have $m_2 \in \alpha ah (\text{map } \alpha w_1) m_1 m_1 \in \alpha ah (\text{map } \alpha w_2) m_2$ 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

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

theorem ah-interleavable:
  $(\alpha ah (\text{map } \alpha w_1) [\ast] \alpha ah (\text{map } \alpha w_2)) \leftrightarrow (w_1 \otimes \alpha w_2' \neq \{\})$
  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 => \'m set) => (\'m set * \'m set) => \'m set => (\'m => \'m set)
where
  ah-update $h$ $F$ $M$ $m$ == if $m \in \text{fst } F$ then $\text{fst } F \cup \text{snd } F \cup M$ 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:** \( \alpha ah (e\#w) = ah-update (\alpha ah w) e \) (mon-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:** \( [h \leq h'; F=F'; M \subseteq M'] \)

\( \Rightarrow \) ah-update \( h F M \leq ah-update h' F' M' \)

by (auto simp add: ah-update-def le-fun-def)

**lemma ah-update-mono2:** \( [h \leq h'; U \subseteq U'; M \subseteq M'] \)

\( \Rightarrow \) ah-update \( h (E,U) M \leq ah-update h' (E,U') M' \)

by (auto simp add: ah-update-def le-fun-def)

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

\( empty[simp]:: (c,[]c) \in trcl t \)

\( cons[simp]:: [(c,a,c') \in t; (c',w,c'') \in trcl t] \Rightarrow (c,a\#w,c'') \in trcl t \)

4.2 Basic properties of transitive reflexive closure

**lemma trcl-empty-cons:** \( (c,[]c) \in trcl t \Rightarrow (c=c') \)

by (auto elim: trcl.cases)

**lemma trcl-empty-simp:** \( (c,[]c) \in trcl t = (c=c') \)

by (auto elim: trcl.cases intro: trcl.intros)
lemma trcl-single[simp]: \((c,[a],c') \in \text{trcl } t\) = \((c,a,c') \in t\)
by (auto elim: trcl_cases)

lemma trcl-uncons: \((c,a\#w,c')\in\text{trcl } t\) \implies \exists \text{ch} . \((c,a,\text{ch})\in t \land (\text{ch},w,c') \in \text{trcl } t\)
by (auto elim: trcl_cases)

lemma trcl-uncons-cases: \[
(c,e\#w,c')\in\text{trcl } S; \\
\forall \text{ch} . [(c,e,\text{ch})\in S; (\text{ch},w,c')\in\text{trcl } S] \implies P \\
\]
by (blast dest: trcl-uncons)

lemma trcl-one-elem: \((c,e,c')\in t \implies (c,[e],c')\in\text{trcl } t\)
by auto

lemma trcl-unconsE[cases set, case-names split]: \[
(c,e\#w,c')\in\text{trcl } S; \\
\forall \text{ch} . [(c,e,\text{ch})\in S; (\text{ch},w,c')\in\text{trcl } S] \implies P \\
\]
by (blast dest: trcl-uncons)

lemma trcl-pair-unconsE[cases set, case-names split]: \[
((s,c),e\#w,(s',c'))\in\text{trcl } S; \\
\forall \text{sh ch} . [[(s,c),e,(\text{sh},\text{ch})]\in S; ((\text{sh},\text{ch}),w,(s',c'))\in\text{trcl } S] \implies P \\
\]
by (fast dest: trcl-uncons)

lemma trcl-concat: \forall c . \[(c,w1,c')\in \text{trcl } t; (c',w2,c'')\in \text{trcl } t \] 
\implies (c,w1\#w2,c'')\in \text{trcl } t

proof (induct w1)
  case Nil thus \?case by (subgoat-tac c=c') auto
next
  case (Cons a w) thus \?case by (auto dest: trcl-uncons)
qed

lemma trcl-unconcat: \forall c . (c,w1\#w2,c')\in \text{trcl } t 
\implies \exists \text{ch} . (c,w1,\text{ch})\in \text{trcl } t \land (\text{ch},w2,c')\in \text{trcl } t

proof (induct w1)
  case Nil hence (c,[],c)\in \text{trcl } t \land (c,w2,c')\in \text{trcl } t by auto 
  thus \?case by fast
next
  case (Cons a w1) note IHP = this
  hence (c,a\#(w1\#w2),c')\in \text{trcl } t by simp
    with trcl-uncons obtain chh where (c,a,\text{chh})\in t \land (\text{chh},w1\#w2,c')\in \text{trcl } t by fast
  moreover with IHP obtain ch where (\text{chh},w1,\text{ch})\in \text{trcl } t \land (\text{ch},w2,c')\in \text{trcl } t by fast
  ultimately have (c,a\#w1,\text{ch})\in \text{trcl } t \land (\text{ch},w2,c')\in \text{trcl } t by auto 
  thus \?case by fast
qed

19
4.2.1 Appending of elements to paths

**Lemma trcl-rev-cons:** \[ (c, w, ch) \in \text{trcl } T; (ch, e, c') \in T \implies (c, w \@ e, c') \in \text{trcl } T \]
by (auto dest: trcl-concat iff add: trcl-single)

**Lemma trcl-rev-uncons:** \((c, w \@ e, c') \in \text{trcl } T \implies \exists ch. (c, w, ch) \in \text{trcl } T \land (ch, e, c') \in T\)
by (force dest: trcl-unconcat)

**Lemma trcl-rev-induct:**

**induct set, consumes 1, case-names empty snoc:**

\[ c', [\[(c, w, ch) \in \text{trcl } S; (ch, e, c') \in S; P c' w c'] \implies P (c, w \@ e, c') \]
by (force dest: trcl-rev-uncons)

**Lemma trcl-cons2:**

\[ (c, e, ch) \in \text{trcl } T; (ch, f, c') \in \text{trcl } T \implies (c, [e, f], c') \in \text{trcl } T \]
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:** \(!A \subseteq B \implies \text{trcl } A \subseteq \text{trcl } B\)
apply (clarsimp)
apply (erule trcl.induct)
apply auto
done

**Lemma trcl-inter-mono:** \(x \in \text{trcl } (S \cap R) \implies x \in \text{trcl } (S \cap R) \implies x \in \text{trcl } R\)
proof –
assume \(x \in \text{trcl } (S \cap R)\)
with trcl-mono[of \(S \cap R\)] show \(x \in \text{trcl } S\) by auto
next
assume \(x \in \text{trcl } (S \cap R)\)
with trcl-mono[of \(S \cap R\)] show \(x \in \text{trcl } R\) by auto
qed
4.2.4 Special lemmas for reasoning about states that are pairs

lemmas trcl-pair-induct = trcl.induct[of (xa1, xa2) xb (xa1, xa2), split-format (complete), consumes 1, case-names empty cons]
lemmas trcl-rev-pair-induct = trcl-rev-induct[of (xa1, xa2) 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]: 
If \((c, w, c') \in \text{trcl } S\);
\([c=c'; w=[]] \Rightarrow P\); 
\([c \in \text{Domain } S; c' \in \text{Range } (\text{Range } S)] \Rightarrow P\)
\([\] \Rightarrow 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 == \{ (\text{add-mset } s \ c, e, \text{add-mset } s' \ c') \mid s \ c \ e \ s \ c' \cdot ((s, c), e, (s', c')) \in gtrs \}
lemma gtrl-s; ((s, c), e, (s', c')) \in gtrs \Rightarrow (\text{add-mset } s \ c, e, \text{add-mset } s' \ c') \in \text{gtr } gtrs
by (unfold gtr-def , auto)
lemma gtrl1; ((s, c), w, (s', c')) \in \text{trcl } gtrs
\Rightarrow (\text{add-mset } s \ c, w, \text{add-mset } s' \ c') \in \text{trcl } (\text{gtr } gtrs)
by (induct rule: trcl-pair-induct) (auto dest: gtrl-s)
lemma \textit{gtrE}:
\[(c, e, c') \in \text{gtr} T;\]
\[!s \in s'; c' = \text{add-mset} s c; (s, (s', ce)) \in T \implies P\]
\[\begin{array}{l}
\text{by (unfold gtr-def) auto}
\end{array}\]

lemma \textit{gtr-empty-conf-s} [simp]:
\[((\#), w, c') \in \text{gtr} S;\]
\[(c, w, (\#)) \in \text{gtr} S\]
\[\begin{array}{l}
\text{by (auto elim: gtrE)}
\end{array}\]

lemma \textit{gtr-empty-conf1} [simp]:
\[((\#), w, c') \in \text{trcl} (\text{gtr} S)) \iff (w=\[] \land c'=[\#])\]
\[\begin{array}{l}
\text{by (induct w) (auto dest: trcl-uncons)}
\end{array}\]

lemma \textit{gtr-empty-conf2} [simp]:
\[((c, w, \#)) \in \text{trcl} (\text{gtr} S)) \iff (w=\[] \land c=\#)\]
\[\begin{array}{l}
\text{by (induct w) rule: rev-induct) (auto dest: trcl-rev-uncons)}
\end{array}\]

lemma \textit{gtr-find-thread}:
\[(c, e, c') \in \text{gtr} gtrs;\]
\[!s \in s'; c' = \text{add-mset} s c; (s, (s', ce)) \in gtrs\]
\[\implies P\]
\[\begin{array}{l}
\text{by (unfold gtr-def) auto}
\end{array}\]

lemma \textit{gtr-step-cases}[cases set, case-names loc other]:
\[(c, e, c') \in \text{gtr} gtrs;\]
\[!s' \in s'; \quad \text{add-mset} s c; (s, (s', ce)) \in gtrs\]
\[\implies P;\]
\[\begin{array}{l}
\text{by (auto elim!): gtr-find-thread mset-single-cases)
\end{array}\]

lemma \textit{gtr-rev-cases}[cases set, case-names loc other]:
\[(c, e, \text{add-mset} s' ce) \in \text{gtr} gtrs;\]
\[!s \in s'; \quad \text{add-mset} s c; (s, (s', ce)) \in gtrs\]
\[\implies P;\]
\[\begin{array}{l}
\text{by (auto elim!): gtr-find-thread mset-single-cases)
\end{array}\]

5.2 Invariants

lemma \textit{gtr-preserve-s}:
\[(c, e, c') \in \text{gtr} T;\]
\[P c;\]
\[!s \in s'; e. \quad \text{add-mset} s c; (s, (s', ce)) \in T\]
\[\implies P \quad (\text{add-mset} s' c')\]
\[\begin{array}{l}
\text{by (unfold gtr-def) blast}
\end{array}\]
lemma gtr-preserve: 
\[(c,w,c') \in \text{trcl} (\text{gtr } T); \]
\[P c; \]
\[!!s \ c ' e. [P (\text{add-mset } s \ c); ((s,c),e,(s',c')) \in T] \Longrightarrow P (\text{add-mset } s' \ c')\]
\[\] 
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 × 's multiset), 'l) LTS
assumes env-no-step-s[cases set, case-names csp]: 
\[\] 

— 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]: 
\[(s,c),w,(s',c') \in \text{trcl } gtrs; \]
\[!! csp. c'=csp+c \Longrightarrow P \] 
\[\] 
proof
have \[(s,c),w,(s',c') \in \text{trcl } gtrs \Longrightarrow \exists csp. c'=csp+c\] 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(I)] 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 \[(s,c),w,(s',c') \in \text{trcl } gtrs \] 
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

23
spawn The thread was spawned by the step

env The thread was not involved in the step

\textbf{lemma} (in \texttt{env-no-step}) \texttt{rev-cases-p[cases set, case-names loc spawn env]}:

\textbf{assumes} \texttt{STEP: (c,e,add-mset s' ce') \in gtrs and}
\texttt{LOC: \texttt{!}s ce. \texttt{[ c\in\{\texttt{#s\#}\}+ce; ((s,ce),e,(s',ce'))\in gtrs \texttt{]} \Rightarrow P and}
\texttt{SPAWN: !ss ss' ce csp.}
\texttt{\texttt{[ c=\texttt{add-mset ss ce}; ce'=\texttt{add-mset ss'} (csp+ce);}
\texttt{((s,ce),e,(ss',add-mset s' (csp+ce)))\in gtrs \texttt{]} \Rightarrow P and}
\texttt{ENV: !ss ss' ce csp.}
\texttt{\texttt{[ c=\texttt{add-mset ss (add-mset s' ce)}; ce'=\texttt{add-mset ss'} (csp+ce);}
\texttt{((ss,add-mset s' ce),e,(ss',csp+(add-mset s' ce)))\in gtrs \texttt{]} \Rightarrow P}

\textbf{shows} \texttt{P}

\textbf{proof} (rule \texttt{gtr-rev-cases}[OF \texttt{STEP}], \texttt{goal-cases})

\textbf{case 1} thus \texttt{?thesis using LOC by auto}

\textbf{next}

\textbf{case} \texttt{CASE: (2 cc ss ss' ce)}

\textbf{hence} \texttt{CASE': c = \{\texttt{#s\#}\} + ce \Rightarrow ce' = \{\texttt{#s\#'}\} + cc ((s, ce), e, ss', \{\texttt{#s\#'}\} + cc) \in gtrs by simp-all}

\textbf{from} \texttt{env-no-step-s[OF \texttt{CASE'(3)}]} \textbf{obtain} csp \textbf{where} \texttt{EQ: add-mset s' cc = csp + ce by auto}

\textbf{thus} \texttt{?thesis proof (cases rule: mset-anplus-dist-cases)}

\textbf{case left note} \texttt{CC=\texttt{this}}

\textbf{with} \texttt{CASE' have ce'=\{\texttt{#s\#'}\} + (csp-\{\texttt{#s\#'}\}) + ce by (auto simp add: union-associative)}

\textbf{moreover from} \texttt{CC(2) have} \texttt{\{\texttt{#s\#'}\}+cc = \{\texttt{#s\#'}\} + (csp-\{\texttt{#s\#'}\}) + ce}

\textbf{by (simp add: union-associative)}

\textbf{ultimately show} \texttt{?thesis using \texttt{CASE'(1,3) CASE(2) SPAWN by auto}}

\textbf{next}

\textbf{case right note} \texttt{CC=\texttt{this}}

\textbf{from} \texttt{CC(1) CASE'(1) have} \texttt{c=\texttt{add-mset ss (add-mset s' (ce - \{\texttt{#s\#'}\})) by (simp add: union-associative)}

\textbf{moreover from} \texttt{CC(2) CASE'(2) have} \texttt{ce'=\texttt{add-mset ss'} (csp+(ce-\{\texttt{#s\#'}\}))}

\textbf{by (simp add: union-associative)}

\textbf{moreover from} \texttt{CC(2) have} \texttt{add-mset s' cc = csp+(add-mset s' (ce-\{\texttt{#s\#'}\}))}

\textbf{by (simp add: union-associative)}

\textbf{ultimately show} \texttt{?thesis using \texttt{CASE'(3) CASE(3) CC(1) ENV by metis}}

\textbf{qed}

\textbf{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-ancons[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-anconc[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-ancons[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) = 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' 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 \((\text{add-mset } s \ (\text{add-mset } ss \ c), \ e, \ \text{add-mset } s \ (\text{add-mset } ss' \ c')) \in \text{gtr } T\) by
\((\text{auto simp add: add-mset-commute})\)
qed

\begin{verbatim}
lemma gtrp2gtr:
\[(s,c),w,(s',c')\in trcl (gtrp T) \implies (\text{add-mset } s \ c, \text{map le-rem-s } w, \text{add-mset } s' \ c')\in trcl (\text{gtr } T)
by (induct rule: trcl-pair-induct) (auto dest: gtrp2gtr-s)\]
\end{verbatim}

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

\begin{verbatim}
lemma (in env-no-step) gtr2gtrp[cases set, case-names gtrp]:
assumes A: (\text{add-mset } s \ c,w,c')\in trcl (gtr gtrs)
and CASE: !s' ce' ww. \[(c'=\text{add-mset } s' \ ce'; \ w=\text{map le-rem-s } ww; \ ((s,c),ww,(s',ce'))\in trcl (gtrp gtrs)]\n\implies P
shows P
proof
have \#s. c. \text{add-mset } s \ c,w,c')\in trcl (gtr gtrs) \implies \exists s' ce' ww. c'=\text{add-mset } s' ce' \w=\text{map le-rem-s } ww \\land ((s,c),ww,(s',ce'))\in trcl (gtrp gtrs)
proof (induct w)
case Nil thus \#case by auto
next
case (\text{Cons } e \ w) then obtain ch where SPLIT: (\text{add-mset } s \ c,e,ch)\in gtr gtrs
(ch,w,c')\in trcl (gtr gtrs) by (auto dest: trcl-unscons)
  from gtr2gtrp-s[OF SPLIT(1)] obtain sh ceh ee where FS: ch = \text{add-mset } sh \ ceh \ e = \text{le-rem-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'=\text{add-mset } s' ce' w=\text{map le-rem-s } ww \ ((sh,ceh),ww,(s',ce'))\in trcl (gtrp gtrs)
\end{verbatim}

27
ultimately have \((s,c)\in trcl\(\text{gtrp }\)gtrs)\) 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 \(A\): \(((s,c),(s',c'))\in gtrp T\)
and INIT: \(P\(\text{add-mset } s c\)\)
and PRES: \(s c s' c' e. [P (\text{add-mset } s c); ((s,c),(s',c'))\in T]\)
shows \(P (\text{add-mset } s' c')\)
proof –
from gtr-preserve-s[OF gtrp2gtr-s[OF A], where \(P=P\), OF INIT] PRES show \(P (\text{add-mset } s' c')\) by blast
qed

lemma gtrp-preserve:
assumes \(A\): \(((s,c),w,(s',c'))\in trcl (\text{gtrp } T)\)
and INIT: \(P (\text{add-mset } s c)\)
and PRES: \(s c s' c' e. [P (\text{add-mset } s c); ((s,c),(s',c'))\in T]\)
shows \(P (\text{add-mset } s' c')\)
proof –
from gtr-preserve[OF gtrp2gtr[OF A], where \(P=P\), OF INIT] PRES show \(P (\text{add-mset } s' c')\) 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**\( ('n, 'p, 'ba) edge = ('n × ('p, 'ba) edgeAnnot × 'n) \)

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

**definition**

\[ \text{initialproc } fg \ p = \ p = \text{main } fg \lor (\exists u \ v. (u, \text{Spawn } p, v) \in \text{edges } fg) \]

**lemma** main-is-initial[\(\text{simp}\)]: initialproc \( fg \ (\text{main } fg) \)

by (unfold initialproc-def) simp

**locale** flowgraph =

**fixes** \( fg :: ('n, 'p, 'ba, 'm, 'more) flowgraph-rec-scheme (\text{structure}) \)

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

### 6.2 Basic properties

**lemma** (in flowgraph) spawn-no-mon[\(\text{simp}\)]:

\((u, \text{Spawn } p, v) \in \text{edges } fg \implies \text{mon } fg \ p = {} \)

29
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 ==
  (\forall u. l. \neg(u,l,\text{return } fg \ p) \in \text{edges } fg) \land \text{entry } fg \ p \neq \text{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 \neq 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: \[
\text{\textit{initialproc}} \; fg \; p; \; (\text{\textit{entry}} \; fg \; p, l, v) \in \text{\textit{edges}} \; fg
\]
\[ \implies \exists p', l=\text{Call} \; p' \land \text{\textit{isolated-ret}} \; fg \; p' \]

6.4 Example Flowgraph

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

definition example-fg == 
  edges = \{(0::nat,0::nat),\text{Call} \; 1,(0,1)), \;(1,0),\text{Spawn} \; 0,(1,0)), 
  \;(1,0)\}, \;\text{\textit{Call}} \; 0, \; (1,0)), \;\}\}
  main = 0,
  entry = \lambda p. \; (p,0),
  return = \lambda p. \; (p,1),
  mon = \lambda p. \; \text{if} \; p=1 \; \text{then} \; \{0\} \; \text{else} \; \{\},
  \text{\textit{proc-of}} = \lambda (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
  apply (simp add: initialproc-def)
  apply (simp add: initialproc-def)
  apply (simp add: 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

\[
\text{mon-n fg n} == \text{mon fg (proc-of fg n)}
\]

**definition** — The monitors of a stack are the monitors of all its nodes

\[
\text{mon-s fg s} == \bigcup\{ \text{mon-n fg n} | n . n \in \text{set s} \}
\]

**definition** — The monitors of a configuration are the monitors of all its stacks

\[
\text{mon-c fg c} == \bigcup\{ \text{mon-s fg s} | s . s \in \# c \}
\]

— The monitors of a step label are the monitors of procedures that are called by this step

**definition** \text{mon-e} :: (’b, ’c, ’d, ’a, ’e) flowgraph-rec-scheme ⇒ (’c, ’f) label ⇒ ’a set where

\[
\text{mon-e fg e} = (\text{case e of (LCall p) ⇒ mon fg p | - ⇒ {}})
\]

**lemma** \text{mon-e-simps [simp]}:

\[
\begin{align*}
\text{mon-e fg (LBase a)} &= \{} \\
\text{mon-e fg (LCall p)} &= \text{mon fg p} \\
\text{mon-e fg (LRet)} &= \{} \\
\text{mon-e fg (LSpawn p)} &= \{}
\end{align*}
\]

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** \text{mon-w fg w} == \bigcup\{ \text{mon-e fg e} | e . e \in \text{set w} \}

**lemma** \text{mon-s-alt}: \text{mon-s fg s} == \bigcup (\text{mon fg ‘ proc-of fg ‘ set s})

by (unfold mon-s-def mon-n-def) (auto intro!: eq-reflection)

**lemma** \text{mon-c-alt}: \text{mon-c fg c} == \bigcup (\text{mon-s fg ‘ set-mset c})

by (unfold mon-c-def set-mset-def) (auto intro!: eq-reflection)

**lemma** \text{mon-w-alt}: \text{mon-w fg w} == \bigcup (\text{mon-e fg ‘ set w})
by (unfold mon-w-def) (auto intro!: eq-reflection)

**lemma** mon-sI: \[ [n \in \text{set } s; m \in \text{mon-n } fg \ n] \implies m \in \text{mon-s } fg \ s \]
  
  by (unfold mon-s-def, auto)

**lemma** mon-sD: \[ m \in \text{mon-s } fg \ s \implies \exists n \in \text{set } s. m \in \text{mon-n } fg \ n \]
  
  by (unfold mon-s-def, auto)

**lemma** mon-n-same-proc:
  proc-of \( fg \) \( n \) = proc-of \( fg \) \( n' \) \implies \text{mon-n } fg \( n \) = \text{mon-n } fg \( n' \)
  
  by (unfold mon-s-def, simp)

**lemma** mon-s-same-proc:
  proc-of \( fg \) \{\# set s\} = proc-of \( fg \) \{\# set s'\} \implies \text{mon-s } fg \( s \) = \text{mon-s } fg \( s' \)
  
  by (unfold mon-s-alt, simp)

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

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

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

**lemma** mon-s-single:
  \[ \text{mon-s } fg \ [n] = \text{mon-n } fg \ n \]
  
  by (unfold mon-s-def) auto

**lemma** mon-s-empty[simp]: \[ \text{mon-s } fg \ [\emptyset] = \{\} \]
  
  by (unfold mon-s-def) auto

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

**lemma** mon-s-unconc:
  \[ \text{mon-s } fg \ (a @ b) = \text{mon-s } fg \ a \cup \text{mon-s } fg \ b \]
  
  by (unfold mon-s-def) auto

**lemma** mon-s-uncons[simp]: \[ \text{mon-s } fg \ (a @ as) = \text{mon-n } fg \ a \cup \text{mon-s } fg \ as \]
  
  by (rule mon-s-unconc[where a=[a], simplified])

**lemma** mon-c-union-conc:
  \[ \text{mon-c } fg \ (a+b) = \text{mon-c } fg \ a \cup \text{mon-c } fg \ b \]
  
  by (unfold mon-c-def) auto

**lemma** mon-c-add-mset-unconc:
  \[ \text{mon-c } fg \ (\text{add-mset } x \ b) = \text{mon-s } fg \ x \cup \text{mon-c } fg \ b \]
  
  by (unfold mon-c-def) auto

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

**lemma** mon-cl: \[ [s \in\# c; m \in \text{mon-s } fg \ s] \implies m \in \text{mon-c } fg \ c \]
  
  by (unfold mon-c-def, auto)

**lemma** mon-cD:
  \[ m \in \text{mon-c } fg \ c \implies \exists s. \ s \in\# c \land m \in \text{mon-s } fg \ s \]
  
  by (unfold mon-c-def, auto)

**lemma** mon-s-mono:
  \[ set s \subseteq set s' \implies \text{mon-s } fg \ s \subseteq \text{mon-s } fg \ s' \]
  
  by (unfold mon-s-def) auto
lemma mon-c-mono: \( c \subseteq \# c' \implies \text{mon-c} fg c \subseteq \text{mon-c} fg c' \)
by (unfold mon-c-def) (auto dest: mset-subset-eqD)

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

lemma mon-w-single[simp]: \( \text{mon-w} fg \[e\] = \text{mon-e} fg e \)
by (unfold mon-w-def, auto)

lemma mon-w-unconc: \( \text{mon-w} fg (wa@wb) = \text{mon-w} fg wa \cup \text{mon-w} fg wb \)
by (unfold mon-w-def) auto

lemma mon-w-uncons[simp]: \( \text{mon-w} fg \#w = \text{mon-e} fg \#w \cup \text{mon-w} fg w \)
by (rule mon-w-unconc[where wa=[e], simplified])

lemma mon-w-ileq: \( w \preceq w' \implies \text{mon-w} fg w \subseteq \text{mon-w} fg w' \)
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 == \( \forall s s'. \{\# s, s'\#\} \subseteq \# c \implies \text{mon-s} fg s \cap \text{mon-s} fg s' = \{\} \)

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

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

lemma valid-split1: \( \text{valid} fg (c+c') \implies \text{valid} fg c \land \text{valid} fg c' \land \text{mon-c} fg c \cap \text{mon-c} fg c' = \{\} \)
apply (unfold valid-def)
apply (auto simp add: mset-le-incr-right)
apply (erule mon-cD)+
apply (subgoal-tac \{\#s\#\}+\{\#sa\#\} \subseteq \# c+c')
apply (auto dest!: multi-member-split)
done

lemma valid-split2: \([\text{valid} fg c; \text{valid} fg c'; \text{mon-c} fg c \cap \text{mon-c} fg c' = \{\}] \implies \text{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: \(\text{valid} fg (c+c') \longleftrightarrow (\text{valid} fg c \land \text{valid} fg c' \land \text{mon-c} fg c \cap \text{mon-c} fg c' = \{\})\)
by (blast dest: valid-split1 valid-split2)
lemma valid-add-mset-conc:
valid fg (add-mset x c) \iff (valid fg c' \land mon-s fg x \cap 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 = {} = \Rightarrow valid fg c
proof (unfold valid-def, intro allI impI)
  fix s s'
  assume A: mon-c fg c = {} and B: \{#s, s'\#\} \not\subseteq # c
  from mon-c-mono[OF B, of fg] A have mon-s fg s = {} by (auto simp add: mon-c-unconc)
  thus mon-s fg s \cap 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 \Rightarrow 'n list \Rightarrow bool where
 atU-s U [] = False
| atU-s U (u#r) = (u\in U)

lemma atU-s-decomp[simp]: atU-s U (s@ss') = (atU-s U s \lor (s=[] \land 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 == \exists s. s \in# c \land atU-s U s

lemma atU-fmt: [atU U c; !!ui r. [ui#r \in# c; ui\in 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) \land atU U (ce+c)
by (unfold atU-def) (auto simp add: union-ac)

lemma atU-union[simp]: atU U (c1+c2) = (atU U c1 \lor atU U c2)
by (auto simp add: atU-add elims: atU-union-cases)
lemma atU-empty[simp]: \( \neg \text{atU } U \{\#\} \)
by (unfold atU-def, auto)

lemma atU-single[simp]: \( \text{atU } U \{\#s\} = \text{atU-s } U s \)
by (unfold atU-def, auto)

lemma atU-single-top[simp]: \( \text{atU } U \{\#u\#r\} = (u \in U) \)
by (auto)

lemma atU-add-mset[simp]: \( \text{atU } U (\text{add-mset } c c2) = (\text{atU-s } U c \lor \text{atU } U c2) \)
unfolding add-mset-add-single[of c c2] atU-union by auto

lemma atU-xchange-stack: \( \text{atU } U (\text{add-mset } (u\#r) c) \implies \text{atU } U (\text{add-mset } (u\#r') c) \)
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 == \( \exists su sv. \{\#su\}\} + \{\#sv\} \subseteq \# c \land \text{atU-s } U su \land \text{atU-s } V sv \)

lemma atUV-empty[simp]: \( \neg \text{atUV } U V \{\#\} \)
by (unfold atUV-def) auto

lemma atUV-single[simp]: \( \neg \text{atUV } U V \{\#s\} \)
by (unfold atUV-def) auto

lemma atUV-union[simp]:
\( \text{atUV } U V (c1+c2) \iff (\text{atUV } U V c1) \lor (\text{atUV } U V c2) \lor (\text{atUV } U c1 \land \text{atU } V c2) \lor (\text{atUV } V c1 \land \text{atU } U c2) \)
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]:
\( \text{atUV } U V (\text{add-mset } c c2) \iff (\text{atUV } U V c2) \lor (\text{atU } U \{\#c\} \land \text{atU } V c2) \lor (\text{atU } V \{\#c\} \land \text{atU } U c2) \)
unfolding add-mset-add-single[of c c2]
unfolding atUV-union
by auto

36
lemma atUV-union-cases[case-names left right lr rl, consumes 1]: [ 
  atUV U V (c1+c2);
  atUV U V c1 \implies P;
  atUV U V c2 \implies P;
  [atUV U c1; atUV V c2] \implies P;
  [atUV V c1; atU U c2] \implies P
] \implies 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 \Rightarrow
  ('n conf \times ('p,'ba) label \times 'n conf) set

for fg
where
  — A base edge transforms the top node of one stack and leaves the other stacks untouched.
  refpoint-base: \[ (u,Base a,v) \in \text{edges} \; fg; \; \text{valid} \; fg (\{\#u\#r\#\}+c) \] \Rightarrow
  (add-mset (u\#r) c,LBase a,add-mset (v\#r) c) \in \text{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: \[ (u,Call p,v) \in \text{edges} \; fg; \; \text{valid} \; fg (\{\#u\#r\#\}+c);
  \text{mon} \; fg \; p \cap \text{mon-c} \; fg \; c = \{\} \] \Rightarrow
  (add-mset (u\#r) c,LCall p, add-mset (entry \; fg \; p \#v\#r\#) c) \in \text{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: \[ \text{valid} \; fg (\{\#\text{return} \; fg \; p\#r\#\}+c) \] \Rightarrow
  (add-mset (return \; fg \; p\#r\#) c,LRet,(add-mset r c)) \in \text{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: \[ (u,Spawn p,v) \in \text{edges} \; fg; \; \text{valid} \; fg (\text{add-mset} (u\#r) c) \] \Rightarrow
  (add-mset (u\#r) c,LSpawn p, add-mset (v\#r) (add-mset (entry \; fg \; p) c)) \in \text{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

\[
\text{trss} :: ('(n,'p,'ba,'m,'more)) \text{ flowgraph-rec-scheme} \Rightarrow
(('(n \text{ list} \ast \text{'n conf}) \ast (\text{'p,'ba}) \text{ label} \ast (\text{'n \text{ list} \ast \text{'n conf})) \ast ) \text{ set}
\]

for \( fg \)

where

\[
\begin{align*}
\text{trss-base} : & \quad [(u,\text{Base} a,v) \in \text{edges} \ fg] \Rightarrow \\
& ((u\#r,c), \text{LBase} a, (v\#r,c) \in \text{trss} \ fg) \\
\mid \text{trss-call} : & \quad [(u,\text{Call} p,v) \in \text{edges} \ fg; \text{mon} \ fg \ p \cap \text{mon-c} \ fg \ c = \{\} ] \Rightarrow \\
& ((u\#r,c), \text{LCall} p, ((\text{entry} \ fg \ p)\#v\#r,c)) \in \text{trss} \ fg \\
\mid \text{trss-ret} : & \quad (((\text{return} \ fg \ p)\#r,c), \text{LRet}(r,c)) \in \text{trss} \ fg \\
\mid \text{trss-spawn} : & \quad [(u,\text{Spawn} p,v) \in \text{edges} \ fg] \Rightarrow \\
& ((u\#r,c), \text{LSpawn} p, (v\#r,\text{add-mset} [\text{entry} \ fg \ p] \ c)) \in \text{trss} \ fg
\end{align*}
\]

— The interleaving semantics is generated using the general techniques from Section 5

abbreviation \( tr \) where \( tr \ fg \) \( = \ gtr \ (\text{trss} \ fg) \)

— We also generate the \( \text{loc/env-}\) semantics

abbreviation \( trp \) where \( trp \ fg \) \( = \ gtrp \ (\text{trss} \ fg) \)

7.6 Basic properties

7.6.1 Validity

lemma (in \text{flowgraph}) \text{trss-valid-preserve-s}:

\[
[\text{valid} \ fg \ (\text{add-mset} \ s \ c); ((s,c), t, (s',c')) \in \text{trss} \ fg] \Rightarrow \text{valid} \ fg \ (\text{add-mset} \ s' \ c')
\]

apply (erule \text{trss-cases})

apply (simp-all add: \text{valid-unconc} \text{ mon-c-unconc})

by (blast dest: \text{mon-n-same-proc} \text{ edges-part}+)

lemma (in \text{flowgraph}) \text{trss-valid-preserve}:

\[
[((s,c), w, (s',c')) \in \text{trcl} \ (\text{trss} \ fg); \text{valid} \ fg \ \{s#\} + c] \Rightarrow \text{valid} \ fg \ \{\#s'\} + c'
\]

by (induct rule: \text{trcl-pair-induct}) (auto intro: \text{trss-valid-preserve-s})

lemma (in \text{flowgraph}) \text{tr-valid-preserve-s}:

\[
[(c,c',c') \in \text{tr}(fg); \text{valid} \ fg \ c] \Rightarrow \text{valid} \ fg \ c'
\]

by (rule \text{gtr-validate-s}[\text{where} \ P=\text{valid} \ fg]) (auto dest: \text{trss-valid-preserve-s})

lemma (in \text{flowgraph}) \text{tr-valid-preserve}:

\[
[(c,w,c') \in \text{tr}(fg); \text{valid} \ fg \ c] \Rightarrow \text{valid} \ fg \ c'
\]

by (rule \text{gtr-validate}[\text{where} \ P=\text{valid} \ fg]) (auto dest: \text{trss-valid-preserve-s})

lemma (in \text{flowgraph}) \text{trp-valid-preserve-s}:

\[
[((s,c), e, (s',c')) \in \text{trp} \ fg; \text{valid} \ fg \ (\text{add-mset} \ s \ c)] \Rightarrow \text{valid} \ fg \ (\text{add-mset} \ s' \ c')
\]

by (rule \text{gtrp-validate-s}[\text{where} \ P=\text{valid} \ fg]) (auto dest: \text{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 \(\Rightarrow\) ((c,e,c')\in refpoint fg) \(\iff\) ((c,e,c')\in tr fg)

**Proof**

- **Have** ((c,e,c')\in trcl (refpoint fg)) \(\Rightarrow\) valid fg c \(\Rightarrow\) ((c,e,c')\in trcl (tr fg))
  - **Induct** rule: trcl.induct (auto simp add: refpoint-eq-s tr-valid-preserve-s)
  - **Moreover have** ((c,e,c')\in trcl (tr fg)) \(\Rightarrow\) valid fg c \(\Rightarrow\) ((c,e,c')\in trcl (refpoint fg))
  - **Induct** rule: trcl.induct (auto simp add: refpoint-eq-s tr-valid-preserve-s)

**Ultimately show** valid fg c \(\Rightarrow\) ((c,e,c')\in trcl (refpoint fg)) = ((c,e,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'\in c ] \(\Rightarrow\) P;
  - [!!p u v. [\forall e=LSpawn p; (u,Spawn p,v)\in edges fg;
      \exists sd s'=u; hd s'=v; c'=\{#| entry fg p |#}\}+c ] \(\Rightarrow\) P

**By** (force elim!: trss-cases-s)

**Lemma** trss-c-fmt-s: 

- ((s,c),e,(s',c'))\in trss fg]
  - \(\Rightarrow\) \exists csp. c'=csp+c ∧ (csp={#} \lor (\exists p. e=LSpawn p \land csp={#\} entry fg p |#}))

**By** (force elim!: trss-c-cases-s)

**Lemma** trss-c'-split-s: 

- ((s,c),e,(s',c'))\in trss fg; csp;
  - [ c'=csp+c; mon-c fg csp={[]} ] \(\Rightarrow\) P

**By** (force elim!: trss-c-cases-s)

**Apply** (erule trss-c-cases-s)
**Apply** (subgoal-tac c'=\{#\}+c)
**Apply** (fastforce)
**Apply** auto
lemma \( \text{trss-c-cases} \): \( \forall s\ c\ [\]

\[
((s,c),w,(s',c'))\in \text{trcl} \ (\text{trss } fg);
\]

\( \neg\text{csp} \) \[
[\]
\[
[\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
\[
(\]
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)\in 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')\in trcl (trss fg) \Longrightarrow mon-c fg c' = mon-c fg c
apply (auto elim!: trss-c-cases simp add: mon-c-anconc)
proof –
  fix s where s \in\# csp and M: x \in mon-s fg s by (unfold mon-c-def, auto)
moreover assume \( \forall s, s \in\# csp \rightarrow (\exists p. s = [entry fg p] \land (\exists u v. (u, Spawn p, v) \in edges fg) \land initialproc fg p) \)
ultimately obtain p u v where s=[entry fg p] and (u,Spawn p,v)\in edges fg
hence mon-s fg s = {} by simp
with M have False by simp
thus x \in mon-c fg c .
qed

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

lemma trss-no-empty-s[simp]: ([],c),(s',c')\in trss fg = False 
by (auto elim!: trss.cases)
lemma trss-no-empty[simp]:
  assumes A: ([],c),(s',c')\in trcl (trss fg)
  shows w=[] \land s'=[] \land c=c'
proof –
  note A
moreover {
  fix s
  have ((s,c),w,(s',c')\in trcl (trss fg) \Longrightarrow s=[] \Longrightarrow w=[] \land s'=[] \land c=c'
    by (induct rule: trcl-pair-induct) auto
} ultimately show ?thesis by blast
qed
lemma trs-step-cases[cases set, case-names NO-SPA WN SPAWN]:
assumes A: (c,e,c') ∈ tr fg
assumes A-NO-SPA WN: !! s ce s' csp. [ (s,ce),(s',ce) ∈ trss fg; c = {#s#} + ce; c' = {#s'#} + ce ] ⇒ P
assumes A-SPA WN: !! s ce s' p. [ (s,ce),LSpawn p,(s',{#entry fg p[#]}+ce) ∈ trss fg; c = {#s#} + ce; c' = {#s'#} + {#entry fg p[#]} + ce; e = LSpawn p ] ⇒ P
shows P
proof –
from A show ?thesis proof (erule-tac gtr-find-thread)
fix s ce s' ce'.
assume FMT: c = add-mset s ce c' = add-mset s' ce'
assume B: ((s, ce), e, s', ce') ∈ trss fg thus ?thesis proof (cases rule: trss-c-cases-s)
  case no-spawn thus ?thesis using FMT B by (−) (rule A-NO-SPA WN, auto)
next
  case (spawn p) thus ?thesis using FMT B by (−) (rule A-SPA WN, auto simp add: union-assoc)
qed
qed

7.7 Advanced properties

7.7.1 Stack composition / decomposition

lemma trss-stack-comp-s:
((s,c),e,(s',c')) ∈ trss fg =⇒ ((s @ r, c), e, (s' @ r, c')) ∈ trss fg
by (auto elim!: trss_cases intro: trss_intros)

lemma trss-stack-comp:
((s,c),w,(s',c')) ∈ trcl (trss fg) =⇒ ((s @ r, c), w, (s' @ r, c')) ∈ 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) ∈ trss fg .
  also note IHP(3)
finally show ?case .
qed

lemma trss-stack-decomp-s:
[(s @ r, c), e, (s', c') ∈ trss fg; s ≠ []]
⇒ ∃ sp'. s' = sp' @ r ∧ ((s, c), e, (sp', c') ∈ trss fg
proof

— For $s \neq []$, we use induction by $w$

have IM: \(!s \ c. \ [((s@r,c),w,(r,c'))\in trcl (trss fg); \ s\neq []] \Longrightarrow \exists wa wb ch. w=wa@wb \land ((s,c),wa,([],ch))\in trcl (trss fg) \land ((r,ch),wb,(r,c'))\in trcl (trss fg)\)

proof (induct $w$)

  case Nil thus \(?case by (auto)\)

next

  case (Cons c w) note IHP=this

    then obtain sh ch where SPLIT1: \(((s@r,c),c,(sh,cb))\in trcl (trss fg)\) and SPLIT2: \(((sh,cb),w,(rc))\in trcl (trss fg)\) by (fast dest: trcl-unscons)

    \{ assume CASE: c=LRet

      with SPLIT1 obtain p where EDGE: s@r=return fg p \# sh cb 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\neq []

        with SHFMT have \(\exists ss. ss\neq [] \land sh=ss@r\) by blast

        moreover \{

          assume CC: ss=[]

          with CASE SHFMT EDGE have ((s,c),[e],[[],cb])\in trcl (trss fg) e#w=[e]@w by (auto intro: trss-ret)

          moreover from SPLIT2 SHFMT CC have ((r,cb),w,(rc))\in trcl (trss fg) by simp

          ultimately have \(?case by blast\)

          } ultimately have \(?case \lor (\exists ss. ss\neq [] \land sh=ss@r)\) by blast

          moreover \{

            assume e\neq LRet

            with SPLIT1 IHP(3) have \(\exists ss. ss\neq [] \land sh=ss@r\) by (force elim!: trss.cases simp add: append-eq-Cons-conv)

            \} moreover \{

              assume (\exists ss. ss\neq [] \land sh=ss@r)

              then obtain ss where CASE: ss\neq [] \land sh=ss@r by blast

              with SPLIT2 have \(((s@r,c), w, r, c') \in trcl (trss fg)\) by simp

            from IHP(1)[OF this CASE(1)] obtain wa wb cb where IHAPP: w=wa@wb ((ss,cb),wa,([],cb))\in trcl (trss fg) ((r,cb'),wb,(r,c'))\in trcl (trss fg) by blast

            moreover from CASE SPLIT1 have ((s@r, c), e, ss@r, ch) \in trss fg by simp

            from trss-stack-decomp-s[OF this IHP(3)] have ((s, c), e, ss, cb) \in trss fg by auto

        \}

    \}

— If $s = []$, the proposition follows trivially

apply (cases $s=[]$)

apply fastforce

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

\!
wa \ wb \ ch. \ w=wa@wb \land ((s,c),wa,([],ch))\in trcl (trss fg);

(r,ch),wb,(r,c'))\in trcl (trss fg) \Longrightarrow P\]

by (cases $s$, simp) (auto intro: trss.intros elim!: trss.cases)
with IHAPP have \(((s, c), e \# w, (\llbracket, ch')\}) \in \text{trcl}(\text{trss}\ fg)\) by (rule-tac \text{trcl.cons})

moreover from IHAPP have \(e \# w=\(e \# w)\# w\) by auto

ultimately have \(?case\) by blast
\}
ultimately show \(?case\) by blast

qed

assume \(((s \oplus r, c), w, r, c') \in \text{trcl}(\text{trss}\ fg)\) \(s \neq []\) !! \(wa\) \(wb\) \(ch\). [] \(w=wa\# wb; (s,c),wa,(\llbracket, ch))\) \(\in\) \text{trcl}(\text{trss}\ fg); \((r, ch), wb, (r', c')\) \(\in\) \text{trcl}(\text{trss}\ fg) \(\Rightarrow\) \(P\) thus \(P\)

by (blast dest: \text{IM})

qed

\textbf{lemma} \text{trss-return-cases[cases set]}: !! \(u\) \(r\) \(c\).

\(((u \# r, c), w, (r', c')) \in \text{trcl}(\text{trss}\ fg)\);

\((\llbracket, u\# r, c') \in \text{trcl}(\text{trss}\ fg)\) \(\Rightarrow\) \(P\);

\(!!\) \(wa\) \(wb\) \(ch\). [] \(w=wa\# wb; (\llbracket, u\# c))\) \(\in\) \text{trcl}(\text{trss}\ fg); 

\(\Rightarrow\) \(P\)

proof (induct \(w\) rule: \text{length-compl-induct})

case \text{Nil} thus \(?case\) by auto

next

case (\text{Cons} \(e\) \(w\)) note \text{IHP=this}

then obtain \(sh\) \(ch\) where \text{SPLIT1}: \(((u \# r, c), e, (sh, ch)) \in \text{trss}\ fg\) and \text{SPLIT2}: 

\(((sh, ch), w, (r', c')) \in \text{trcl}(\text{trss}\ fg)\) by (fast dest: \text{trcl-ancons})

\{

fix \(ba\) \(q\)

assume \text{CASE}: \(e=L\text{Base}\ ba \lor e=L\text{Spawn}\ q\)

with \text{SPLIT2} obtain \(v\) where \(E\): \(sh=v\# r\ ((\llbracket, u\# c), e, ([\llbracket], ch)) \in \text{trss}\ fg\) by (auto elim!: \text{trss.cases intro: \text{trss.intros}})

with \text{SPLIT2} have \(((v \# r, ch), w, (r', c')) \in \text{trcl}(\text{trss}\ fg)\) by simp

hence \(?case\) proof (cases rule: \text{IHP}(1)[of \(w\), simplified, cases set])

case (1 \(s\# u\)) note \text{CC=this}

with \(E(2)\) have \(((u\# [\llbracket], c), e) \# w, (s \# [u\# c', c') \in \text{trcl}(\text{trss}\ fg)\) by simp

from \text{IHP}(3)[OF \text{CC}(1) \ this] show \(?thesis\).

next

case (2 \(wa\ wb\) \(ct\)) note \text{CC=this}

with \(E(2)\) have \(((\llbracket, u\# c), e \# wa, ([\llbracket], ct)) \in \text{trcl}(\text{trss}\ fg)\) \(e \# w = (e \# wa) \# wb\) by simp-all

from \text{IHP}(4)[OF \ this(2) \ 1 \ CC(3)] show \(?thesis\) .

qed

\}
moreover {\}

assume \text{CASE}: \(e=L\text{Ret}\)

with \text{SPLIT1} have \(sh=r\ (((u\# c), [\llbracket], c)) \in \text{trcl}(\text{trss}\ fg)\) by (auto elim!: \text{trss.cases intro: \text{trss.intros}})

with \text{IHP}(4)[OF - \ this(2)] \text{SPLIT2} have \(?case\) by auto

moreover {\}

fix \(q\)

assume \text{CASE}: \(e=L\text{Call}\ q\)

with \text{SPLIT1} obtain \(u\) \(where\ \text{SHFMT}: sh=\text{entry}\ fg\ q \# u \# r\ (((u\# c), e, ([\llbracket], ch) \in \text{trcl}\ (\text{trss}\ fg) .

qed
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#w, [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), CCC(1)]
  CCC(1) have e#w = (e#w @ 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 ?thesis by (cases e, auto)

  qed

lemma (in flowgraph) trss-find-call:

  !v r' c'. [ (((sp],c),w,(v#r',c')) ∈ trcl (trss fg); r'≠[] ]

  ⇒ ∃ r h ch p wa wb.

  w = wa@[LCall p]#wb ∧

  proc-of fg v = p ∧

  (((sp],c),wa,(r,h,ch)) ∈ trcl (trss fg) ∧

  ((r,h,ch),LCall p,((entry fg p)#r',ch)) ∈ trcl (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 RHFM[i 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: \( w = wa \circ LCall \ p \# \ \text{wb} \ \text{proc-of} \ fg \ u = p \ (([sp], c), wa, (rt, ct)) \in \text{trcl} \ (trss \ fg) \ ((rt, ct), LCall \ p, \ \text{entry} \ fg \ p \# \ r', ct) \in \text{trss} \ fg \ ((\{\text{entry} \ fg \ p\}, ct), \ \text{wb}, ([u], ch)) \in \text{trcl} \ (trss \ fg) \ \text{by (blast)} \)

moreover

have \( ((\{\text{entry} \ fg \ p\}, ct), \ \text{wb}@[e], ([v], c')) \in \text{trcl} \ (trss \ fg) \ \text{proof} – \)

note IHAPP(5)

also from SPLIT2 have \( (\{u\}, ch), e, ([v], c') \in \text{trss} \ fg \ \text{by (auto elim!): trss.cases intro!: trss.intros}) \)

finally show ?thesis .

qed

moreover have \( (\exists u, rh = u\#r') \lor \ ?case \)

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]: \( rh = (return \ \text{fg} \ q) \# v \# r' \) by simp

with \( \text{IHPP}(1)(of \ w \ \text{return} \ \text{fg} \ q) \ v \# r' \ ch, \ \text{OF - SPLIT1[simplified]} \) obtain rt ct wa wb where

IHAPP: \( w = wa \circ LCall \ q \# \ \text{wb} \ (\{sp\}, \ c), wa, rt, ct) \in \text{trcl} \ (trss \ fg) \ ((rt, ct), LCall \ q, \ \text{entry} \ fg \ q \# v \# r', ct) \in \text{trss} \ fg \ ((\{\text{entry} \ fg \ q\}, ct), \ \text{wb}, ([\text{return} \ \text{fg} \ q], ch) \in \text{trcl} \ (trss \ fg) \ \text{by force}) \)

then obtain u where RTFMT [simp]: \( rt = u\#r' \) and PROC-OF-U: proc-of fg u = proc-of fg v by (auto elim: trss.cases intro: edges-part)

from IHAPP(1) have LENWA: length \( wa \leq \text{length} \ w \) by auto

from IHPP(1)(OF LENWA IHAPP(2)[simplified] IHP(3)) obtain rhh chh p waa wb where

IHAPP': \( \text{waa}@[\text{LCall} \ p \# \ \text{wb} \ \text{proc-of} \ fg \ u = p \ (\{\text{sp}, c\}, \text{waa}, (\text{rhh}, \text{chh})) \in \text{trcl} \ (trss \ fg) \ ((\text{rhh}, \text{chh}), LCall \ p, \ \text{entry} \ fg \ p \# r', \text{chh}) \in \text{trss} \ fg \ ((\{\text{entry} \ fg \ p\}, \text{chh}), \text{waa}, ([u], ct)) \in \text{trcl} \ (trss \ fg) \) by blast

by blast

from IHAPP IHAPP' PROC-OF-U have \( w@[e] = waa@LCall \ p \# (\text{waa}@LCall \ q\#\text{wb}@[e]) \land \ \text{proc-of} \ fg \ v = p \ \text{by auto}

moreover have \( ((\{\text{entry} \ fg \ p\}, \text{chh}), \text{waa}@LCall \ q\#\text{wb}@[e], ([v], c') \in \text{trcl} \ (trss \ fg) \) \ \text{proof} — \)

note IHAPP'(5)

also from IHAPP have \( ((u, ct), LCall \ q, \ \text{entry} \ fg \ q \# [v], ct) \in \text{trss} \ fg \ \text{by (auto elim!: trss.cases intro!: trss.intros}) \)

also from trss-stack-comp[OF IHAPP(4)] have \( ((\text{entry} \ fg \ q\#[v], ct), \text{wb}, (\text{return} \ \text{fg} \ q\#[v], ch)) \in \text{trcl} \ (trss \ fg) \ \text{by simp} \)

also from CC have \( ((\text{return} \ \text{fg} \ q\#[v], ch), e, ([v], c') \in \text{trss} \ fg \) by (auto intro:
trss-ref)

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'])) ∈ trcl (trss fg)
and EX: !!uh ch wa wb. [w=wa@(LCall p)#wb;
(!([sp],c),wa,(uh),ch))∈trcl (trss fg);
(!([uh],ch),LCall p,([entry fg p) #[u',ch])∈trss fg;
(uh,Call p,u')∈edges fg;
(!([entry fg p],ch),wb,([return fg p],c'))∈trcl (trss fg)
] ⟹ 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) ∈ trcl (trss fg)
(!([eh],ch),LCall p,([entry fg p, u'], ch) ∈ trss fg
(!([entry fg p], ch), wb, ([return fg p], c') ∈ trcl (trss fg)
by auto
moreover from FC(3) obtain uh where ADD: rh=[uh] (uh,Call p,u')∈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'))∈trcl (trss fg)
⟹ proc-of fg u = proc-of fg u'
proof (induct w rule: rev-induct)
case Nil thus ?case by auto
next
\begin{enumerate}
\item case (snoc e w) note IHAPP=this then obtain sh ch where SPLIT1: ((s@[u],c),w,(sh,ch))∈trcl (trss fg) and SPLIT2: ((sh,ch),c,(s'@[u'],c'))∈trss fg by (fast dest: trcl-rev-unscons)
\item from SPLIT2 have sh#[] by (auto elim!: trss.cases)
\item then obtain ssh uh where SHFMT: sh=ssh@[uh] by (blast dest: list-rev-decomp)
\item with IHAPP(1) of ssh uh ch] SPLIT1 have proc-of fg u = proc-of fg uh by auto
\item also from SPLIT2 SHFMT have proc-of fg uh = proc-of fg u' by (cases rule: trss.cases) (cases ssh, auto simp add: edges-part)+
\item finally show ?case .
\end{enumerate}
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:

\[(\text{entry } fg\ p,c,w,([\text{return } fg\ q],c'))\in\text{trcl}\ (\text{trss } fg) \implies p=q\]

**using** trss-bot-proc-const[of \(\llbracket\text{entry } fg\ p\rrbracket\) return \(\llbracket\text{return } fg\ q\rrbracket\), simplified].

**lemma** trss-2empty-to-2return: \(\exists w'.\ w=w'@[\text{LRet}] \land ((s,c),w',([\text{return } fg\ p],c'))\in\text{trcl}\ (\text{trss } fg)\)

**proof** —

**assume** \((s,c),w,([\cdot],c'))\in\text{trcl}\ (\text{trss } fg)\ s\neq\llbracket\cdot\rrbracket\)

**hence** \(w\neq\llbracket\cdot\rrbracket\) by **auto**

**then obtain** \(w'\) \(\in\) \(\text{where WD: } w=w'@[\text{LRet}]\) by (blast dest: list-rev-decomp)

**with** \(A(1)\) obtain \(sh\ ch\) \(\text{where SPLIT: } ((s,c),w',(sh,ch))\in\text{trcl}\ (\text{trss } fg)\ ((sh,ch),e,([\cdot],c'))\in\text{trss } fg\) by (fast dest: trcl-rev-uncons)

**from** SPLIT(2) obtain \(p\) \(\in\) \(\text{where } e=\text{LRet}\) \(sh=[\text{return } fg\ p]\) \(ch=c'\) by (cases rule: trss.cases, auto)

**with** SPLIT(1) WD show ?thesis by blast

**qed**

**lemma** trss-2return-to-2empty: \(\exists w'.\ w=w'@[\text{LRet}] \land ((s,c),w',([\text{return } fg\ p],c'))\in\text{trcl}\ (\text{trss } fg)\)

**apply** (subgoal-tac ([return fg p],c'),LRet,([\cdot],c'))\in\text{trss } fg)

by (auto dest: trcl-rev-cons intro: trss.intros)

7.7.2 Adding threads

**lemma** trss-env-increasing-s: \((s,c),e,(s',c')\in\text{trss } fg\) \(\implies c\subseteq\#e'\)

by (auto elim!: trss.cases)

**lemma** trss-env-increasing: \((s,c),w,(s',c')\in\text{trcl}\ (\text{trss } fg)\) \(\implies c\subseteq\#e'\)

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:

\((s,c),e,(s',c')\in\text{trss } fg\) \(\implies\) \(\text{mon-e } fg\ e\cap\text{mon-c } fg\ c\ =\ \{\}\)

by (erule trss.cases) auto

**lemma** (in flowgraph) trss-mon-w-no-ctx:

\((s,c),w,(s',c')\in\text{trcl}\ (\text{trss } fg)\) \(\implies\) \(\text{mon-w } fg\ w\cap\text{mon-c } fg\ c\ =\ \{\}\)

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:

\(\exists cn.\ ((s,c),e,(s',c')\in\text{trss } fg;\ \text{mon-e } fg\ e\cap\text{mon-c } fg\ cn\ =\ \{\})\)

\(\implies\) \(\exists\ csp.\ e=csp+c\wedge\text{mon-c } fg\ csp\ =\ \{\}\wedge((s,cn),e,(s',csp+cn))\in\text{trss } fg\)

by (erule trss.cases) (auto intro!: trss.intros)

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

\(\llbracket((s,c),w,(s',c')\in\text{trcl}\ (\text{trss } fg))\]

\(\implies\) \(\forall cn.\ \text{mon-w } fg\ w\cap\text{mon-c } fg\ cn\ =\ \{\})\)

\(\implies\) \(\exists\ csp.\ e=csp+c\wedge\text{mon-c } fg\ csp\ =\ \{\}\wedge((s,cn),w,(s',csp+cn))\in\text{trcl}\ (\text{trss } fg))\)

48
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 refl)
    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-fg 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 cs where S2: c' = csph + ch mon-c fg csph = {} ((sh, csph + cn), w, (s', csph + (csph + cn))) ∈ trcl (trss fg) by blast
    then have S1 S2 have c' = (csph + csph) + c mon-fg fg (csph + csph) = {} ((s, cn), e # w, (s', (csph + csph) + cn)) ∈ trcl (trss fg) by (auto simp add: mon-c-unconc)
    thus (s, c, w, (s', c' + ce)) ∈ mon-w fg w ∩ mon-c fg ce = {} ∧ ((s, cn), e # w, s', csph + 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), 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, 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))\(\in\)trss \(fg\) ((\((sh, ch), w, (s’, c’+ce)\))\in trcl \((trss \ fg)\) by (fast dest; trcl-unscons)

from trss-c-fml-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+ce))\in trss \(fg\) mon-e \(fg\) \(e\) \(\cap\) mon-c \(fg\) \(ce\) = \(\{\}\) by blast+

moreover from CHFMT HYP(1) SPLIT(2) have \(((sh,csp+ce),w,(s’,c’))\in trcl\((trss \ fg)\) mon-w \(fg\) \(w\) \(\cap\) mon-c \(fg\) \(ce\) = \(\{\}\) by blast+

ultimately show ?case by auto

qed

lemma trss-xchange-context-s:

assumes \(A: ((s,c),e,(s’,csp+ce))\in trss \(fg)\)
and \(M:\) mon-c \(fg\) \(cn\) \(\subseteq\) mon-c \(fg\) \(c\)
shows \(((s, cn), e, (s’, csp+cn))\in trss \(fg)\)

proof –

from trss-drop-context-s[of - \(\#\), simplified, OF \(A\)] have DC: \(((s, \{\#\}), e, s’, csp)\in trss \(fg)\) mon-e \(fg\) \(e\) \(\cap\) mon-c \(fg\) \(c\) = \(\{\}\) by simp-all

with \(M\) have mon-e \(fg\) \(e\) \(\cap\) 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+ce))\in trcl \((trss \ fg)\)
and \(M:\) mon-c \(fg\) \(cn\) \(\subseteq\) mon-c \(fg\) \(c\)
shows \(((s, cn), w, (s’, csp+cn))\in trcl \((trss \ fg)\)

proof –

from trss-drop-context[of - \(\#\), simplified, OF \(A\)] have DC: \(((s, \{\#\}), w, s’, csp)\in trcl \((trss \ fg)\) mon-w \(fg\) \(w\) \(\cap\) mon-c \(fg\) \(c\) = \(\{\}\) by simp-all

with \(M\) have mon-w \(fg\) \(w\) \(\cap\) 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’))\in trss \(fg)\)
and \(C: \) !!csp. \[c’:=csp+c\]; \(((s,\{\#\}),e,(s’,csp))\in trss \(fg)\] \(\implies\) \(P\)

shows \(P\)

using \(A\) proof (cases rule: trss-c-cases-s)

case no-spawn with trss-xchange-context-s[of \(s\ e\ s’\ \{\#\}\) \(fg\) \(\{\#\}\) \(A\) \(C\) show \(P\) by auto

next

case (spawn \(p\ u\ v)\) with trss-xchange-context-s[of \(s\ 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’))\in trcl \((trss \ fg)\)
and \(C: \) !!csp. \[c’:=csp+c\]; \(((s,\{\#\}),w,(s’,csp))\in trcl \((trss \ fg)\)] \(\implies\) \(P\)

shows \(P\)

50
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 efflowgraph)* **iso-ret-no-ret:** !!u c. [isolated-ret fg p;
\[
\text{proc-of } fg \ u = p; \\
\ u \neq \text{return } fg \ p; \\
((\{u\},c),w,([\text{return } fg \ p\,],c')) \in \text{trcl } (\text{trss } fg) \\
\] 
\[\implies \text{False} \]

**proof** (induct \(w\) rule: length-compl-induct)

**case** Nil thus \(\text{Nil}\) by auto

**next**

**case** (Cons \(e\) \(w\)) note \(\text{IHP} = \text{this}\)

then obtain \(sh\ ch\ \text{where } SPLIT1: ((\{u\},c),e,(sh,ch)) \in \text{trss } fg\) and \(SPLIT2: ((sh,ch),w,([\text{return } fg \ p\,],c')) \in \text{trcl } (\text{trss } fg)\) by (fast dest: trcl-uncons)

**show** \(\text{?case proof } (\text{cases } e)\)

**case** \(\text{LRet with SPLIT1 } \text{IHP}(3,4)\) show \(\text{False}\) by (auto elim!: trss.cases)

**next**

**case** \(\text{LBase with SPLIT1 } \text{IHP}(2,3)\) obtain \(v\) where \(A: sh = [v]\) proc-of \(fg\) \(v = p\ \wedge \text{return } fg \ p\) by (force elim!: trss.cases simp add: edges-part isolated-ret-def)

**with** \(\text{IHP } SPLIT2\) **show** \(\text{False}\) by auto

**next**

**case** \(\text{LSpawn } q\) with \(\text{SPLIT1 } \text{IHP}(2,3)\) obtain \(v\) where \(A: sh = [v]\) proc-of \(fg\) \(v = p\ \wedge \text{return } fg \ p\) by (force elim!: trss.cases simp add: edges-part isolated-ret-def)

**with** \(\text{IHP } SPLIT2\) **show** \(\text{False}\) by auto

**next**

**case** \(\text{LCall } q\) with \(\text{SPLIT1 } \text{IHP}(2,3)\) obtain \(u\) where \(A: sh = \text{entry } fg\) \(q\#[uh]\) proc-of \(fg\) \(uh = p\ \wedge \text{return } fg \ p\) by (force elim!: trss.cases simp add: edges-part isolated-ret-def)

**with** \(\text{SPLIT2}\) **have** \(B: ([\text{entry } fg \ q\#[uh],ch,w,([\text{return } fg \ p\,],c')]) \in \text{trcl } (\text{trss } fg)\)

**by** simp

from \(\text{trss-return-cases[OF } B]\) obtain \(w1\) \(w2\) \(ct\) where \(C: w = w1 \cdot w2\) length \(w2 \leq \text{length } w\) \(([\text{entry } fg \ q\#[ch],w1,([ct])] \in \text{trcl } (\text{trss } f)g\) \(([uh],ct),w2,([\text{return } fg \ p\,],c')) \in \text{trcl } (\text{trss } fg)\) by (auto)

from \(\text{IHP}(1)[\text{OF } C(2) \ \text{IHP}(2) \ A(2,3) \ C(4)]\) **show** \(\text{False}\).

**qed**

**qed**

— The first step of an initial procedure is a call

**lemma** (in eflowgraph) initial-starts-with-call:

\[
\begin{align*}
\text{I } ([\text{entry } fg \ p\,],c,e,(s',c')) \in \text{trss } fg; \text{ initialproc } fg \ p \]
\[
\implies \exists p'. \ e = \text{LCall } p' \wedge \text{isolated-ret } fg \ p' \\
\] 

**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** \(A: w \neq []\) initialproc \(fg\ p\)

\[
((\text{entry } fg \ p\,],c),w,([v],c')) \in \text{trcl } (\text{trss } fg) \\
\] 

**shows** \(\text{False}\)

**proof**

**from** \(A\) obtain \(sh\ ch\ e\ w'\) where \(\text{SPLIT}: ([\text{entry } fg \ p\,],c,e,(sh,ch)) \in \text{trss } fg\)

\((sh,ch),w',([v],c')) \in \text{trcl } (\text{trss } fg)\) by (cases \(w,\) simp, fast dest: trcl-uncons)

**from** initial-starts-with-call[\(\text{OF } SPLIT(1) \ A(2)\)] obtain \(p'\) where \(CE: e = \text{LCall } p'\) isolated-ret \(fg\) \(p'\) **by** blast

52
with SPLIT(1) obtain u′ where sh=entry fg p p′#[u′] by (auto elim! : trss.cases)
with SPLIT(2) have ((entry fg p′#[u′],ch),w′,([v],c′))є trcl (trss fg) by simp
then obtain wa ct where (((entry fg p′[,ch],wa,([],ct))є trcl (trss fg) by (crule-tac trss-return-cases, auto)
  then obtain wa′ p′′ where (((entry fg p′[,ch],wa′,([return fg p′′],ct))є trcl (trss fg) by (blast dest: trss-return-empty)
  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 ≠ []
  initialproc fg p
  (([(entry fg p],c),w,(r′,c′))є trcl (trss fg)
  length r′ ≤ 1
  shows False
proof (cases r′)
  case Nil
  have ([(entry fg p],c),w,([],ct))є 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′)є trcl (trss fg) by (blast)
  show ?thesis proof (cases w′)
    case Nil
    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′≠[] 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 per-

\(^2\)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.
formed 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 to derive interleaving semantics on multisets and configurations with an explicit local thread (loc/env-semantics, cf. Section 5.4).

**inductive-set**

\[ \text{ntrs :: } (\exists n, p, ba, m, \text{more}) \text{ flowgraph-rec-scheme } \Rightarrow \]

\[ ((n \text{ list } \times \text{ n conf}) \times (p, ba) \text{ label list } \times (n \text{ list } \times \text{ n conf})) \text{ set} \]

**for fg**

**where**

— A macrostep transforms one thread by first calling a procedure and then doing a same-level path

\[ \text{ntrs-step: } \exists((u, r, c), \text{LCall } p, (\text{entry } fg \ p, u)^{u'}, (v, c')) \in \text{trss } fg; \]

\[ (([\text{entry } fg \ p], c, v, c')) \in \text{trcl } (\text{trss } fg) \Rightarrow \]

\[ (u, r, c), \text{LCall } p, u^w, v^{u'}, c') \in \text{ntrs } fg \]

**abbreviation** \( \text{ntr where ntr } fg \equiv \text{gtr } (\text{ntrs } fg) \)

**abbreviation** \( \text{ntrp where ntrp } fg \equiv \text{gtrp } (\text{ntrs } fg) \)

**interpretation** \( \text{ntrs: env-no-step ntrs } fg \)

**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** \( \text{ntrs-is-trss-s: } (s, c, w, (s', c')) \in \text{ntrs } fg \Rightarrow (s, c, w, (s', c')) \in \text{trcl } (\text{trss } fg) \)

**proof** (erule ntrs cases, auto)

**fix** \( p \ r \ a \ u' \ v \ w \)

**assume** \( A: (u \neq r, c), \text{LCall } p, (\text{entry } fg \ p, u', r, c) \in \text{trss } fg ((\text{entry } fg \ p), c, w, [v], c') \in \text{trcl } (\text{trss } fg) \)

**from** \( \text{trss-stack-comp } \{ \text{OF } A(2), \text{ of } u' \# r \} \) \_[**have**] (entry \( fg \ p, u', r, c), w, v \# u' \# r, c') \in \text{trcl } (\text{trss } fg) \) by simp

**with** \( A(1) \) \_[**show**] (entry \( fg \ p, u', r, c') \in \text{trcl } (\text{trss } fg) \) 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 efloowgraph) ntr-sl-move-left: !!ce u r w r’ ce’.
\[\{\{\#[entry fg p]\#\},ww,\{\# u\# r\#\}+ce\}\in trcl (ntr fg)\];
\{\{(u),ce),(w,(r’,ce’))\}\in trcl (trss fg);
initialproc fg p;
length r’ \leq 1; w\neq[]\]
\[\implies \exists wu’. \{\{\#[entry fg p]\#\}, wu’,\{\# r’\# r\#\}+ce’\}\in trcl (ntr fg)\]
proof (induct uu 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 cc wr) note IHP=\this
— In the induction step, we extract the last macrostep
then obtain cch where SPLIT: \{\{\#[entry fg p]\#\},ww,ch\}\in trcl (ntr fg) (ch,ce,\{\# u\# r\#\}+ce)\by (fast dest: trcl-rev-uncons)
— The last macrostep first executes a call and then a same-level path
from SPLIT(2) obtain q wu s u h ceh u’ w t c et where
STEPFMT: ce=LCall q\#wu ch=+add-mset (uh\# rh) ceh add-mset (u\# r) ce 
= add-mset (v\# uh’\# rh) cet ((uh\# rh,ceh).LCall q.(entry fg q\# uh’\# rh,ceh))\in trss 
fg ((\{entry fg q,ceh\},wu,t,ce))\in trcl (trss fg)
by (auto elim: grtE ntr,r 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’t)
— If the sl/ret path was executed on the same thread as the last macrostep
case note CASE=\this hence C’: u=vt r=uh’\# rh ce=ct by auto
— we append it to the last macrostep.
with STEPFMT(5) IHP(3) have NEWPATH: \{\{entry fg q,ceh\},wu\# w,(r’,ce’)\}\in 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\},wu\# w,([v’],ce’)\}\in trcl (trss fg) by simp
— and thus remains a valid macrostep
from grtl-s[OF ntrs-step[OF STEPFMT(4), simplified, OF this]] have
\{add-mset (uh \# rh) ceh, LCall q \# wu\# w, add-mset (v’ \# uh’ \# rh) ce’\}
\in 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]\#\}, wu\# [LCall q\# wu\# w]\{\# r’\# r\#\} + ce’\}\in 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

56
— The macrostep now ends with a returning path, and thus gets a same-level path

\[
\text{have NEWSL: } \left[ ([uh], ceh), \text{LCall } q \# \text{ was } @ w, ([uh'], ce') \right] \in \text{trcl (trss } fg) \\
\text{proof —}
\]

\[
\text{from STEPFMT(4) have } \left( ([uh], ceh), \text{LCall } q, (\text{entry } fg \ q \# ([uh'], ceh)) \in \text{trss } fg \right) \text{ by (auto elim!: trss.cases intro: trss.intro)}
\]

\[
\text{also from trss-stack-comp (OF NEWSPATH) CC have } \left( (\text{entry } fg \ q \# ([uh'], ceh), \text{was} @ w, ([uh'], ce')) \in \text{trcl (trss } fg) \right) \text{ by auto}
\]

\[
\text{finally show } \text{thesis}. \\
\text{qed}
\]

— Hence we can apply the induction hypothesis and get the proposition

\[
\text{from IHP(1)(OF - NEWSL) SPLIT STEPFMT(2) IHP(4) CC C'(2) show } \text{thesis by auto}
\]

\text{qed}

next — If the sl/ret path was executed on a different thread than the last macrostep

\[
\text{case (env } cc) \text{ note } \text{CASE = this}
\]

— 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

\[
\text{from STEPFMT(5) obtain } \text{cspt where CETFMT: cet=cspt+ceh !s. s } \in \# \\
\text{cspt } \Rightarrow \exists p, s=[\text{entry } fg \ p] \wedge \text{initialproc } fg \ p
\]

\[
\text{by (unfold initialproc-def) (erule trss-c-cases, blast)}
\]

— The spawned threads do not hold any monitors yet

\[
\text{hence CSPT-NO-MON: } \text{mon-c } fg \ \text{cspt } = \{ \} \text{ by (simp add: c-of-initial-no-mon)}
\]

— 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

\[
\text{from CASE(1) CETFMT(1) have } u\#r } \in \# \text{ cspt+ceh by auto}
\]

\[
\text{thus } \text{thesis proof (cases rule: mset-un-cases[cases sel!])}
\]

— The sl/ret path cannot have been executed on a freshly spawned thread due to the restrictions we made on the flowgraph

\[
\text{case left — Thread was spawned with CETFMT obtain } q \text{ where } u=\text{entry} \\
\text{fg q r=[]} \text{ initialproc } fg \ q \ \text{by auto}
\]

\[
\text{with IHP(3,5,6) no-retsl-from-initial have } \text{False by blast}
\]

\[
\text{thus } \text{thesis ..}
\]

next — Hence let’s assume the sl/ret path is executed on a thread that was already there before the last macrostep

\[
\text{case right note CC = this}
\]

— We can write the configuration before the last macrostep in a way that one sees the thread that executed the sl/ret path

\[
\text{hence CEHFMT: ceh=\{# u\#r \}\+(ceh-\{# u\#r \}) by auto}
\]

\[
\text{have CHFMT: ch = \{# u\#r \\} + (\{# u\#r \}\+(ceh-\{# u\#r \}))}
\]

\text{proof —}

\[
\text{from CEHFMT STEPFMT(2) have } ch = \{# u\#r \\} + (\{# u\#r \}\+(ceh-\{# u\#r \})) \text{ by simp}
\]

\[
\text{thus } \text{thesis by (auto simp add: union-ac)}
\]

\text{qed}

— There are not more monitors than after the last macrostep
have MON-CE: mon-c fg (%# uh # rh #)+ (ceh-%# u # r #)) ⊆ mon-c fg
ce proof –
  have mon-n fg uh ⊆ mon-n fg u # using STEPFMT(4) by (auto elim:
trss.cases dest: mon-n-same-proc edges-part)
  moreover have mon-c fg (ceh-%# u # r #)) ⊆ mon-c fg ce 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 cspt # where CE’FMT: ce’=cspt’+ce mon-c fg cspt’ = {}
b y (–) (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’, cspt’+ (%# uh # rh #)+ (ceh
-%# u # r #)) ∈ 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 uu’ where
NNPATH: (%# [entry fg p #], uu’, %# r #)+ (ceh-%# u # r #))
(ceh-%# u # r #)) ∈ 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 #)+ (cspt’+ (%# uh # rh #)+ (ceh-%# u # r #)), ee,
(ceh-%# u # r #)) ∈ ntr fg
  proof –
  — This is because the sl/ret path has not allocated any monitors
  have MON-CEH: mon-c fg (%# r #)+ (cspt’+ (ceh-%# u # r #)))
⊆ mon-c fg ceh proof –
  from IHP(3,5) trss-bot-proc-const[of # u w # - ce] mon-n-same-proc
have mon-s fg r’ ⊆ mon-n fg u by (cases r’ simp, force)
  moreover from CEHFMT have mon-c fg ceh = mon-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 cspt=%#], simplified,
OF STEPFMT(4)]
  moreover from trss-xchange-context[OF - MON-CEH, of [entry fg q] wus
[v, cspt] STEPFMT(5) CETFMT(1) have
((%# [entry fg q] q, %# r #)+ (ceh-%# u # r #)), wus, [v],
(ceh-%# u # r #)) ∈ trcl (trss fg) by blast
  moreover note STEPFMT(1)
ultimately have \(((u\# rh.\{\# r' @ r\#\}) + (cs p' + \mathit{ceh} - \{# u \# r\#\}))\), \(ce . (w\# u\# rh. cs p t + (\{\# r' @ r\#\}) + (cs p' + \mathit{ceh} - \{# u \# r\#\}))\) ∈ \text{ntrs fg} by (auto intro: ntrs.intros).

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

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

from trcl-rev-cons[OF \mathit{NNPATH} this] have \(\{\# \mathit{entry fg} p \#\}, \mathit{u} w'@ [\mathit{ce}], \{\#\mathit{vt} \# u\# rh\#\} + (cs p t + (\{\# r' @ r\#\}) + (cs p' + \mathit{ceh} - \{# u \# r\#\}))\) ∈ trcl (\text{ntr fg}) .

— And show that we got the right configuration

moreover from \text{CC CETFMT CASE}(3)[symmetric] CASE(2) CE'FMT(1) have \(\{\#\mathit{vt} \# u\# rh\#\} + (cs p t + (\{\# r' @ r\#\}) + (cs p' + \mathit{ceh} - \{# u \# r\#\}))\) = \(\{# r' @ r\#\} + \mathit{ce}\) by (simp add: union-ac)

ultimately show ?thesis by auto

— 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 \(\text{?case by (auto intro: exI[of - []])} \) 

next

case (snoc \text{cstart} w c e c') note \text{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 \(w'\# w c e c'\) where \(\text{IHP'}: (\{\# \mathit{entry fg} p \#\}) \wedge \text{trcl (ntr fg)} (c, e, c') ∈ \text{tr} fg\) by blast

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

from gtr-find-thread[OF \text{IHP'}(2)] obtain \(s c e s' c e'\) where \(\mathit{TSTEP}: c = \text{add-mset} s c e c' = \text{add-mset} s' c e' ((s, e), c, (s', c')) ∈ \text{trss fg}\) 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

59
As it is the last step, the procedure call will not return and thus is a valid macrostep.

That can be appended to the initial normalized path

...
$(s,c),ww,(s',c')\in tcl (ntrs fg)$;
$$!!\text{csp\,}\begin{cases} c'=c\text{sp}+c; \text{!!s.}\ s\in\#c\implies\exists\ p\ u\ v.\ s=[\text{entry\ fg\ p}]\land u,\text{Spawn\ p,v}\in\text{edges\ fg}\land\text{initialproc\ fg\ p}\end{cases}\rightarrow P$$
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 $A: (\langle (s,c),e,(s',c')\rangle)\in\text{ntrp\ fg}$
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 $A: (\langle (s,c),e,(s',c')\rangle)\in\text{trcl (ntrp\ fg)}$
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**
$\text{mon-ww\ fg\ ww} ==\ \text{foldl\ (}\cup\text{\{}}\ (\text{map\ (}\text{mon-w\ fg}\text{)\ ww})$  

**definition**
$\text{mon-loc\ fg\ ww} ==\ \text{mon-ww\ fg\ (map\ \text{le-rem-s\ (loc\ ww})$  

**definition**
$\text{mon-env\ fg\ ww} ==\ \text{mon-ww\ fg\ (map\ \text{le-rem-s\ (env\ ww})$  

**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-ww fg ee U mon-ww fg ww
by (unfold mon-ww-def, auto simp add: foldl-un-empty-eq[of mon-ww fg ee])

**lemma** mon-ww-unconc:
mon-ww fg (ww1@ww2) = mon-ww fg ww1 U 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 ⇒ {} | ENV a ⇒ 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 ⇒ {} | ENV a ⇒ mon-w fg a) ∪ 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 ∪ 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 ⇒ {} | LOC a ⇒ 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 ⇒ {} | LOC a ⇒ mon-w fg a) ∪ 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 ∪ 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 ≤ w' ⇒ mon-ww fg w ⊆ mon-ww fg w'
  by (induct rule: less-eq-list-induct) auto

lemma mon-ww-cil:
  w ∈ w1 ⊗ α w2 ⇒ mon-ww fg w = mon-ww fg w1 ∪ mon-ww fg w2
  by (induct rule: cil-set-induct-fix) auto
lemma mon-loc-cil:
  w ∈ w1 ⊗ α w2 ⇒ mon-loc fg w = mon-loc fg w1 ∪ mon-loc fg w2
  by (induct rule: cil-set-induct-fix) auto
lemma mon-env-cil:
  w ∈ w1 ⊗ α w2 ⇒ mon-env fg w = mon-env fg w1 ∪ mon-env fg w2

62
by (induct rule: cil-set-induct-fixα) auto

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

lemma mon-env-ileq: w ≪ w' ⟹ mon-env fg w ⊆ mon-env fg w'
by (induct rule: less-eq-list-induct) auto

lemma mon-loc-ileq: w ≪ w' ⟹ mon-loc fg w ⊆ 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'))∈ntrs fg
    ⟹ mon-s fg s ⊆ mon-s fg s' ∧ 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')∈ntr fg ⟹ mon-c fg c ⊆ 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'))∈ntrp fg
    ⟹ mon-s fg s ⊆ mon-s fg s' ∧ mon-c fg c ⊆ 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-anconc)
apply (erule ntrs-c-cases-s)
apply (auto simp: mon-c-unconc)
done

lemma (in flowgraph) ntrp-mon-increasing: ((s,c),e,(s',c'))∈trcl (ntrp fg)
\[ \Rightarrow \text{mon-s } f g \subseteq \text{mon-s } f g \ s' \land \text{mon-c } f g \ c \subseteq \text{mon-c } f g \ c' \]

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]
lemmas (in flowgraph) ntrs-mon-w-no-ctx:

assumes A: \((s',c')\) 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:
\((s,c),\text{ENV}\ e,(s',c')\) in ntrp fg \Rightarrow \text{mon-env } fg \ w \cap \text{mon-s } fg \ s = \{}
by (auto elim!: grtrp.cases dest!: ntrs-mon-e-no-ctx simp add: mon-c-unconc)

lemma (in flowgraph) ntrp-mon-loc-e-no-ctx:
\((s,c),\text{LOC}\ e,(s',c')\) in ntrp fg \Rightarrow \text{mon-loc } fg \ w \cap \text{mon-c } fg \ c = \{}
by (auto elim!: grtrp.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) \Rightarrow \text{mon-env } fg \ w \cap \text{mon-s } fg \ s = \{}

lemma (in flowgraph) ntrp-mon-loc-w-no-ctx:
\((s,c),w,(s',c')\) in trcl (ntrp fg) \Rightarrow \text{mon-loc } fg \ w \cap \text{mon-c } fg \ c = \{}

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: \((s,c),\text{ee},(s',c')\) in ntrs fg
and B: mon-w fg ee \cap mon-c fg cn = \{}
shows \exists csp. c' = csp + c \land mon-c fg csp = \{}
\land ((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=\Call{L}{p\#w}{s'\#v\#u'\#r}\ ((u\#r,c),\Call{L}{p}{(\text{entry } fg\ p\#u\#r\ c})\in \text{trss } fg\ ([([\text{entry } fg\ p]\ c),w,([v],c'))\in \text{trcl } (\text{trss } fg)\} by (blast elim!: ntrs.cases[simplified])

with ntrs-modify-context-s[OF S(4)] B have \((u\#r,cn),\Call{L}{p}{(\text{entry } fg\ p\#u\#r\ cn)}\in \text{trss } fg\) by auto

moreover from S ntrs-modify-context[OF S(5)] B obtain csp where c' = csp + c mon-c fg csp = \{}
\((([\text{entry } fg\ p]\ cn),w,([v],csp+cn))\in \text{trcl } (\text{trss } fg)\) by auto
ultimately show \(?thesis\ using S\ by\ (auto\ intro!: ntrs-step)

64
qed

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

\(((s,c),w,(s',c')) \in \text{trcl}(\text{ntrs fg})\)
\implies \forall cn. \text{mon-ww} fg w \cap \text{mon-c} fg cn = \{\}
\rightarrow (\exists \text{esp}, c' = csp + c \land \text{mon-c} fg \text{esp} = \{\} \land
((s,cn),w,(s',csp+cn)) \in \text{trcl}(\text{ntrs fg}))

proof (induct rule: trcl-pair-induct)
next
case empty thus \text{case by simp}

proof (intro allI impI)
  fix cn
  assume MON: \text{mon-ww} fg (e \not\# w) \cap \text{mon-c} fg cn = \{\}
  from ntrs-modify-context-s[OF IHP(1)] MON obtain csp where S1: \text{ch} = csp + c \land \text{mon-c} fg csp = \{\}
  with MON have \text{mon-ww} fg w \cap \text{mon-c} fg (csp+cn) = \{\}
    by (auto simp add: mon-c-unconc)
  with IHP(3)[rule-format] obtain csp where S2: c' = csp + ch \land \text{mon-c} fg csp = \{\}
    ((s, cn), csp+cn) \in ntrs fg by blast
  from S1 S2 have c' = (csp+ch) + c \land \text{mon-c} fg (csp+ch) = \{\}
    ((s, cn), c\not# w, (s', (csp+ch)+cn)) \in \text{trcl}(ntrs fg)
    by (auto simp add: union-assoc mon-c-unconc)
  \text{thus } \exists \text{esp}, c' = csp + c \land \text{mon-c} fg \text{esp} = \{\} \land ((s, cn), e \not\# w, s', csp + cn)
    \in trcl (ntrs fg)
    by blast
qed

lemma ntrs-xchange-context-s:
  assumes A: ((s,c),ee,(s',esp+c)) \in ntrs fg
  and B: mon-c fg cn \subseteq mon-c fg c
  shows ((s,cn),ee,(s',esp+cn)) \in 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)) \in trcl (trss fg)
  ((entry fg p), c, w, [v], csph) \in trcl (trss fg)
  proof
    from ntrs.cases[OF A, simplified] obtain ce ce' p r u u' v w where s = u \not\# r c = ce ee = LCall p \# w s' = v \not\# u' \# r csp + ce = ce' ((u \# r, c), LCall p, entry fg p \# u' \# r, c) \in trss fg
    \text{hence } 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], csph) \in trcl (trss fg)
    by auto
    then show \text{thesis} ..
  qed
  from ntrs-step[simplified, OF trss-xchange-context-s\where csp=[\#], simplified, OF S(4) \text{ B}] trss-xchange-context[OF S(5) \text{ B}] S show \text{thesis by simp}
qed

lemma ntrs-replace-context-s:
  assumes A: ((s,c+cr),ee,(s',c'+cr)) \in ntrs fg

65
and $B$: mon-c fg crn $\subseteq$ mon-c fg cr

shows $((s, c + crn), ee, (s', c' + crn)) \in \text{ntrs fg}$

proof –

from ntrs-cases-s[OF A] obtain csp where G: $e' + cr = csp + (e + cr)$ . hence

$F$: $c' = csp + c$ by (auto simp add: union-assoc[symmetric])

from B have MON: mon-c fg $(e + crn) \subseteq$ mon-c fg $(e + cr)$ by (auto simp add: mon-c-unconc)

from ntrs-xchange-context-s[OF - MON] A G have $((s, c + crn), ee, (s', csp + (e + crn))) \in \text{ntrs fg}$ by auto

with $F$ show ?thesis by (simp add: union-assoc)

qed

lemma (in flowgraph) ntrs-xchange-context: $!!s \in c' \in cn$ [\[
((s, c), uu, (s', c')) \in \text{trcl (ntrs fg)};
\]

mon-c fg cn $\subseteq$ mon-c fg c

\[
\implies \exists \text{csp}.
\]

c' = csp + c $\land$ $(s, cn), uu, (s', csp + cn)) \in \text{trcl (ntrs fg)}$

proof (induct uu)

case Nil note CASE=this

thus $?case$ by (auto intro: exI [of - \{

next

case (Cons ee uu) note IH$_P$=this

then obtain sh ch where SPLIT: $((s, c), ee, (sh, ch)) \in \text{ntrs fg} ((sh, ch), uu, (s', c')) \in \text{trcl (ntrs fg)}$ by (fast dest: trcl-uncons)

from ntrs-cases-s[OF SPLIT(1)] obtain csph where CHFMT: ch = csp + ch

$!!s, s \in \# csph \implies \exists p u v. s$ = [entry fg p] $\land$ $\exists u, Spawn p, v \in$ edges fg $\land$ initialproc $\implies$ fg $p$ by blast

with ntrs-xchange-context-s SPLIT(1) IHP(3) have $((s, cn), ee, (sh, csph + cn)) \in \text{ntrs fg}$ 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 I: 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: $(sh, csph + cn), uu, (s', csp + (csph + cn)) \in \text{trcl (ntrs fg)}$ by blast

note SND

finally have $((s, cn), ee \# uu, s', (csp + csph) + cn) \in \text{trcl (ntrs fg)}$ 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: $((s, c + cr), uu, (s', c' + cr)) \in \text{trcl (ntrs fg)}$

and $B$: mon-c fg crn $\subseteq$ mon-c fg cr

shows $((s, c + crn), uu, (s', c' + crn)) \in \text{trcl (ntrs fg)}$
proof

from ntrs-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-c fg (c+crn) ⊆ mon-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)))∈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')∈ntr fg
  and B: mon-w fg e ∩ mon-c fg cn = {}
  shows (c+cn,e,c'+cn)∈ntr fg

proof

from gtrE[OF A] obtain s ce s' ce' where NTRS: c = add-mset s ce c' = add-mset s' ce' (s, ce), e, s', ce')) ∈ ntrs fg .

from ntrs-mon-e-no-ctx[OF NTRS(3)] B have M: mon-w fg e ∩ (mon-c fg (c+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))∈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')∈trcl (ntr fg); mon-ww fg w ∩ mon-c fg cn = {}]
  ⊢ (c+cn,w,c'+cn)∈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'))∈ntrs fg
  and B: mon-w fg e ∩ mon-c fg cn = {}
  shows (s,c+cn),e,(s',c'+cn)∈ntrs fg

lemma (in flowgraph) ntrp-add-context-s:
  [( (s,c),e,(s',c'))∈ntrp fg; mon-w fg (le-rem-s e) ∩ mon-c fg cn = {} ]
  ⊢ (s,c+cn),e,(s',c'+cn)∈ntrp fg
  apply (erule gtrp.cases)
  by (auto dest: ntrp-add-context-s intro!: gtrp.intros)

lemma (in flowgraph) ntrp-add-context:
  [( (s,c),w,(s',c'))∈trcl (ntrp fg); mon-ww fg (map le-rem-s w) ∩ mon-c fg cn = {} ]
  ⊢ (s,c+cn),w,(s',c'+cn)∈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: \(((s,c), ee, (s', c')) \in ntrs fg\)
  shows \(((s@r, c), ee, (s@r, c')) \in ntrs fg\)
  using A
  by (auto dest: trss-stack-comp trss-stack-comp-s elim!: ntrs_cases intro!: ntrs_step[simplified])

lemma ntrs-stack-comp: \(((s,c), ww,(s', c')) \in trcl (ntrs fg)\)
  \implies \(((s@r, c), ww,(s@r, c')) \in trcl (ntrs fg)\)
  by (induct rule: trcl-pair-induct) (auto intro!: trcl_cons[OF ntrs-stack-comp-s])

lemma (in flowgraph) ntrs-stack-comp-s:
  assumes A: \(((s,c), ee, (s', c')) \in ntrs fg\)
  and B: \(mon-s fg r \cap mon-env fg \{ee\} = \{\}\)
  shows \(((s@r, c), ee, (s@r, c')) \in ntrs fg\)
  using A
  proof (cases rule: gtrp_cases)
    case gtrp-loc then obtain e where CASE: \(ee = LOC e ((s,c), ee, (s', c')) \in ntrs fg\)
    by auto
    hence \(((s@r, c), ee, (s@r, c')) \in ntrs fg\) 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: \(s' = s c = \{\#sm\} + ce\)
      \(c' = \{\#sm'\} + ce' \subseteq ENV e ((sm, \{\#s\} + ce), (sm', \{\#s\} + ce')) \in ntrs fg\)
    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: \(\{\#s\} + ce = csp + (\{\#s\} + ce)\) \(\text{mon-c fg csp} = \{\}\) \(\text{mon-env fg csp} = \{\}\)
      \((sm, \{\#s\} @ r\} + ce), e, sm', csp + (\{\#s @ r\} + ce)\) \(\in ntrs fg\) 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 \((sm, \{\#s @ r\} + ce), e, sm', \{\#s @ r\} + ce') \in ntrs fg\)
    by (simp add: union-ac)
    with CASE(1, 2, 3, 4) show ?thesis by (auto intro: gtrp.gtrp-env)
  qed

lemma (in flowgraph) ntrs-stack-comp:
  \[\(((s,c), ww,(s', c')) \in trcl (ntrs fg); mon-s fg r \cap mon-env fg ww = \{\}\)\]
  \implies \(((s@r, c), ww,(s@r, c')) \in trcl (ntrs fg)\)
  by (induct rule: trcl-pair-induct) (auto intro!: trcl_cons[OF ntrs-stack-comp-s])

lemma ntrs-stack-top-decomp-s:
  assumes A: \(((u\#@r, c), ee, (s', c')) \in ntrs fg\)
  and EX: \(!v u' p. \[\]
    s' = v\# @ u'\# @ r;
    \(((u), ee, ([v, u'], c')) \in ntrs fg;\)
    \((u, Call p, u') \in edges fg\)
  \] \implies P

68
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 ((w \# r, c), LCall p, (entry fg p \# u' \# r, c)) \in \text{trss fg} (((\text{entry fg} p), c), w, ([v], c')) \in \text{trcl (trss fg}) by (simp)

from trss_stack-decomp-s[where $s=[u]$, simplified, OF CASE(3)] have SDC: 

$((\text{entry} p, f, u'), c) \in \text{trss fg}$ by auto

with CASE(1,4) have $((\text{entry} p, f, u'), c) \in \text{ntrs fg}$ by (auto intro: ntrs.ntrs_step)

moreover from SDC have $(u, \text{Call p}, u') \in \text{edges fg}$ by (auto elim: trss_cases)

ultimately show ?thesis using CASE(2) by (blast intro: EX)

qed

lemma ntrs_stack_decomp_s:

assumes $A: ((u \# s \# r, c), ee, (s', c')) \in \text{ntrs fg}$

and $EX: !v u' p$. [ $s' = v \# u' \# s \# r; ((u \# s, c), ee, (v \# u', c')) \in \text{ntrs fg}; (u, \text{Call p}, u') \in \text{edges fg}$]

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:

$!! u s r c P$. [ $!! v r r. [s' = v \# r \# r \# r; ((u \# s, c), u, (v \# r, c')) \in \text{trcl (ntrs fg})] \Rightarrow P$

$!! v r r. [s' = v \# r \# r \# r; ((u \# s, c), u, (v \# r, c')) \in \text{trcl (ntrs fg})] \Rightarrow P$

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 $F: sh = vh \# uh \# s \# r ((u \# s, c), e, vh \# uh \# s, c) \in \text{ntrs fg}$ $(u, \text{Call p}, uh) \in \text{edges fg}$ by blast

from $F(1)$ split(2) Cons.hyps[of $vh uh s r ch$] obtain $v' rr$ where $S: s' = v' \# r \# r \# r \# r; ((vh \# uh \# s, c), w, (v' \# r, c')) \in \text{trcl (ntrs fg})$ by auto

from trcl_cons[of $F(2)$ $S(2)$ $S(1)$ Cons.prems(2)] show ?thesis by blast

qed

qed

lemma ntrip_stack_decomp_s:

assumes $A: ((u \# s \# r, c), ee, (s', c')) \in \text{ntrp fg}$

and $EX: !v r r. [s' = v \# r \# r \# r; ((u \# s, c), ee, (v \# r, c')) \in \text{ntrp fg}] \Rightarrow P$

shows $P$

using $A$

proof (cases rule: gtrp_cases)

69
For a pair of entered and passed monitors, as required by the
We first need to define an abstraction function that maps a macrostep on a
8.5.1 Abstraction function for normalized paths
In this section, we describe the relation of the consistent interleaving oper-
8.5 Relation to monitor consistent interleaving
with $S$ show $P$ by (rule-tac EX) (auto intro: gtrp.gtrp-env)
qed

lemma ntrp-stack-decomp: \!\!u s r c P. \[
\begin{array}{c}
((u\#s@r,c),ww,(s',c'))\in\text{trcl (ntrp fg)}; \\
\text{!!v rr. } [s'=v\#r@r; ((u\#s,c),ww,(v\#r,r,c'))\in\text{trcl (ntrp fg)}] \implies P
\end{array}
\]
proof (induct ww)
next

\begin{itemize}
\item \textbf{Nil} thus \texttt{?case} by fastforce
\end{itemize}

\end{proof}

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.

\textbf{definition}
\begin{itemize}
\item \texttt{an fg e } \texttt{== if e =[]} then \{\},\{\} else \texttt{(mon-e fg (hd e), mon-w fg (tl e))}
\end{itemize}

\textbf{lemma} \texttt{an-simps[simp]}:
\begin{itemize}
\item \texttt{an fg [] } \texttt{= (},\{\},\{\})
\item \texttt{an fg (e\#w)} \texttt{= (mon-e fg e, mon-w fg w)}
\end{itemize}
\texttt{by (unfold an-def, auto)}
We also need an abstraction function for normalized loc/env-paths

\[ \alpha_{nl} \text{fg} e \triangleq \alpha \text{fg} (\text{le-rem-s} e) \]

**lemma** \( \alpha_{nl-def} \): \( \alpha_{nl} \text{fg} \text{fg} \triangleq \alpha \text{fg} \circ \text{le-rem-s} \)

by (rule eq-reflection[OF ext]) (auto simp add: \( \alpha_{nl-def} \))

These are some ad-hoc simplifications, with the aim at converting \( \alpha_{nl} \) back to \( \alpha_{n} \)

**lemma** \( \alpha_{nl-simps} \):

\[ \alpha_{nl} \text{fg} (\text{ENV x}) = \alpha_{n} \text{fg} x \]

\[ \alpha_{nl} \text{fg} (\text{LOC x}) = \alpha_{n} \text{fg} x \]

by (unfold \( \alpha_{nl-def} \), auto)

**lemma** \( \alpha_{nl-simps1} \):

\[ (\alpha_{nl} \text{fg}) \circ \text{ENV} = \alpha_{n} \text{fg} \]

\[ (\alpha_{nl} \text{fg}) \circ \text{LOC} = \alpha_{n} \text{fg} \]

by (unfold \( \alpha_{nl-def}' \text{comp-def} \)) (simp-all)

**lemma** \( \alpha_{n-col} \): (\( \alpha_{n} \text{fg} \circ \text{le-rem-s} = \alpha_{nl} \text{fg} \)

unfolding \( \alpha_{nl-def}' \text{[symmetric]} \).

**lemma** \( \alpha_{n-fst-snd} \):

\[ \text{fst} (\alpha_{n} \text{fg} w) \cap \text{snd} (\alpha_{n} \text{fg} w) = \text{mon-ww fg w} \]

by (induct w) auto

We now derive specialized introduction lemmas for \( \otimes_{\alpha_{n} \text{fg}} \)

**lemma** \( \text{cil-\alpha_{n}-cons-helper} \): \( \text{mon-pl} (\text{map} (\alpha_{n} \text{fg}) w) = \text{mon-ww fg wb} \)

apply (unfold \( \text{mon-pl-def} \))

apply (induct wb)

apply simp-all

apply (unfold \( \text{mon-ww-def} \))

apply (subst foldl-un-empty-eq)

apply (case_tac a)

apply simp-all

done

**lemma** \( \text{cil-\alpha_{n}-cons-helper} \):

\[ \text{mon-pl} (\text{map} (\alpha_{n} \text{fg}) w) = \text{mon-ww fg (map le-rem-s wb)} \]

by (simp add: \( \alpha_{n-col} \text{cil-\alpha_{n}-cons-helper[\text{symmetric]}} \))

**lemma** \( \text{cil-\alpha_{n}-cons1} \):

\[ \text{map le-rem-s} w \cap \text{mon-ww fg wb} = \{\} \]

\[ \Rightarrow \text{e#w} \in \text{e#wa} \otimes_{\alpha_{n} \text{fg}} \text{wb} \]

apply (rule \( \text{cil-cons1} \))

apply assumption

apply (subst \( \text{cil-\alpha_{n}-cons-helper} \))

apply assumption

done

71
lemma cil-αn-cons2: \[ \{ w \in w_{\alpha} \otimes \alpha n \text{fg} e \} \cap \text{mon-ww fg} \text{wa} = \{ \} \]

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-s fg \(s' = mon-s fg s \cup \text{fst (an fg e)}\)

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

lemma (in flowgraph) ntr-mon-s:
\((c,e,c')\in\text{ntr fg} \Rightarrow \text{mon-c fg } c' = mon-c fg c \cup \text{fst (an fg e)}\)
by (erule gtrE) (auto simp add: mon-c-anconc 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-c fg (add-mset s' c') = mon-c fg (add-mset s c) \(\cup \text{fst (anl fg e)}\)
using ntr-mon-s[OF gtrp2gtr-s[OF A]] by (unfold anl-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)\in)trcl (ntr fg); valid fg (ca+ cb)] \implies
\exists ca' cb' wa wb.
c'=(ca+ cb)' \land
w\in(wa\otimes_{an} fb) \land
\{mon-c fg ca \land (mon-c fg cb \cup mon-ww fg wb) = \emptyset \land
mon-c fg cb \land (mon-c fg ca \cup mon-ww fg wa) = \emptyset \land
\} \land
\} \land (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,\in)\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 ntr fg .
   — And separate the threads that where spawned during the first step from the ones
   that where already there
   then obtain esp where CEHFMT: ceh= esp+ ce mon-c fg esp= \emptyset by (auto elim!: ntr-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 ntr-c-mon-c-no-cs\OR{\OF \One OF NTRS(\emptyset)} have MONED: mon-w fg e \cap mon-c fg
   cc = \emptyset by (auto simp add: mon-c-unconc)
   — Needed later: The intermediate configuration is valid
   from ntr-valid-preserve-s\OR{\OR{\OF SPLIT(1) IHP(\emptyset)}} 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 \ONE OF NTRS have CHFMT: ch= (\emptyset +\#\emptyset \#)+ csp+ (ca- \emptyset \emptyset \#) +cb
   by (simp add: union-ac)
     — and by the induction hypothesis, we split the path from the intermediate
   configuration
   with \One OF SPLIT(2) CHVALID obtain ca' cb' wa wb where IHAPP:
   c'=(ca+ cb)' \land
w\in(wa\otimes_{an} fb) \land
\} \land (mon-c fg cb \cup mon-ww fg wb) = \emptyset
mon-c fg cb ∩ ( mon-c fg ((#sh#)+csp+(ca−{#s#})) ∪ mon-ww fg wa)=\{}

((#sh#)+csp+(ca−{#s#}),wa,ca')∈trcl ( ntr fg)
(cb,wb,cb')∈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#})))
f\ g by blast
from gtrl-s[OF this] CASE(1) show ?thesis by (auto simp add: union-assoc)
qed
with IHAPP(5) have (ca,e,#w,ca')∈trcl ( ntr fg) by simp
moreover
— and that we can prepend the first step to the interleaving
have e#w ∈ e#we ⊆ ntrs fg wb
proof
— from ntrs-called-mon[OF NTRS(3)] have fst (an fg e) ⊆ mon-s fg sh .
with IHAPP(3) have fst (an fg e) ∩ mon-ww fg wb = \{} by (auto simp add: mon-c-unconc)
from cil-an-const[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-wg 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 IH(1) SPLIT(2) CHVALID obtain ca' cb' wa wb where IHAPP:
c'=ca'+cb' e\∈wa⊗an 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)=\{}

74
\[(ca, wa, ca') \in \text{trcl } \left( ntr \ fg \ (\{\#s\#\} + csp + (cb - \{\#s\#\}), wb, cb') \right) \in \text{trcl } \left( ntr \ fg \right)\]

by blast

moreover

have FIRSTSTEP: \( (cb, e, \{\#s\#\} + csp + (cb - \{\#s\#\})) \in ntr \ fg \) proof

from CASE(2) have mon-c fg \((cb - \{\#s\#\}) \subseteq mon-c fg ce \) by (auto simp add: mon-c-unconc)

with \( \text{ntr-s-xchange-context-s NTRS}(3) \) CEHFMT CASE(2) have \((s, cb - \{\#s\#\}), e, (sh, csp + (cb - \{\#s\#\})))\)

proof by blast

from gtr1-s[OF this] CASE(1) show \(?thesis\) by (auto simp add: union-associ)

qed

with IHAPP(4) have \( PA: \ (cb, e \# wb, cb') \in \text{trcl } \left( ntr \ fg \right) \) by simp

moreover

have \( e \# w \in wa \cap \alpha n \ fg \ e \# wb\)

proof

from ntr-called-mon[OF NTRS(3)] have \( \text{fst } (\alpha n \ fg \ e) \subseteq mon-s fg sh\). 

with IHAPP(4) have \( \text{fst } (\alpha n \ fg \ e) \cap mon-ww fg \ wa = \{\}\) by (auto simp add: mon-c-unconc)

from cil-an-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 \( \text{ntr-mon-increasing-s[OF FIRSTSTEP] IHAPP}(3) \) 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 \text{flowgraph.ntr-split} for the semantics with a distinguished local thread. The proof follows exactly the same ideas, but is more complex.

\textbf{lemma (in flowgraph) ntrp-split:}

\( !s \ c1 c2 s' c'. \)

\([((s,c1+c2),w,(s',c')) \in \text{trcl } \left( ntrp \ fg \right); \ \text{valid } fg \ (\{\#s\#\} + c1 + c2)]\)

\( \Rightarrow \exists w1 w2 c1! c2'. \)

\( w \in w1 \cap \alpha n \ fg \ (\text{map ENV } w2) \wedge \)

\( c' = c1' + c2' \wedge \)

\( ((s,c1),w1,(s',c1')) \in \text{trcl } \left( ntrp \ fg \right) \wedge \)

\( (c2,w2,c2') \in \text{trcl } \left( ntrp \ fg \right) \wedge \)

\( \text{mon-ww } fg \ (\text{map le-rem-s } w1) \cap \text{mon-c } fg \ c2 = \{\}\wedge \)

\( \text{mon-ww } fg \ w2 \cap \text{mon-c } fg \ (\{\#s\#\} + c1) = \{\}\)

\textbf{proof (induct w)}

\textbf{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))∈ntrp fg ((sh,ch),w,(s',c'))∈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))∈ntrp fg by auto 
from ntrp-cases-s[OF CASE(2)] obtain csp where CHFMT: ch=(csp+c1)+c2 \s. s ∈# csp \implies \exists p u v. s = [entry fg p] \land (u, Spawn p, v) ∈ edges fg \land initialproc fg p by (simp add: union-assoc, blast)
with c-of-initial-no-mon have CSPNOMON: mon-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 c' c' where IHAPP: w ∈ w1 ⊆ ntrp mon-ww fg (map ENV w2) c' = c1' + c2' ((sh,csp + c1), w1, s', c1') ∈ trcl (ntrp fg) 
  (c2, w2, c2') ∈ trcl (ntrp fg) mon-ww fg (map le-rem-s w1) ∩ mon-c fg c2 = {}
  mon-ww fg w2 ∩ mon-c fg (csp+c1) = {}
  have ee#w ∈ ee#w1 ⊆ ntrp mon-ww fg (map ENV w2) proof (rule cil-cons1) 
  from ntrp-mon-env-w-no-ctx[OF SPLIT(2), unfolded mon-env-def] have mon-ww fg (map le-rem-s (env w)) ∩ mon-s fg sh = {} .
  moreover have mon-ww fg w2 ⊆ mon-ww fg (map le-rem-s (env w)) proof 
   from cil-subset-il IHAPP(1) ileq-interleave have map ENV w2 ≤ w by blast 
   from le-list-filter[OF this] have env (map ENV w2) ≤ env w by (unfold env-def) blast 
   hence map ENV w2 ≤ env w by (unfold env-def) simp 
   from le-list-map[OF this, of le-rem-s] have w2 ≤ map le-rem-s (env w) by simp 
   thus thesis by (rule mon-ww-ileq)
qed 
ultimately have mon-ww fg w2 ∩ mon-s fg sh = {} by blast 
with ntrp-mon-s[OF CASE(2)] CASE(1) show fst (csp fg ee) ∩ mon-pl (map (csp fg) (map ENV w2)) = {} by (auto simp add: cil-ax-cons-helper)
qed (rule IHAPP(1))
moreover 
have ((s,c1), ee#w1, (s',c1'))∈trcl (ntrp fg) proof 
  from ntr-exchange-context-s[of s c1+c2 e sh csp fg c1] CASE(2) CHFMT(1) have ((s,c1), sh, csp + c1) ∈ ntrp fg by (auto simp add: mon-c-unconc union-ac)
  with CASE(1) have ((s,c1), ee, sh, csp + c1) ∈ ntrp fg by (auto intro: gtrp,gtrp-loc)
  also note IHAPP(3)
  finally show ?thesis .
qed 
moreover from CASE(1) ntrp-mon-e-no-ctx[OF CASE(2)] 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 CASE(2)] CHFMT(1) IHAPP(6)
have mon-uw 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
  case gtrp-env then obtain e ss ce ssh ceh where CASE: ee=ENV e c1+c2=add-mset ss ce sh=s ch=add-mset ssh ceh ((ss,add-mset s ce),e,(ssh,add-mset s ceh))∈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-unplsm-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 by (simp add: union-assoc add-mset-commute)

  proof –

  from ntrs-valid-preserve-s[OF grtl-s, OF CASE(5)] Cons.prems(2) CASE(2)
  have valid fg ({#ssh#} + ({#ss#} + 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 ∈ w1 ⊗anl fg map ENV w2 c' = c1' + c2'

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

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

  have ee ∈ w ∈ (ee#w1) ⊗anl 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 (anl fg ee)

  ⊆ mon-s fg ssh by auto

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

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

  ultimately show fst (anl fg ee) ∩ mon-pl (map (anl fg) (map ENV w2))

  = {} by auto

  qed (rule IHAPP(1))

  moreover

  have SS: ((s,c1),e,c,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-exchange-context-s[of ss {#s#}+(c1-{#ss#})+c2 e ssh csp fg ({#s#}+(c1-{#ss#}))] have
\[(ss, \text{add-mset} s (c1 - \{\#ss\}), e, ssh, \text{add-mset} s (csp + (c1 - \{\#ss\})))]
\in \text{ntr s}\ fg\ by\ \text{(auto simp add: mon-c-unconc union-ac)}
\text{from gtrp, gtrp-conv[OF this] left(1) symmetric} \ CASE(1)\ \text{show ?thesis by}
\text{(simp add: union-ac)}
\text{qed}
\text{from trcl.cons[OF this IHAPP(3)] have} \ ((s, c1), ee \# w1, s', c1') \in \text{trcl (ntrp \ fg)}.
\text{moreover}
\text{from ntrp-mon-e-no-ctx[OF CASE(5)] left CASE(1) IHAPP(5) have mon-ww}
\begin{aligned}
&fg\ (\text{map}\ le-rem-s\ (ee\#w1)) \cap\ mon-c\ fg\ c2 = \{\} \ \text{by (auto simp add: mon-c-unconc)}
\end{aligned}
\text{moreover}
\text{from ntrp-mon-increasing-s[OF SS] IHAPP(6) have mon-ww fg w2 \cap mon-c}
\begin{aligned}
&fg\ (\{\#s\} + c1) = \{\} \ \text{by (auto simp add: mon-c-unconc)}
\end{aligned}
\text{moreover note IHAPP(2,4)}
\text{ultimately show ?thesis by blast}

\text{next}

\text{— Made an \text{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}
\text{case right}
\text{with HFMT(1) CASE(4) CEHFMT have CHFMT': ch = c1 + (csp + \{\#ssh\} + (c2 - \{\#ss\}))}
\text{by (simp add: union-ac)}
\text{have VALID: valid fg (\{\#s\} + c1 + ((csp + \{\#ssh\}) + (c2 - \{\#ss\}))]}  
\text{proof –}
\text{from ntr-valid-preserve-s[OF gtrI-s, OF CASE(5)] Cons.prems(2) CASE(2) have valid fg ((\{\#ss\} + ((\{\#s\} + ceh)) by (auto simp add: union-ac add-mset-commute)}
\text{with right CEHFMT show ?thesis by (auto simp add: union-ac add-mset-commute)}
\text{qed}
\text{from Cons.hyps[OF - VALID, of s' c'] CHFMT' SPLIT(2) CASE(3) obtain}
\begin{aligned}
&w1\ w2\ c1'\ c2'\ \text{where IHAPP:}\ w \in w1 \odot_{\text{anl s}}\ fg\ \text{map}\ \text{ENV}\ w2\ c' = c1' + c2'
\end{aligned}
\begin{aligned}
&(\{s, c1, w1, s', c1'\}) \in \text{trcl (ntrp s)}\ (\text{map}\ le-rem-s\ w1) (\text{csp +}\ \{\#ssh\}) + (c2 - \{\#ss\}))
\end{aligned}
\begin{aligned}
&w2, c2' \in \text{trcl (ntrp s)}
\end{aligned}
\begin{aligned}
&\text{mon-ww fg (map}\ le-rem-s\ w1) \cap\ mon-c\ fg\ (\text{csp +}\ \{\#ssh\}) + (c2 - \{\#ss\})) = \{\}
\end{aligned}
\text{by blast}
\text{have ee \# w \in w1 \odot_{\text{anl s}}\ fg\ map\ \text{ENV}\ (e\#w2) proof (simp add: CASE(1), rule cil-cons2)}
\text{from IHAPP(5) have mon-ww fg (map}\ le-rem-s\ w1) \cap\ mon-s\ fg\ ssh = \{\}
\text{by (auto simp add: mon-c-unconc)}
\text{moreover from ntrp-mon-s[OF CASE(5)] CASE(1) have} \ \text{fst (\text{anl s}\ f gg) ee c' +}\ mon-s\ f g\ ssh\ by\ auto
\text{ultimately have} \ \text{fst (\text{anl s}\ f gg\ ee)} \cap\ mon-ww\ f g\ (\text{map\ le-rem-s}\ w1) = \{\}
\text{by auto}
\text{moreover have mon-pl (\text{map}\ (\text{anl s}\ f g)\ w1) = mon-ww\ f g\ (\text{map}\ le-rem-s\ w1) by (unfold\ \text{anl-def'}) (simp add: cil-an-cons-helper[symmetric])}
\text{ultimately show} \ \text{fst (\text{anl s}\ f (\text{ENV}\ e)) \cap\ mon-pl\ (\text{map}\ (\text{anl s}\ f g)\ w1) = \{\}
\text{using CASE(1) by auto}
\text{qed (rule IHAPP(1))}
\text{moreover}
\text{have SS: (c2, e, csp + \{\#ssh\} + (c2 - \{\#ss\})) \in \text{ntrp s} proof –}
from right HFMT(1) have \(#s+\)ce-\(#s+\)c1+(\(#s+\)-\(#s+\)) \(#s+\)+ceh = csp+(\(#s+\)+c1+(\(#s+\)-\(#s+\))) by (simp-all add: union-ac)
  with CASE(5) ntr-trans-context-s[of ss \(#s+\)+c1+(\(#s+\)-\(#s+\))] e
ssh csp fg c2 \(#s+\)-\(#s+\)] have
  ((ss, (c2 - \(#s+\))), e, ssh, csp+(c2 - \(#s+\))) \(\in\) ntr 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 (e2, e # w2, c2') \(\in\) trcl (ntr fg).
  moreover
  from ntr-mon-increasing-s[OF SS] IHAPP(5) have mon-ww fg (map le-rem-s
w1) \(\cap\) mon-c fg c2 = {} by (auto simp add: mon-c-unconc)
  moreover
  from ntr-trans-contexts[OF CASE(5)] right IHAPP(6) have mon-ww fg
(e#w2) \(\cap\) 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')\(\in\)trcl (ntr fg)
  and VALID: valid fg (ca+cb)
  shows \(\exists\ ca' cb' wa wb.
    c'=ca'+cb' \land
    w\(\in\)wa\(\otimes\)cb an fg(wb) \land
    mon-c fg ca \(\cap\) (mon-c fg cb \(\cup\) mon-ww fg wb) = {}
    \land
    mon-c fg cb \(\cap\) (mon-c fg ca \(\cup\) mon-ww fg wa) = {}
    \land
    (ca,wa,ca')\(\in\)trcl (ntr fg) \land
    (cb,wb,cb')\(\in\)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 multisets-configurations.
The general version for loc/env-configurations is shown later.

**lemma** (in flowgraph) ntr-unsplit:
  assumes A: w\(\in\)wa\(\otimes\)an fg wb and
B: (ca,wa,ca')\(\in\)trcl (ntr fg)
  (cb,wb,cb')\(\in\)trcl (ntr fg)
  mon-c fg ca \(\cap\) (mon-c fg cb \(\cup\) mon-ww fg wb) = {}
  mon-c fg cb \(\cap\) (mon-c fg ca \(\cup\) mon-ww fg wa) = {}
  shows (ca+cb,w,ca'+cb')\(\in\)trcl (ntr fg)
  proof —
  — We have to generalize and rewrite the goal, in order to apply Isabelle’s induction
method
  from A have \(\forall\ ca cb.\ (ca,wa,ca')\(\in\)trcl (ntr fg) \land\ (cb,wb,cb')\(\in\)trcl (ntr fg) \land

79
\text{mon-c} \, fg \, ca \cap (\text{mon-c} \, fg \, cb \cup \text{mon-ww} \, fg \, w2) = \{\} \land \text{mon-c} \, fg \, cb \cap (\text{mon-c} \, fg \, ca \cup \text{mon-ww} \, fg \, wa) = \{\} →

\{ca+cb, w, ca'+cb\} \in \text{trcl} (\text{ntr} \, fg)

We prove the generalized goal by induction over the structure of consistent interleaving

\textbf{proof (induct rule: cil-set-induct-fixa)}

- If both words are empty, the proposition is trivial

\textbf{case empty thus ?case by simp}

\textbf{next}

- The first macrostep of the combined path was taken from the left operand of the interleaving

\textbf{case (left e w' w1' w2) thus ?case}

\textbf{proof (intro allI, goal-cases)}

\textbf{case (I ca cb)}

\textbf{hence I:} w' ∈ w1' \otimes \alpha \, fg \, w2 \, \text{fst} (\alpha \, fg \, e) \cap \text{mon-pl} (\text{map} (\alpha \, fg) \, w2) = \{\}

\text{!ca cb.}

[\text{[(ca, w1', ca')] ∈ \text{trcl} (\text{ntr} \, fg);}]

(cb, w2, cb') ∈ \text{trcl} (\text{ntr} \, fg);

\text{mon-c} \, fg \, ca \cap (\text{mon-c} \, fg \, cb \cup \text{mon-ww} \, fg \, w2) = \{\};

\text{mon-c} \, fg \, cb \cap (\text{mon-c} \, fg \, ca \cup \text{mon-ww} \, fg \, w1') = \{\} \implies

\text{(ca + cb, w', ca' + cb') ∈ trcl (ntr fg)}

\text{(ca, e \# w1', ca') ∈ trcl (ntr fg)} (cb, w2, cb') ∈ trcl (ntr fg)

\text{mon-c} \, fg \, ca \cap (\text{mon-c} \, fg \, cb \cup \text{mon-ww} \, fg \, w2) = \{\}

\text{mon-c} \, fg \, cb \cap (\text{mon-c} \, fg \, ca \cup \text{mon-ww} \, fg \, w1') = \{\} \, \text{by blast+}

- Split the left path after the first step

\textbf{then obtain cah where SPLIT:} (ca,e,ca')∈\text{ntr} \, fg \, (cah,w1',ca')∈\text{trcl} (\text{ntr} \, fg)

\text{by (fast dest: trcl-uncs)}

- and combine the first step of the left path with the initial right context

\textbf{from ntr-add-context-s[OF SPLIT(1), where cah=cb] I(7) have (ca + cb, e, cah + cb) ∈ ntr fg by auto}

\textbf{also}

- The rest of the path is combined by using the induction hypothesis

\textbf{have (ca + cb, w', ca' + cb') ∈ trcl (ntr fg) proof –}

\textbf{from I(2,6,7) ntr-mon-s[OF SPLIT(1)] have MON-CAH:} mon-c \, fg \, cah \cap (\text{mon-c} \, fg \, cb \cup \text{mon-ww} \, fg \, w2) = \{\} \, \text{by (cases e) (auto simp add: cil-\alpha-cons-helper)}

\text{with I(7) have MON-CB: mon-c \, fg \, cb \cap (\text{mon-c} \, fg \, cah \cup \text{mon-ww} \, fg \, w1') = \{\} by auto}

\textbf{from I(3)[OF SPLIT(2) I(5) MON-CAH MON-CB] show ?thesis .}

\textbf{qed}

\textbf{finally show ?case .}

\textbf{qed}

\textbf{next}

- The first macrostep of the combined path was taken from the right path – this case is done completely analogous

\textbf{case (right e w' w2' w1) thus ?case}

\textbf{proof (intro allI, goal-cases)}

\textbf{case (I ca cb)}

\textbf{hence I:} w' ∈ w1 \otimes \alpha \, fg \, w2' \, \text{fst} (\alpha \, fg \, e) \cap \text{mon-pl} (\text{map} (\alpha \, fg) \, w1) =
{} !!ca cb.
  \[(ca, \mathit{w1}, ca') \in \mathcal{trcl}(\mathit{ntr}\ fg)\];
  \((cb, \mathit{w2}', cb') \in \mathcal{trcl}(\mathit{ntr}\ fg)\);
  \(\mathit{mon-c}\ fg\ ca \cap (\mathit{mon-c}\ fg\ cb \cup \mathit{mon-ww}\ fg\ \mathit{w2'}) = \{\}\);  
  \(\mathit{mon-c}\ fg\ cb \cap (\mathit{mon-c}\ fg\ ca \cup \mathit{mon-ww}\ fg\ \mathit{w1}) = \{\}\) "#proof (\mathit{ntr}\ fg) 
  (ca + cb, w', ca' + cb') \in \mathcal{trcl}(\mathit{ntr}\ fg)
  (ca, \mathit{w1}, ca') \in \mathcal{trcl}(\mathit{ntr}\ fg)\)" (cb, e\#w2', cb') \in \mathcal{trcl}(\mathit{ntr}\ fg)
  \(\mathit{mon-c}\ fg\ ca \cap (\mathit{mon-c}\ fg\ cb \cup \mathit{mon-ww}\ fg\ (e\#w2')) = \{\}\)
  \(\mathit{mon-c}\ fg\ cb \cap (\mathit{mon-c}\ fg\ ca \cup \mathit{mon-ww}\ fg\ \mathit{w1}) = \{\}\)" by \texttt{blast+}
  \textbf{then obtain cbh where SPLIT:} \((cb, e, cbh) \in \mathcal{trcl}(\mathit{ntr}\ fg)\)\ (cbh, w', cb') \in \mathcal{trcl}(\mathit{ntr}\ fg)\) by (fast dest:\ \texttt{trcl-unscons})
  \textbf{from} \texttt{ntr-add-context-s} (OF SPLIT(1))\ \textbf{where cn=ca} \ I(6) \ \textbf{have} \ (ca + cb, e, ca + cbh) \ \in \mathcal{trcl}(\mathit{ntr}\ fg)\ \textbf{by} (auto simp add: union-commute)
  \textbf{also}
  \textbf{have} \ (ca + cbh, w', ca' + cb') \ \in \mathcal{trcl}(\mathit{ntr}\ fg)\ \textbf{proof} --
  \textbf{from} \texttt{I(2,6,7) ntr-mon-s} (OF SPLIT(1)) \ \textbf{have} MON-CBH: \(\mathit{mon-c}\ fg\ cbh \cap (\mathit{mon-c}\ fg\ ca \cup \mathit{mon-ww}\ fg\ \mathit{w1}) = \{\}\) by (cases e) (auto simp add: cil-ca-cons-helper)
  \textbf{with} \texttt{I(6) have} MON-CA: \(\mathit{mon-c}\ fg\ ca \cap (\mathit{mon-c}\ fg\ cbh \cup \mathit{mon-ww}\ fg\ \mathit{w2'}) = \{\}\) by \texttt{auto}
  \textbf{from} \texttt{I(3)} (OF \texttt{I(4)} SPLIT(2) MON-CA MON-CBH) \ \textbf{show} \ ?thesis .
  \textbf{qed}
  \textbf{finally show} \ ?case .
  \textbf{qed}
  \textbf{qed}
  \textbf{with} B \ \textbf{show} \ ?thesis \ \textbf{by} \ \texttt{blast }
\textbf{qed}

\textbf{lemma} \ (in flowgraph) \ \texttt{ntrp-unsplit:}
\textbf{assumes} A: \(w \in wa \odot \mathit{andr}\ fg\) \ (map ENV wb) \ \textbf{and}
B: \((s, ca), wa, (s', ca') \in \mathcal{trcl}(\mathit{ntrp}\ fg)\)
\((cb, wb, cb') \in \mathcal{trcl}(\mathit{ntrp}\ fg)\)
\(\mathit{mon-c}\ fg\ ((\#s\#) + ca) \cap (\mathit{mon-c}\ fg\ cb \cup \mathit{mon-ww}\ fg\ \mathit{w1}) = \{\}\)
\(\mathit{mon-c}\ fg\ cb \cap (\mathit{mon-c}\ fg\ ((\#s\#) + ca) \cup \mathit{mon-ww}\ fg\ (\mathit{map}\ \mathit{le-rem-s}\ wa)) = \{\}\)
\(\textbf{shows} \ ((s, ca + cb), w, (s', ca' + cb')) \in \mathcal{trcl}(\mathit{ntrp}\ fg)\)
\textbf{proof} --
\{ \textbf{fix} wb' \}
\textbf{have} \(w \in wa \odot \mathit{andr}\ \mathit{ntrp}\ wb' \implies \forall s\ ca\ cb\ wb.\ wb'\ =\ \mathit{map}\ ENV\ wb\ \wedge\ \)
\((s, ca), wa, (s', ca') \in \mathcal{trcl}(\mathit{ntrp}\ fg)\) \ \wedge\ \((cb, wb, cb') \in \mathcal{trcl}(\mathit{ntrp}\ fg)\) \ \wedge\ \mathit{mon-c}\ fg\ ((\#s\#) + ca) \ \cap\ \(\mathit{mon-c}\ fg\ cb \cup \mathit{mon-ww}\ fg\ \mathit{w1}) = \{\}\ \wedge\ \mathit{mon-c}\ fg\ cb \ \cap\ \(\mathit{mon-c}\ fg\ ((\#s\#) + ca) \ \cup\ \mathit{mon-ww}\ fg\ (\mathit{map}\ \mathit{le-rem-s}\ wa)) = \{\}\ \implies\)
\((s, ca + cb), w, (s', ca' + cb') \in \mathcal{trcl}(\mathit{ntrp}\ fg)\)
\textbf{proof} \ (induct rule: cil-set-induct-fixa)
\textbf{case} empty \ \textbf{thus} ?case \ \textbf{by} \ \texttt{simp}
\textbf{next}
\textbf{case} (left e \ w' \ w1' \ w2')
\textbf{thus} ?case
\textbf{qed

81
\[
\text{proof (intro allI impI, goal-cases)}
\]
\[
\text{case (1 s ca cb wb)}
\]
\[
\text{hence } I: w' \in w' \odot_\alpha n_l f g \ w2 \ \text{fst (} \alpha n_l f g \ e) \cap \text{ mon-pl (} \text{map (} \alpha n_l f g \text{) w2)} = {}
\]
\[
\text{hence } I: w' \in w' \odot_\alpha n_l f g \ w2 \ \text{fst (} \alpha n_l f g \ e) \cap \text{ mon-pl (} \text{map (} \alpha n_l f g \text{) w2)} = {}
\]
\[
\text{hence } I: w' \in w' \odot_\alpha n_l f g \ w2 \ \text{fst (} \alpha n_l f g \ e) \cap \text{ mon-pl (} \text{map (} \alpha n_l f g \text{) w2)} = {} \]
\[
\text{by blast+}
\]
\[
\text{then obtain } \text{sh cah where SPLIT: (s,ca),e,(sh,cah)} \in \text{ntrp f g (} \text{(sh,cah),w1'},(s',ca')\text{)\in trcl (ntrp f g) by (fast dest: trcl-uncons)}
\]
\[
\text{from ntrp-add-context-s(OF SPLIT(1), of cb] I(8) have (s, ca + cb), e, sh, cah + cb} \in \text{ntrp f g by auto}
\]
\[
\text{also have ((sh,cah+c+cb),w1',s',ca'+cb'))\in trcl (ntrp f g) \text{ proof (rule I(3))}
\]
\[
\text{from ntrp-mon-s(OF SPLIT(1)] I(2, 4, 7, 8) show 1: mon-c f g (\{}n\text{sh#}\} + cah) \cap (\text{mon-c f g cb} \cup \text{mon-ww f g w#}) = {} \}
\]
\[
\text{by (cases e) (rename-tac a, case-tac a, simp add: cil-cm-cons-helper, fastforce simp add: cil-cm-cons-helper)+}
\]
\[
\text{from I(8) I show mon-c f g cb} \cap (\text{mon-c f g} (\{}n\text{sh#}\} + cah) \cup \text{mon-ww f g (map le-rem-s w1')) = {} \}
\]
\[
\text{by auto}
\]
\[
\text{qed (auto simp add: I(4,6) SPLIT(2)}
\]
\[
\text{finally show ?case .}
\]
\[
\text{qed}
\]
\[
\text{next}
\]
\[
\text{case (right ee w' w2' w1) }
\]
\[
\text{thus ?case}
\]
\[
\text{proof (intro allI impI, goal-cases)}
\]
\[
\text{case (1 s ca cb wb)}
\]
\[
\text{hence I: w' \in w1} \odot_\alpha n_l f g \ w2' \ \text{fst (} \alpha n_l f g \ ee) \cap \text{ mon-pl (} \text{map (} \alpha n_l f g \text{) w1)} = {}
\]
\[
\text{hence I: w' \in w1} \odot_\alpha n_l f g \ w2' \ \text{fst (} \alpha n_l f g \ ee) \cap \text{ mon-pl (} \text{map (} \alpha n_l f g \text{) w1)} = {}
\]
\[
\text{w2'} = \text{map ENV wb;}
\]
\[
\text{((s, ca), w1', s', ca') } \in \text{trcl (ntrp f g);}
\]
\[
\text{(cb, wb, cb')} \in \text{trcl (ntrp f g);}
\]
\[
\text{mon-c f g (\{}n\text{sh#}\} + cah) \cap (\text{mon-c f g cb} \cup \text{mon-ww f g w#}) = {} ;}
\]
\[
\text{mon-c f g cb} \cap (\text{mon-c f g} (\{}n\text{sh#}\} + cah) \cup \text{mon-ww f g (map le-rem-s w1)}) = {}
\]
\[
\text{I } \Rightarrow (\{(s, ca + cb), w1', s', ca'+cb') \in \text{trcl (ntrp f g)}
\]
\[
\text{ee#w2'} = \text{map ENV wb}
\]

82
\[(s, ca), w1, s', ca') \in \text{trel}(\text{ntrp fg})
\]
\[(cb, wb, cb') \in \text{trel}(\text{ntr fg})
\]
\[\text{mon-c fg} (\{\#s\#\} + ca) \cap (\text{mon-c fg} cb \cup \text{mon-ww fg} wb) = \{}
\]
\[\text{mon-c fg} cb \cap (\text{mon-c fg} (\{\#s\#\} + ca) \cup \text{mon-ww fg} (\text{map le-rem-s w1})) = \{}
\]
\[\text{by fastforce+}
\]
\[\text{from I(4) obtain } e \text{ wb' where } \text{EE: wb'=e#wb' ee=ENV e w2'=map ENV wb'} \text{ by (cases wb, auto)}
\]
\[\text{with I(6) obtain } cbh \text{ where } \text{SPLIT:} (cb, e, cbh) \in \text{trel}(\text{ntr fg}) \text{ by (fast dest: trcl-uncs)}
\]
\[\text{have } ((s, ca + cbh)), ee, (s, ca + cbh)) \in \text{ntrp fg} \text{ proof -}
\]
\[\text{from gtrE[OF SPLIT(1)] obtain sb ceb sbh cebh where } \text{NTRS: cb = add-mset sb ceb cbh = add-mset sbh cebh ((sb, ceb), e, sbh, cebh) \in ntrs fg} .
\]
\[\text{from ntrs-add-context-s[OF NTRS] of } \{\#s\#\} +\{ca\} \text{ EE(1) I(7) have}
\]
\[(\{s, add-mset s (ca+cbh)), e, sbh, add-mset s (ca+cbh)) \in \text{ntrs fg by (auto simp add: union-nc)}
\]
\[\text{from gtrp-env[OF this] NTRS(1,2) EE(2) show } \text{thesis by (simp add: union-nc)}
\]
\[\text{qed}
\]
\[\text{also have } ((s, ca+cbh), w', (s', ca'+cbh)) \in \text{trel}(\text{ntrp fg}) \text{ proof (rule I(3))}
\]
\[\text{from ntr-mon-s[OF SPLIT(1)] I(2, 4, 7, 8) EE(2) show 1: mon-c fg cbh \cap (mon-c fg (\{\#s\#\} + ca) \cup \text{mon-ww fg} (\text{map le-rem-s w1})) = \{\}
\]
\[\text{by (cases e) (simp add: cil-axl-cons-helper; fastforce simp add: cil-axl-cons-helper)}
\]
\[\text{from I(7) I EE(1) show mon-c fg (\{\#s\#\} + ca) \cap (mon-c fg cbh \cup \text{mon-ww fg} wb) = \{\} by auto}
\]
\[\text{qed (auto simp add: EE(3) I(5) SPLIT(2))}
\]
\[\text{finally show } ?\text{case .}
\]
\[\text{qed}
\]
\[\text{qed}
\]
\[\text{with } A B \text{ show } ?\text{thesis by blast}
\]
\text{qed}

And finally we get the desired theorem: \textit{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.

\textit{Theorem (in flowgraph) ntr-interleave: valid fg (ca+cb) \implies (ca+cb,w,c')\in\text{trel}(\text{ntr fg}) \iff }
\[
(\exists ca' cb' wa wb.
\quad c'\leftarrow ca'+cb' \land
\quad w\in(\text{wa}\cap\text{fn fg}w) \land
\quad \text{mon-c fg ca} \cap (\text{mon-c fg} cb \cup \text{mon-ww fg} wb) = \{\} \land
\quad \text{mon-c fg cb} \cap (\text{mon-c fg} ca \cup \text{mon-ww fg} wa) = \{\} \land
\quad (ca,wa,ca')\in\text{trel}(\text{ntr fg}) \land (cb,wb,cb')\in\text{trel}(\text{ntr fg}))
\]
\text{by (blast intro: trcl-split ntr-nsplit)}

— Here is the corresponding version for executions with an explicit local thread
**Theorem (in flowgraph) ntrp-interleave:**

\[
\text{valid}\ fg\ \{(\#s\#)+c1+c2\} \implies \\
((s,c1+c2),w,(s',c')) \in \text{trcl}\ (\text{ntrp}\ fg) \iff \\
(\exists\ w1\ w2\ c1'\ c2'). \\
\]

\[
\begin{align*}
&\quad \text{w} \in \text{w1} \otimes_{\alpha_n} \text{fg}\ (\text{map}\ ENV\ w2) \land \\
&\quad c' = c1' + c2' \land \\
&\quad ((s,c1),w1,(s',c1')) \in \text{trcl}\ (\text{ntrp}\ fg) \land \\
&\quad (c2,w2,c2') \in \text{trcl}\ (\text{ntrp}\ fg) \land \\
&\quad \text{mon-ww}\ fg\ (\text{map}\ \text{le-rem-s}\ w1) \cap \\
&\quad \text{mon-c}\ fg\ c2 = \{\} \land \\
&\quad \text{mon-ww}\ fg\ w2 \cap \text{mon-c}\ fg\ ((\#s\#)+c1) = \{\} \\
\end{align*}
\]

**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:**

\[
\begin{align*}
&\quad (c,w,c') \in \text{trcl}\ (\text{ntr}\ fg); \\
&\quad \text{mon-c}\ fg\ (\text{add-mset}\ s\ \text{cl}) \cap \text{mon-c}\ fg\ c \cup \text{mon-ww}\ fg\ w = \{\} \\
\end{align*}
\]

**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:**

\[
\begin{align*}
&\quad (c,w,\{\#s\#\}+ce) \in \text{trcl}\ (\text{ntr}\ fg); \\
&\quad \text{valid}\ fg\ c\ \implies \exists\ s\ ce\ w1\ w2\ ce1'\ ce2'. \\
&\quad c=\{\#s\#\}+ce \land \\
&\quad ce'=ce1'+ce2' \land \\
&\quad w \in \text{w1} \otimes_{\alpha_n} \text{fg}\ w2 \land \\
&\quad \text{mon-s}\ fg\ s \cap \text{mon-c}\ fg\ ce \cup \text{mon-ww}\ fg\ w2 = \{\} \land \\
&\quad \text{mon-c}\ fg\ ce \cap \text{mon-s}\ fg\ s \cup \text{mon-ww}\ fg\ w1 = \{\} \land \\
&\quad ((\#s\#),w1,\{\#s\#\}+ce1') \in \text{trcl}\ (\text{ntr}\ fg) \land \\
&\quad (ce,\text{w2},ce2') \in \text{trcl}\ (\text{ntr}\ fg) \\
\end{align*}
\]

— 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 \(\text{?case} \ldots\)

next

— The initial configuration has the form \(\{\#s\}\) + \(ce\).

case (add \(ce\) \(s\))

— We split the path by this initial configuration

from mon-split[(\(\text{OF}\ add\-\prems\{1,2\}\)] obtain \(ce'\) \(ce2'\) \(w1\) \(w2\) where

\[\text{SPLIT: add-mset } s' ce'=ce' + ce2' w \in w1 \tau_{\text{an}} fg w2\]

\[\text{mon-c fg ce } \cap (\text{mon-s fg s } \cup \text{mon-ww fg w1}) = \{\}\]

\[\text{mon-s fg s } \cap (\text{mon-c fg ce } \cup \text{mon-ww fg w2}) = \{\}\]

\[\{\#s\}, w1, ce'\} \in \text{trcl (ntr fg)}\]

\[(ce, w2, ce2') \in \text{trcl (ntr fg)}\]

by auto

— And then check whether splitting off \(s\) was the right choice

from SPLIT(1) show \(\text{?thesis} \) by fastforce

next

— Our choice was correct, \(s'\) is generated by some descendant of \(s''\)

case left

with SPLIT show \(\text{?thesis} \) by fastforce

next

— Our choice was not correct, \(s'\) is generated by some descendant of \(ce\)

case right with SPLIT(6) have \(C\): \((ce, w2, \{\#s'\}) + (ce2' - \{\#s'\})\) \(\in\) \text{trcl (ntr fg)}

by \((\text{simp-all add: valid-uncone})\)

from \(\text{add.hyps}[\text{OF } C \text{ VALID}(1)]\) obtain \(st\) \(cet\) \(w21\) \(w22\) \(ce21\) \(ce22\) where

IHAPP:

\[cc=\{\#st\}\ ) + cet\]

\[ce2' = \{\#s'\} = ce21' + ce22'\]

\[w2 \in w21 \cap_{\text{an}} fg w22\]

\[\text{mon-s fg st } \cap (\text{mon-c fg cet } \cup \text{mon-ww fg w22}) = \{\}\]

\[\text{mon-c fg cet } \cap (\text{mon-s fg st } \cup \text{mon-ww fg w21}) = \{\}\]

\[(\{\#s\}, w21, \{\#s'\} + ce21') \in \text{trcl (ntr fg)}\]

\[(cet, w22, ce22') \in \text{trcl (ntr fg)}\]

by \(\text{blast}\)

— And finally we add the path from \(s\) again. This requires some monitor sorting

and the associativity of the consistent interleaving operator.

from \(\text{cil-assoc2} \{\text{of } w1 - w2 \text{ w22 w21}\}\) \(\text{SPLIT}(2)\) \(\text{IHAPP}(3)\) obtain \(wl\) where

\(C\text{ASSOC: w} \in w21 \cap_{\text{an}} fg w21\)

by \((\text{auto simp add: cil-commute})\)

from \(\text{CASSOC } \text{IHAPP}(1,3,4,5) \text{ SPLIT}(3,4)\) \(\text{have COMBINE: add-mset s cet, w1, ce1' + ce22' } \in \text{trcl (ntr fg)}\) using \(\text{ntr-ansplit}[\text{OF } \text{CASSOC}(2) \text{ SPLIT}(5)\]

\(\text{IHAPP}(7)]\) by \((\text{auto simp add: mon-c-uncconc mon-ww-cil Int-Un-distrib})\)

moreover from \(\text{CASSOC } \text{IHAPP}(1,3,4,5) \text{ SPLIT}(3,4)\) \(\text{have mon-s fg st } \cap (\text{mon-c fg } \{\#s\} + cet) \cup \text{mon-ww fg w1} ) = \{\}\)

\(\text{mon-c fg } \{\#s\} + cet ) \cap (\text{mon-s fg st } \cup \text{mon-ww fg w2} ) = \{\}\)

by \((\text{auto simp add: mon-c-uncconc mon-ww-cil-cil})\)

moreover from \(\text{right } \text{IHAPP}(1,2)\) have \(\{\#s\} + cet = \{\#st\} + (\{\#s\} + cet)\)

\(ce' = ce21' + (ce1' + ce22')\)

by \((\text{simp-all add: union-cc})\)

moreover note \(\text{IHAPP}(6) \text{ CASSOC}(1)\)

ultimately show \(\text{?thesis} \) by fastforce

qed
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 \( R[v] \sqsubseteq \text{rhs} \), where \( R[v] \in V \) and \( \text{rhs} \) 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:

\[
\text{for } (u, \text{Call } q, v) \in \mathcal{E}: \quad S^k[v] \supseteq \{(\text{mon}(q) \cup M \cup M', \hat{P}) \mid (M, P) \in S^k[u] \land (M', P') \in S^k[q], \hat{P} \subseteq P \cup P' \land |\hat{P}| \leq 2\}
\]

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 \((\mathcal{P}(\mathcal{L}), \subseteq)\) for some type \( \mathcal{L} \). 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 \( S\text{-cs } fg \):

\[
\begin{align*}
\langle (u, \text{Call } q, v) &\in \mathcal{E} \rangle; \langle (u, M, P) \in S\text{-cs } fg \rangle; \\
(\text{return } fg q, M, Ps) &\in S\text{-cs } fg \quad \Rightarrow (v, \text{mon } fg q \cup M \cup Ms, P') \in S\text{-cs } fg
\end{align*}
\]

\(^3\)Variables are identified by control nodes here
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 $\subseteq$.

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 $P$.

\textbf{inductive-set}

$S\text{-cs} :: (\text{'n \times 'p \ set \times 'm set \times 'p multiset}) \ set$

\textbf{for} $fg k$

\textbf{where}

$S\text{-init}$: $(\text{entry } fg p, \{\}, \{\#\}) \in S\text{-cs } fg k$

$S\text{-base}$: $[(u, Base a, v) \in \text{edges } fg; (u, M, P) \in S\text{-cs } fg k] \Rightarrow (v, M, P) \in S\text{-cs } fg k$

$S\text{-call}$: $[(u, \text{Call } q, v) \in \text{edges } fg; (u, M, P) \in S\text{-cs } fg k; (\text{return } fg q, Ms, Ps) \in S\text{-cs } fg k; P' \subseteq \# + Ps; \text{ size } P' \leq k]$

$S\text{-call}$: $\Rightarrow (v, mon fg q \cup M \cup Ms, P') \in S\text{-cs } fg k$

$S\text{-spawn}$: $[(u, \text{Spawn } q, v) \in \text{edges } fg; (u, M, P) \in S\text{-cs } fg k; P' \subseteq \# + P; \text{ size } P' \leq k]$

$S\text{-spawn}$: $\Rightarrow (v, M, P') \in S\text{-cs } fg k$

The intuition underlying this constraint system is the following: The $S\text{-init}$-constraint describes that the procedures entry node can be reached with the empty path, that has no monitors and spawns no procedures. The $S\text{-base}$-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 $S\text{-call}$-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 $S\text{-spawn}$-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],\#),v,(\[v\],c'))\in trcl (trss fg);
  size P\leq k; (\lambda p. [entry fg p]) ' # P \subseteq # c']
\Rightarrow (v,mon-w fg w,P)\in 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))\in trcl (trss fg) ((sh,ch),e,([v],c'))\in 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)\in 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)\in 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') \in 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))\in trcl (trss fg)
    (([ut],ct),LCall q,([entry fg q,v],ct))\in trss fg
    (ut,Call q,v)\in edges fg
    ((([entry fg q],ct),w2,([return fg q],c'))\in 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],c'))\in trcl (trss fg) by (auto simp add: union-ac)
  from FC(1) have LEN: length w1 \leq length w length w2 \leq length w by auto
  from mset-map-split-orig-le SLP(1) snoc.prems(3) obtain P1 P2 where PSPLIT: P=P1+P2 (\lambda p. [entry fg p]) ' # P1 \subseteq # ct (\lambda p. [entry fg p]) ' # P2
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)** $\subseteq \#$ 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: (u, 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 (λp. [entry fg p]) ' # P1 ⊆ # {#|entry fg q|} (λp. [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@[e]),P)∈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

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 (λp. [entry fg p]) ' # P ⊆ # c' M =
\[ mon-w \text{ fg } w \text{ by blast} \]

\text{note IHAPP(1)}

\text{also from S-base have } \{([u],c'),\text{LBase a,}(v,c')\} \in \text{trss fg by (auto intro: trss-base)}

\text{finally have } \{([entry fg p], \{\#\}), w \otimes \text{[LBase a]}, (v, c') \} \in \text{trcl (trss fg)}.

\text{moreover from IHAPP(4) have } M = \text{mon-w fg} (w \otimes \text{[LBase a]} \text{ by (simp add: mon-w-unconc)}

\text{ultimately show } \text{?case using IHAPP(2,3,4) by blast}

\text{next case } (\text{S-call } u q v M P Ms Ps P') \text{ then obtain } p \text{ csp1 w1 where REACHING-PATH:}

\{([entry fg q], \{\#\}), w1, [u], \text{csp1}\} \in \text{trcl (trss fg)} \text{ size } P \leq k (\lambda p. \text{[entry fg p]})

\# P \sqsubseteq\# \text{ csp1 Ms = mon-w fg w1 by blast}

\text{from S-call obtain } \text{csp2 w2 where SL-PATH: } \{([entry fg q], \{\#\}), w2, [\text{return fg q}, \text{csp2}] \} \in \text{trcl (trss fg)} \text{ size } Ps \leq k (\lambda p. \text{[entry fg p]}) \quad \# Ps \sqsubseteq\# \text{ csp2 Ms = mon-w fg w2}

\text{by (blast dest: trss-er-path-proc-const)}

\text{from trss-c-no-mon[OF REACHING-PATH(1)] trss-c-no-mon[OF SL-PATH(1)]}

\text{have NOMON: mon-c fg csp1 = {}} \text{ mon-c fg csp2 = {}} \text{ by auto}

\text{have } \{([entry fg q], \{\#\}), w1 \circ \text{LCall } \text{q\# w2@LRet} [\text{[v,} \text{csp1+csp2}]\} \in \text{trcl (trss fg)} \text{ proof –}

\text{note REACHING-PATH(1)}

\text{also from trss-call[OF S-call(1)] NOMON have } \{([u],\text{csp1}),\text{LCall } q,([\text{entry fg q},v],\text{csp1})\} \in \text{trss fg by (auto)}

\text{also from trss-add-context[OF trss-stack-comp[OF SL-PATH(1)]]} \text{ NOMON have } \{([\text{entry fg q},v],\text{csp1}),w2,([\text{return fg q},v],\text{csp1+csp2})\} \in \text{trcl (trss fg)} \text{ by (simp add: union-ac)}

\text{also have } \{([\text{return fg q},v],\text{csp1+csp2}),\text{LRet}([v,\text{csp1+csp2}])\} \in \text{trss fg by (rule trss-ret)}

\text{finally show } \text{?thesis by simp}

\text{qed}

\text{moreover from REACHING-PATH(4) SL-PATH(4) have mon-fg q \cup M \cup Ms = mon-w fg (w1@LCall q\# w2@[LRet]) by (auto simp add: mon-w-unconc)}

\text{moreover have } (\lambda p. \text{[entry fg p]}) \quad \# (P') \sqsubseteq\# \text{ csp1+csp2 (is } \# P' \subseteq\# -)

\text{proof –}

\text{from image-mset-subseteq-mono[OF S-call(6)] have } \# P' \subsetq \# P \quad \# P + \# P \text{ by auto}

\text{also from mset-subset-eq-mono-add[OF REACHING-PATH(3) SL-PATH(3)]}

\text{have } \ldots \subsetq\# \text{ csp1+csp2 .}

\text{finally show } \text{?thesis .}

\text{qed}

\text{moreover note S-call(7)}

\text{ultimately show } \text{?case by blast}

\text{next case } (\text{S-spawn } u q v M P P') \text{ then obtain } p \text{ c' w where IHAPP: } \{([entry fg p], \{\#\}), w, [u], c' \} \in \text{trcl (trss fg)} \text{ size } P \leq k (\lambda p. \text{[entry fg p]}) \quad \# P \sqsubseteq\# \text{ c'} M = \text{mon-w fg w by blast}

\text{note IHAPP(1)}

\text{also from S-spawn(1) have } \{([u],c'),\text{LSpawn } q,([v],\text{add-mset } [\text{entry fg q}, \text{c'}])\} \in \text{trss fg by (rule trss-spawn)}

\text{finally have } \{([entry fg p], \{\#\}), w \otimes \text{[LSpawn q]}, [v], \text{add-mset } [\text{entry fg q}, \text{c'}] \}

\text{ultimately show } \text{(simp add: mon-w-unconc)}

\text{ultimately show } \text{?thesis .}

\text{qed}
moreover from IHAPP(4) have $M$-mon-w $fg$ ($w \oplus [LSpawn q]$) by (simp add: mon-w-unconc)
moreover have ($\lambda p. \ [entry \ fg \ p]) \ 'P' \subseteq Q \ \{\# [entry \ fg \ q] \} \ + \ c'$ (is $?f' \ 'P' \subseteq Q \ \{\# [entry \ fg \ q] \}) + $f' 'P$ by auto

also from mset-subset-eq-mono-add[OF - IHAPP(3)] have . . . $\subseteq Q \ \{\# [entry \ fg \ q] \} + c'$ 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:**

$v, M, P \subseteq S$-cs $fg \ k \longleftrightarrow$

$(\exists p v w. (\{\# [entry \ fg \ p], (\#), w, (v, c')\}) \in trcl (trss \ fg) \wedge$

size $P \leq k \wedge (\lambda p. \ [entry \ fg \ p]) \ 'P' \subseteq Q \ \{\# c' \wedge M$-mon-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 $A$: $(\{\# v, \#\}, ce1,(sh, ch)) \in ntrp \ fg$ and
CASE: $!!p u v w. \ \|

\ _$ce1$=LOC (LCall p#w);$

(u, Call p, u')$ $\in$ edges $fg;$

sh $= [v, u']$;

proc-of $fg$ $v$ $= p;$

mon-c $fg$ $ch$ $= \{\}$;

!!s. $s \in \# ch \implies \exists p u v. s = [entry \ fg \ p] \wedge$

(u, Spawn $p, v) \in$ edges $fg \wedge$

initialproc $fg$ $p;$$

!!P. (\lambda p. \ [entry \ fg \ p]) \ 'P' \subseteq Q \ \{\# ch \implies$

$v, mon$-w $fg \ w, P) \in S$-cs $fg$ (size $P)$

$\ | \implies Q$
shows $Q$
proof —
from $A$ obtain $ce$ where $EE$: $ce1$=LOC $ee$ $(\{\# u\}, \#\}, ee,(sh, ch)) \in ntrp \ fg$ by
(auto elim: qtrp.cases)

have $CHFMT$: $!!s. s \in \# ch \implies \exists p u v. s = [entry \ fg \ p] \wedge (u, Spawn \ p, v) \in$ edges $fg \wedge$

initialproc $fg$ $p$ by (auto intro: ntrs-c-cases-s[OF $EE(2)$])
with $c$-of-initial-no-mon have $CHMON$: mon-c $fg$ $ch$ $= \{\}$ by blast
from $EE(2)$ obtain $p u v w$ where $FIRSTSPLIT$: $ee$=$LCall p\# w (\{\# v\}, \#\}, LCall p, ([entry \ fg \ p, u], (\#)) \in trss \ fg$ $sh = [v, u']$ $(\{\{\\}, (\#), w, (v, ch)) \in trcl (trss$
fg) by (auto elim!: ntrs_cases[simplified])

from FIRSTSPLIT have EDGE: (u, Call p, u') ∊ 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-of fg v = p by simp
have !!P. (λp. [entry fg p]) ↛# P ⊑# ch ⇒ (v, mon-w fg w, P) ∊ S-cs fg (size P)

proof –

fix P assume (λp. [entry fg p]) ↛# P ⊑# 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) ∊ S-cs fg k and
P: proc-of fg v = p and
EDGE: (u, Call p, u') ∊ edges fg

shows ∃ w ch.

\(((v,\{\#\}),LOC (LCall p\#w),(\{v,u\}',ch)) ∊ ntrp fg ∧
size P ≤ k ∧
M = mon-w fg w ∧
mon-n fg v = mon fg p ∧
(λp. [entry fg p]) ↛# P ⊑# ch ∧
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: \(((\{\}),(\{\})),wsl,[v],ch) ∊ trcl (trss fg) size P ≤ k (λp. [entry fg p]) ↛# P ⊑# ch M = mon-w fg wsl by fastforce
from mon-n-same-prec[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-cases[OF SLPATH(1), simplified] have CHFMT: \∀ s. s ∈# ch ⇒ \∃ p. s = [entry fg p] ∧ (\exists 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 = \{

— From the constraints prerequisites, we can construct the first step
have FS: \(((\{\}),(\{\})),LCall p\#wsl,(\{v,u\}',ch)) ∊ ntrp fg proof (rule ntrs-step[where r=[]=, simplified])

from EDGE show \(((\{\}),(\{\})), LCall p, [entry fg p, u'], \{\}) ∊ trss fg by (auto intro: trss-call)

qed (rule SLPATH(1))

hence FSP: \(((\{\}),(\{\})),LOC (LCall p\#wsl),(\{v,u\}',ch)) ∊ 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 $(u, M_l, M_e, h) \in RU-cs \_fg \_U$ corresponds to a path from $\{\#u\} \_L$ to some configuration at $U$, that uses monitors from $M_l$ in the steps of the initial thread, monitors from $M_e$ 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.

\[
\text{inductive-set} \\
RU-cs :: (n, p, ba, m, \text{more}) \text{ flowgraph-rec-scheme } \Rightarrow \text{ 'n set } \Rightarrow \\
\text{ 'n } \times \text{ 'm set } \times \text{ 'm set } \times \text{ ('m } \Rightarrow \text{ 'm set )) set}
\]

for $fg \_U$

where

$\text{RU-init: } u \in U \implies (u,\{\},\lambda x.\{\}) \in RU-cs \_fg \_U$

$\text{RU-call: } \{ (u, Call \_p, u') \in \text{edges } fg \_U; \text{ proc-of } fg \_v = p; (v,M,P) \in S-cs \_fg \_0; (v,M_l,M_e,h) \in RU-cs \_fg \_U; \text{ mon-n } fg \_u \cap M_e = \{\} \} \}
\implies (u, \text{ mon-fg } p \cup M \cup M_l, M_e, ah-update h (\text{ mon-fg } p,M) (M_l \cup M_e)) \in RU-cs \_fg \_U$

$\text{RU-spawn: } \{ (u, Call \_p, u') \in \text{edges } fg \_U; \text{ proc-of } fg \_v = p; (v,M,P) \in S-cs \_fg \_1; q \in \# P; (entry f g q,M_l,M_e,h) \in RU-cs \_fg \_U; (\text{ mon-n } fg \_u \cup \text{ mon-fg } p) \cap (M_l \cup M_e) = \{\} \}
\implies (u, \text{ mon-fg } p \cup M, M_l \cup M_e, ah-update h (\text{ mon-fg } p,M) (M_l \cup M_e)) \in RU-cs \_fg \_U$

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 ($\text{mon-n fg } u \cap \text{Me} = \{\}$). 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 ($\text{(mon-n fg } u \cup \text{mon fg } p) \cap (\text{Ml} \cup \text{Me}) = \{\}$). 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 $A$: $\text{atU } U$ $c'$ and $B$: $(c,w,c') \in \text{trcl } (\text{ntr } fg)$
- **shows** $\exists s w' c1'$. $s \in # \land w' \preceq w \land c1' \subseteq # \land c' \land \text{atU } U$ $c1' \land ((\text{#s#},w',c1') \in \text{trcl } (\text{ntr } fg))$

**proof** –

- **obtain** $u i r c$ with $C$'sMT: $c' = (\text{#ui#r#}) + ce' wi \in U$ 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 $\text{Rsplit}$: $c = (\text{#s#}) + ce$ $ce' = ce1' + ce2'$ $w \in w1 \otimes an$ $fgw2$ $\{\text{#s#}\}, w1, \{\text{#ui#r#}\} + ce1' \in$ trcl (ntr $fg$) by blast
- **with** $C$'sMT have $s \in # \land w1 \preceq w \{\text{#ui#r#}\} + ce1' \subseteq # \land c' \text{ atU } U \{\text{#ui#r#}\} + ce1' \}$
- **by** (auto dest: cil-ileq) **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-at\( U \) 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.

<table>
<thead>
<tr>
<th><strong>Lemma</strong> (in flowgraph) RU-sound:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(!u s' c', [(([u],[#]),w,(s',c')) \in trcl (ntrp fg); atU U (add-mset s' c')]</td>
</tr>
<tr>
<td>(\Rightarrow \exists M l M e h. (u, M l, M e, h) \in RU-cs fg U \land M l \subseteq \text{mon-loc} fg w \land M e \subseteq \text{mon-env} fg w \land h \leq \alpha h (\text{map} (\alpha h fg) w))</td>
</tr>
</tbody>
</table>

— The proof works by induction over the length of the reaching path

<table>
<thead>
<tr>
<th><strong>Proof</strong> (induct ( w ) rule: length-compl-induct)</th>
</tr>
</thead>
<tbody>
<tr>
<td>— For a reaching path of length zero, the proposition follows immediately by the constraint RU-init</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Case</strong> Nil thus ?case by auto (auto intro!: RU-init)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Next</strong></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Case</strong> (Cons ( eel ) ( wwl ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>— For a non-empty path, we regard the first step and the rest of the path</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Then obtain</strong> ( sh ) ( ch ) where SPLIT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>((([u],[#]),eel,(sh, ch)) \in ntrp fg)</td>
</tr>
<tr>
<td>(((sh, ch), wwl,(s', c')) \in trcl (ntrp fg))</td>
</tr>
</tbody>
</table>

| **By** (fast dest: trcl-uncons) |

<table>
<thead>
<tr>
<th><strong>Obtain</strong> ( p u' v w ) where</th>
</tr>
</thead>
<tbody>
<tr>
<td>— The first step consists of an initial call and a same-level path</td>
</tr>
</tbody>
</table>

| **FS-FMT**: \( cel = \text{LOC} (LCall p \# w) (u, \text{Call} p, u') \in \text{edges} fg \text{ sh} = [v, u'] \text{ proc-of} fg v = p \text{ mon-c} fg ch = \{\}\) |

| — The only environment threads after the first step are the threads that where spawned by the first step |

| **And** CHFM: \( \land s. s \in \# \land ch \Rightarrow \exists p u v. s = [\text{entry} fg p] \land (u, \text{Spawn} p, v) \in \text{edges} fg \land \text{initialproc} fg p\) |

| — For the same-level path, we find a corresponding entry in the S-cs-constraint |
from the spawned threads length wwl by for the path possible for normalized paths

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

— 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

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

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

— This does not affect the configuration being at

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

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

— have MON-U-ME: mon-n fg u \cap Me = {} proof — from ntrp-mon-env-a-no-ctx[OF Cons.prems(1)] have mon-env fg wwl \cap mon-n fg u = {} by (auto)

— with mon-env-ileq[OF ILEQ] IHAPP(3) show \textit{?thesis by fast}

qed

from RU-call[OF FS-FMT(2,4) S-ENTRY IHAPP(1) MON-U-ME] have (v,
mon fg p ∪ mon-w fg w ∪ ML, Me, ah-update h (mon fg p, mon-w fg w) (MI ∪ 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) (MI ∪ Me) ≤ ah (map (anl fg) (eel#wwl)) proof (simp add: ah-update-cons)

show ah-update h (mon fg p, mon-w fg w) (MI ∪ Me) ≤ ah (map (anl fg) w1)

from IHAPP(4) have h ≤ ah (map (anl fg) w1).

also from ah-ileq[OF le-list-map[OF ILEQ]] have ah (map (anl fg) w1) ≤ ah (map (anl fg) wwl).

finally show h ≤ ah (map (anl fg) wwl).

next

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

next

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

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

finally show (MI ∪ Me) ⊆ mon-pl (map (anl 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[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-uncnc)

— 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' ∈ trel (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 ntr.gtr2gtrp[where c=|#], simplified, OF RI(5) obtain sr' cre' wwr where RI-NTRP: cr' = add-mset sr' cre' wwr = map le-rem-s wwr ([entry fg q],|#). wwr, (sr', cre') ∈ trel (ntr 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 MI Me h where IHAPP: (entry fg q, MI, Me, h) ∈ RU-cs fg U MI ⊆ mon-loc fg wwr Me ⊆ mon-env fg wwr h ≤ ah (map (anl 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, $\{\#q\}$) $\in$ S-cs fg 1 by auto

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

have MON-MLE-ENV: $Ml$ $\cap$ $Me$ $\subseteq$ mon-env fg wwl proof —

from IHAPP(2,3) have $Ml$ $\cup$ $Me$ $\subseteq$ mon-loc fg wwr $\cup$ mon-env fg wwr by auto

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

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

finally 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 $\cup$ mon-w fg w, $Ml$ $\cup$ $Me$, ah-update h (mon-fg p, mon-w fg w) ($Ml$ $\cup$ $Me$)) $\in$ RU-cs fg U by simp

— Next we have to assemble the rest of the proposition

moreover have mon fg p $\cup$ mon-w fg w $\subseteq$ mon-loc fg (eel#wwl) using FS-FMT(1) by fastforce

moreover have $Ml$ $\cup$ $Me$ $\subseteq$ mon-env fg (eel#wwl) using MON-MLE-ENV by auto

moreover have ah-update h (mon fg p, mon-w fg w) ($Ml$ $\cup$ $Me$) $\leq$ aah (map (aah fg) (eel#wwl)) — Only the proposition about the acquisition histories needs some more work

proof (simp add: ah-update-cons)

have MAP-HELPER: map (aah fg) wwr $\leq$ map (aah fg) wwl proof —

from RI-NTRP(2) have map (aah fg) wwr = map (aah fg) wr by (simp add: aah-aanl)

also from le-list-map[OF RI(2)] have ... $\leq$ map (aah fg) w2 .

also have ... = map (aah fg) (map ENV w2) by simp

also from le-list-map[OF ILEQ] have ... $\leq$ map (aah fg) wwl .

finally show ?thesis .

qed

show ah-update h (mon fg p, mon-w fg w) ($Ml$ $\cup$ $Me$) $\leq$ ah-update (aah (map (aah fg) wwl)) (aah fg eel) (mon-pl (map (aah fg) wwl)) proof (rule ah-update-mono)

from IHAPP(4) have h $\leq$ aah (map (aah fg) wwr) .

also have ... $\leq$ aah (map (aah fg) wwl) by (rule aah-ileq[OF MAP-HELPER])

finally show h $\leq$ aah (map (aah fg) wwl) .

next

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

next
from IHAPP(2,3) mon-pl-ileq[OF MAP-HELPER] show Ml ∪ Me ⊆ mon-pl (map (αnl fg) wsl) by (auto simp add: mon-pl-of-αnl)
qed

ultimately show ?thesis by blast

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)∈RU-cs fg U

∀ w s' c,'

(([u],{#}),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

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: ext[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') ∈ 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-call.hyps(2) S-precise[OF RU-call.hyps(3), simplified] trss-bot-proc-cons[where

s=[] and s'=[]], simplified obtain wsl ch where

SLPATH: ([[entry fg p], {#}), wsl, [v], ch) ∈ trcl (trss fg) M = mon-w fg wsl

by fastforce

from trss-cases[OF SLPATH(1), simplified] have CHFMT: ∀s. s ∈# ch ⇒

∃p. s = [entry fg p] ∧ (∃u v. (u, Spawn p, v) ∈ edges fg) ∧ initialproc p 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))∈ntrp fg proof (rule ntrp-step[where

r=[], simplified])

from RU-call.hyps(1) show (([u], {#}), LCall p, [entry fg p, u'], {#}) ∈ trss fg by (auto intro: trss-call)

qed (rule SLPATH(1))
hence FSP: \(((\{u\},\{\#\}), \text{LOC (LCall p \# wsl)}, (\{v,u\}, ch))\)\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-n fg u' \cap mon-env fg w = {} by (auto simp add: mon-n-def edges-part)
from CHNOMON show mon-ww fg (map le-rem-s w) \cap mon-c fg ch = {} by auto
qed
finally have \(((\{u\}, \{\#\}), \text{LOC (LCall p \# wsl)} \# w, s' @ [u'], c' + ch) \in trcl (ntrp fg)\).
   — 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 (\text{LOC (LCall p \# wsl)} \# w) = mon fg p \in M \cup Ml
using SLPATH(2) IHAPP(3) by auto
moreover have mon-env fg (\text{LOC (LCall p \# wsl)} \# w) = Me using IHAPP(4)
by auto
moreover have oah (map (\text{avl fg}) (\text{LOC (LCall p \# wsl)} \# w)) = ah-update h (\text{mon fg p, M}) (Ml \cup Me) proof —
   have oah (map (\text{avl fg}) (\text{LOC (LCall p \# wsl)} \# w)) = ah-update (oah (map (\text{avl fg} wsl) wsl)) (\text{mon fg p, mon-w fg wsl}) (\text{mon-pl (map (\text{avl fg} wsl) wsl)}) by (auto simp add: ah-update-cons)
also have \ldots = ah-update h (\text{mon fg p, M}) (Ml \cup Me) proof —
from IHAPP(5) have oah (map (\text{avl fg} wsl) wsl) = h .
moreover from SLPATH(2) have (mon fg p, mon-w fg wsl) = (mon fg p, M) by (simp add: mon-pl-of-avl)
moreover from IHAPP(3,4) have mon-pl (map (\text{avl fg} wsl) wsl) = Ml \cup Me
by (auto simp add: mon-pl-of-avl)
   ultimately show \text{?thesis by simp}
qed
finally show \text{?thesis} .
qed
ultimately show \text{?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:
   \(((\text{entry fg p}), \{\#\}), w, s', c' \in trcl (ntrp fg)\) atU U \{\# s'\#\} + c' \hspace{1em} \text{mon-loc fg w = Ml mon-env fg w = Me oah (map (\text{avl fg} wsl) wsl) = 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: \(((\text{entry fg p}), \{\#\}), wsl, [v], ch) \in trcl (trss fg) M = mon-w fg wsl size P \leq 1 (\lambda p. (\text{entry fg p})) \# P \leq\# ch by fastforce
with RU-spawn.hyps(4) obtain che where PFMT: P=(\# q\#\) 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: \forall s. s \in\# ch \implies
\( \exists p. s = [\text{entry } f g p] \wedge (\exists u v. (u, \text{Spawn } p, v) \in \text{edges } f g) \wedge \text{initialproc } f g p \text{ by blast} \)

with c-of-initial-no-mon have CHNOMON: mon-c f g ch = {} by blast
have FS: \(((u),\{\#\}), LCall p\#wsl,([v,u'],ch))\in ntrsp f g \text{ proof (rule ntrsp-step[where r=[], simplified])} \)
from RU-spawn.hyps(1) show \(((u),\{\#\}), LCall p, [\text{entry } f g p, u'], \{\#\}) \in \text{trss } f g \text{ by (auto intro: trss-call)}
qed (rule SLPATH(1))
hence FSP: \(((u),\{\#\}), LOC (LCall p\#wsl),([v,u'],ch))\in ntrsp f g \text{ by (blast intro: gtrsp-loc)}
also have \(((v,u', ch), \text{map ENV (map le-rem-s w)}), [v,u'], \text{che+({\#s'\#}+c'})\in \text{trcl (ntrsp fg)} \text{ proof} –
from IHAPP(3,4) have mon-ww fg (map le-rem-s w) \subseteq Ml \cup Ml 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') \cap mon-ww fg (map le-rem-s w) = {} by (auto simp add: mon-n-def edges-part)
with ntr2ntrp[OF gtrsp2gtr[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) \# \text{map ENV (map le-rem-s w)}, [v,u'], \text{che+({\#s'\#}+c'})\in \text{trcl (ntrsp 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 by 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 \cup Ml by using IHAPP(3,4) by (auto simp add: mon-ww-of-le-rem simp del: map-map)
moreover have aah (map (\#n l fg) (LOC (LCall p \# wsl) \# map ENV (map le-rem-s w))) = ah-update h (mon fg p, M) (Ml \cup Ml) proof –
have aah (map (\#n l fg) (LOC (LCall p \# wsl) \# map ENV (map le-rem-s w))) = ah-update (aah (map (\#n l fg) (map le-rem-s w))) (mon fg p, mon-w fg wsl)
(mon-pl (map (\#n l fg) (map le-rem-s w))) by (simp add: ah-update-cons o-assoc)
also have ... = ah-update h (mon fg p, M) (Ml \cup Ml) proof –
from IHAPP(5) have aah (map (\#n l fg) (map le-rem-s w)) = h by (simp add: aah-and)
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 (\#n l fg) (map le-rem-s w)) = Ml \cup Ml by (auto simp add: mon-pl-of-and aah-and)
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 $(u, Ml, Me) \in RUV-cs fg U V$ 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.

\begin{align*}
\text{inductive-set} \\
RUVCs &::= ('n', 'p', 'ba', 'm', 'more') flowgraph-rec-scheme \Rightarrow \\
'n' set \Rightarrow 'n' set \Rightarrow ('n \times 'm set \times 'm set) set
\end{align*}

\begin{align*}
\text{for } fg U V \\
\text{where}
\begin{align*}
RUVCall &::= (u, Call p, u') \in edges fg; \text{proc-of fg } v = p; (v, M, P) \in S-cs fg 0; \\
&(v, Ml, Me) \in RUV-cs fg U V; \text{mon-n fg } u \cap Me = \{\} \\
\Rightarrow (u, \text{mon fg } p \cup M \cup Ml, Me) \in RUV-cs fg U V \\
\text{RUV-spawn:} \\
&\quad (u, Call p, u') \in edges fg; \text{proc-of fg } v = p; (v, M, P) \in S-cs fg 1; q \in # P; \\
&(entry fg q, Ml, Me) \in RUV-cs fg U V; \\
&(\text{mon-n fg } u \cup \text{mon fg } p) \cap (Ml \cup Me) = \{\} \\
\Rightarrow (u, \text{mon fg } p \cup M, Ml \cup Me) \in RUV-cs fg U V \\
\text{RUV-split-le:} \\
&\quad (u, Call p, u') \in edges fg; \text{proc-of fg } v = p; (v, M, P) \in S-cs fg 1; q \in # P; \\
&(v, Ml, Me, h) \in RU-cs fg U; (entry fg q, Ml', Me', h') \in RU-cs fg V; \\
&(\text{mon-n fg } u \cup \text{mon fg } p) \cap (Me \cup Ml' \cup Me') = \{\}; h \in [s] h' \\
\Rightarrow (u, \text{mon fg } p \cup M \cup Ml, Me \cup Ml' \cup Me') \in RUV-cs fg U V \\
\text{RUV-split-ee:} \\
&\quad (u, Call p, u') \in edges fg; \text{proc-of fg } v = p; (v, M, P) \in S-cs fg 2; \\
&\quad \{\#q\} + \{\#q'\} \subseteq # P; \\
&(entry fg q, Ml, Me, h) \in RU-cs fg U; (entry fg q', Ml', Me', h') \in RU-cs fg V; \\
&(\text{mon-n fg } u \cup \text{mon fg } p) \cap (Me \cup Ml \cup Ml') = \{\}; h \in [s] h' \\
\Rightarrow (u, \text{mon fg } p \cup M, Ml \cup Me \cup Ml' \cup Me') \in RUV-cs fg U V
\end{align*}
\end{align*}

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_eel** 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:** $\forall u s' c'.
\begin{array}{l}
  \left[( (\{u\},\{\#\}), w,(s',c'))\in trcl (ntrp fg); \ atUV U V (\{#s'\}+c') \right] \\
  \implies \exists Ml Me. \\
  \begin{array}{l}
    (u,Ml,Me)\in RUV-cs fg U V \\
    Ml \subseteq mon-loc fg w \\
    Me \subseteq mon-env fg w
  \end{array}
\end{array}$

— 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:** $((\{u\},\{\#\}), ee,(sh, ch))\in ntrp fg ((sh, ch),ww,(s',c'))\in trcl (ntrp fg) \text{ by (fast dest: trcl-uncs)}$

**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: wu \in w1 \odw fg map ENV w2 c' = c1' + c2' ((sh, \{\#\}), w1, s', c1') \in trcl (ntrp fg) (ch, w2, c2') \in trcl (ntrp fg) mon-ww fg (map le-rem-s w1) \cap mon-c fg ch = {} \ \text{mon-ww fg w2} \cap \text{mon-s fg sh} = {} \text{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 = {}**

103
and CHFMT: ∃s ∈# ch → ∃p u v. s = [entry fg p] ∧ (u, Spawn p, v) ∈ edges fg y ∧ initialproc fg p

and S-ENTRY-PAT: ∃P. (λp. [entry fg p]) ‘# P ⊆# ch → (v, mon-w fg w, P) ∈ S-cs fg (size P)

by (rule S-sound-ntrp[OF SPLIT(1)]) blast

from ntrp-mon-ww-fg-nor-ctz[OF SPLIT(2)] edges-part[OF FS-FMT(2)] have MON-PU: mon-ww fg w1 ∩ (mon fg p ∪ mon-n fg w) = {} by (auto simp add: mon-n-def)

from cil-ieq[OF LESPLIT(1)] mon-loc-ieq[of w1 ww fg] mon-loc-ileq[of w1 ww fg] have MON1-LEQ: mon-loc fg w1 ⊆ mon-loc fg w2 ∪ mon-env fg w1 ⊆ mon-env fg w2

by auto

from cil-ieq[OF LESPLIT(1)] mon-loc-ileq[of map ENV w2 ww fg] have MON2-LEQ: mon-ww fg w2 ⊆ mon-env fg w2

by simp

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

from Cons.prems[OF - DECOMP-LOC(1)] have ATUV: 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'#+c1') + c2')

by simp

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

by auto

from RUVC-call[OF FS-FMT(2,4)] S-ENTRY-PAT[of v Simplified] IHAPP(1) have (u, mon fg p ∪ mon-u fg w ∪ Ml, Me) ∈ RUVC-s cs fg w2 using IHAPP(3) MON-PU MON1-LEQ by fastforce

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

moreover have Me ⊆ mon-env fg (ec#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' at-U s1' at-U s2' by (unfold atUV-def) auto

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

obtain q ceh w21 w22 ce21' ce22' where

REV_SPLIT: ch = (#[entry fg q]#)+ceh add-mset s2' ce2' = ce21'+ce22' w2 ∈ w21 @ch fg w222 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 (ntrp fg) (ceh, w22, ce22') ∈ trcl (ntrp fg)

proof —

from ntr-reverse-split[of ch w2 s1' (#s2'#+)+c2'] ntrp-valid-preserve-s[OF SPLIT(1), simplified] C2'FMT LESPLIT(4)
obtain seh ceh w21 w22 ce21' ce22' where
\[*: \text{ch} = \text{\#seh\#} + \text{ce2'} = \text{ce21'} + \text{ce22'} (w2 \in \text{w21} \cap \text{ch} \wedge \text{w22} = \text{\#seh\#} + \text{ce22'}) \]
mon-s fg seh \cap (mon-c fg ceh \cup \text{mon-ww fg w22}) = \{
\text{mon-c fg ceh} \cap (\text{mon-s fg seh} \cup \text{\text{mon-ww fg w21}}) = \{
\text{\text{\#seh\#}, w21, \text{\#s1\#} + \text{ce21'}} \in \text{trcl (ntr fg) (\text{ceh, w22, ce22'}) trcl (ntr fg) by (auto simp add: valid-uncnc)}
\]
from this(1) CHFMT[of seh] obtain q where seh = \text{\[\text{entry fg q}\]} by auto
with * have \text{ch} = \text{\#entry fg q\#} + \text{ceh add-mset s2' ce22'} = \text{ce21'} + \text{ce22'} (w2 \in \text{w21} \cap \text{ch} \wedge \text{w22} = \text{\#entry fg q\#} + \text{ceh \cup \text{\text{mon-ww fg w22}) = \{\}}\text{mon-c fg ceh} \cap (\text{mon-fg q} \cup \text{\text{\text{\#s1\#} + ce21'}) \in \text{trcl (ntr fg) (\text{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 ntrp.gqe2gtrp[where c = \{\#\}, simplified, OF REVSPILT(6)] obtain sq' csp-q w2w1 where
\[ R-CONV: \text{add-mset s' ce21'} = \text{add-mset sq' csp-q w21 = map le-rem-s w21 w21 } \]
\[ μ(\text{\text{\#entry fg q\#}}, \{\#, \}), \text{w21, sq', csp-q} \in \text{trcl (ntrp fg) by auto} \]
from cil-ileq[OF REVSPILT(3)] \text{mon-ww-ileq[of w21 w2 fg] mon-ww-ileq[of w22 w2 fg] have MON2N-LEQ: mon-ww fg w21 \subseteq mon-ww fg w2 mon-ww fg w22 \subseteq mon-ww fg w2} by auto
from REVSPILT(2) show \text{thesis proof} (cases rule: set-unplsm-dist-cases[case-names left't right'])
\[ \text{case left': Both nodes are reached from the same thread} \]
\[ \text{have ATUV: atUV U V \{\#sq'\#\} + \text{csp-q} using right C2'FMT R-STACKS(2,3) left'(1)} \]
\[ \text{by (metis R-CONV(1) add-mset-add-single atUV-union atU-\text{add-mset union-commute})} \]
from Cons.hyps[OF - R-CONV(3) ATUV] cil-length[OF REVSPILT(3)] cil-length[OF LEISPILT(1)] R-CONV(2) obtain MI Me where IHAPP: (entry fg q, MI, Me) \in \text{RUV-cs fg U V MI} \subseteq \text{\text{\#entry fg q\#}} \text{I} \text{by simp}
from REVSPILT(1) S-ENTRY-PAT[of \{\#q\#, \text{simpified}] have S-ENTRY': (v, mon-w fg w, \{\#q\#\}) \in \text{S-cs fg I} \text{by simp}
\text{have MON-COND: (mon-n fg u \cup mon-fg p) \cap (MI \cup Me) = \{}} \text{proof –}
\text{from R-CONV(2) have mon-fw fg w21 = mon-loc fg w21 \cup mon-env fg w21 by simp}
\text{with IHAPP(2,3) MON2N-LEQ(1) MON-PU MON2LEQ show \{\text{thesis by blast}} \]
\text{qed}
from RUV-spawn[OF FS-FMT(2) FS-FMT(4) S-ENTRY - IHAPP(1)] MON-COND] have (u, mon-fg p \cup mon-w fg w, MI \cup Me) \in \text{RUV-cs fg U V by simp}
\text{moreover have mon-fg p \cup mon-w fg w \subseteq mon-loc fg (ee\#ww) using}
\text{FS-FMT(1) by auto}
\text{moreover have MI \cup Me \subseteq mon-env fg (ee\#ww) using IHAPP(2,3) R-CONV(2) MON2N-LEQ(1) MON2LEQ by (auto simp add: mon-ww-of-loc-rem)}
ultimately show \( \text{thesis by blast} \)

next

case right' — The nodes are reached from different threads

from R-STACKS(2,3) have ATUV: atU U (add-set 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''' ∈# ceh w22'≤w22 ce22e' ≤# ce22' atU V ce22e' ([#|sq''|#], w22', ce22e') ∈trcl
(ntr fg) by (auto simp add: valid-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 the soundness of the RU-constraint system, we get the corresponding entries

from RU-sound[OF R-CONV(3) ATUV(1)] obtain Ml Me h where RU:
(entry fg q, Ml, Me, h) ∈ RU-cs fg U Ml ⊆ mon-loc fg w21 Me ⊆ mon-env fg
w21 h ≤ αah (map (canl fg) w21) by blast
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 w22' Me' ⊆ mon-env fg w22' h' ≤ αah (map (canl fg) w22') by auto
from S-ENTRY-PAT[of {#q##}+{#q##}, simplified] REVSPLIT'(1) REVSPLIT'(1) have S-ENTRY: (v, mon-w fg w, {#q##} + {#q##}) ∈ S-cs fg (2::nat)
by (simp add: numerals)

have (u, mon-fg p ∪ mon-w fg w, Ml ∪ Me ∪ Ml' ∪ Me') ∈ RU-cs fg U V
proof (rule RU-splitt-ee[OF FS-FMT(2,4) S-ENTRY : RU(1) RV(1)])
from MON-PU MON2-LEQ MON2N-LEQ R-CONV(2) R-CONV'(2)
mon-ww-ileq[OF REVSPLIT'(2), of fg] RU(2,3) RV(2,3) show (mon-n fg u ∪
mon fg p) ∩ (Ml ∪ Me ∪ Ml' ∪ Me') = {} by (simp add: mon-ww-of-leq) blast
next
from ab-interleavable1[OF REVSPLIT(3)] have αah (map (canl fg) w21)
[∗] αah (map (can fg) w22)

thus h [∗] h'

proof (erule_tac ah-leq-il)
note RU(4)
also have map (canl fg) w21 ≤ map (can fg) w21 using R-CONV(2)
by (simp add: can-canl)

hence αah (map (canl fg) w21) ≤ αah (map (can fg) w21) by (rule
ahh-ileq)

finally show h ≤ αah (map (can fg) w21) .

106
next
note RV(4)
also have map (αn fg) w22′ ≼ map (αn fg) w22 using R-CONV′(2)
REVSPPLIT′(2) by (simp add: αn-αn[symmetric] le-list-map map-map[symmetric]
del: map-map)
hence αah (map (αn fg) w22′) ≤ αah (map (αn fg) w22) by (rule
αah-ileq)
finally show h′ ≤ αah (map (αn fg) w22).
qed
qed (simp)
moreover have mon fg p ∪ mon-w fg w ⊆ mon-loc fg (ee#ww) using
FS-FMT(1) by auto
moreover have MI ∪ Me ∪ MI′∪ Me′ ⊆ mon-env fg (ee#ww) using RV(2,3)
RU(2,3) mon-ww-ileq[OF REVSPPLIT′(2), of fg] MON2N-LEQ R-CONV(2) R-CONV′(2)
MON2-L EQ 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] br(1) DECOMP-LOC(1) obtain
MI Me h where RU: (v, MI, Me, h) ∈ RU-cs fg U MI ⊆ mon-loc fg w1 Me
⊆ mon-env fg w1 h ≤ αah (map (αn fg) w1) by auto
obtain MI′ Me′ h′ q′ where RV: [entry fg q′] ∈# (entry fg q′, MI′, Me′, h′) ∈ RU-cs fg V MI′ ⊆ 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

obtain q′ w2′ c2′ where REVSPPLIT: [entry fg q′] ∈# (entry fg q′, w2′ w2′ ⊆ c2′ ⊆ c2′ aut V c2′ ([c2′#entry fg q′#w2′]), c2′ ∈ trcl (nttr fg)
using ntr-rereverse-split-atU[OF - br(2) LESP LIT′(4)] ntrp-valid-preserve-s[OF
SPLIT(1), simplified] CHFMT by (simp add: valid-unconc) blast
from ntrp,gt2trp[where: e=({}, simplified, OF REVSPPLIT′(5)] obtain s2i′ c2i′ w2w′ where R-CONV: c2i′=add-mset s2i′ c2i′ w2′ w2′=map le-rem-s w2′
(((entry fg q′), [], w2w′), w2w′, s2i′, c2i′) ∈ trcl (nttrp fg).
from RU-sound[OF R-CONV(3), of V] REVSPPLIT′(4) R-CONV′(1) obtain
MI′ Me′ h′ where RV: (entry fg q′, MI′, Me′, h′) ∈ RU-cs fg V MI′ ⊆ mon-loc fg
ww2′ Me′ ⊆ mon-env fg ww2′ h′ ≤ αah (map (αn 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 REVSPPLIT′(2), of fg] R-CONV′(2) by (auto simp
add: mon-ww-of-le-rem)
moreover have αah (map (αn fg) ww2′) ≤ αah (map (αn fg) w2) using
REVSPPLIT′(2) R-CONV′(2) by (auto simp add: αn-αn[symmetric] le-list-map
map-map[symmetric]) simp del: map-map intro: αah-ileq del: predicate2I
ultimately show thesis using that REVSPPLIT(1) by (blast intro: order-trans)
qed
from S-ENTRY-PAT[of {}, simplified] RV(1) have S-ENTRY: (v, mon-w fg w, {}) ∈ S-cs fg I by simp

107
have \((u, \text{mon } fg \ p \cup \text{mon-w } fg \ w \cup \text{MI}, Me \cup \text{MI'} \cup Me') \in \text{RUV-cs } fg \ U \ V\)

**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 \((\text{mon-n } fg \ u \cup \text{mon } fg \ p) \cap (Me \cup \text{MI'} \cup Me') = \{\}\) by blast

next

from ah-interleavable1[OF LESPLIT(1)] have \(\alpha ah \ (\text{map } (\text{onl } fg) \ w1)\) [\(*\]

\(\alpha ah\ (\text{map } (\text{onl } fg) \ w2)\) by simp

thus \(h \ [\*]\) \(h'\) using RU(4) RV(5) by (auto elim: ah-leq-il)

qed (simp)

moreover have \(\text{mon } fg \ p \cup \text{mon-w } fg \ w \cup \text{MI} \subseteq \text{mon-loc } fg \ (ee \# \ ww)\) using FS-FMT(1) MON1-LEQ RU(2) by (simp) blast

moreover have \(Me \cup \text{MI'} \cup Me' \subseteq \text{mon-env } fg \ (ee \# \ ww)\) using MON1-LEQ MON2-LEQ RU(3) RV(3,4) by (simp) blast

ultimately show \(\text{thesis}\) 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 \(\text{MI } Me \ h\) where \(\text{RV: } (v, \text{MI}, Me, h) \in \text{RU-cs } fg \ V \ \text{MI} \subseteq \text{mon-loc } fg \ w1 \ Me \ \subseteq \text{mon-env } fg \ w1 \ h \ \leq ah (\text{map } (\text{onl } fg) \ w1)\) by auto

obtain \(\text{MI'} \ Me' \ h' \ q'\) where \(\text{RU: } (\text{entry } fg \ q') \in \# \ (\text{entry } fg \ q', \text{MI'}, Me', h') \in \text{RU-cs } fg \ U \ \text{MI'} \subseteq \text{mon-ww } fg \ w2 \ Me' \ \subseteq \text{mon-ww } fg \ w2 \ h' \leq ah (\text{map } (\text{onl } fg) \ w2)\)

proof —

— We have to extract the interesting thread from the spawned threads in order to get an entry in \(RU \ fg \ V\)

obtain \(q' \ w2' \ c2i' \ h' \) where REVSPILT: \([\text{entry } fg \ q'] \in \# \ ch \ w2' \subseteq w2 \ c2i' \ \subseteq \# \ c2' \ atU \ U \ c2' (\{\# | \text{entry } fg \ q' \} \w2', \{\# | \text{entry } fg \ q' \} \w2') \in \text{trcl } (\text{ntr } fg)\)

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.trgt2trp[where \(\equiv \{\#\},\) simplified, OF REVSPILT(5)] obtain \(s2i' \ c2ic' \ w2w' \) where R-CONV: \(c2'i' = \text{add-mset } s2i' \ c2ie' \ w2w' = \text{map le-rem-s } w2w' (((\text{entry } fg \ q'), \{\#\}), \ w2w', \ s2i', \ c2ie') \in \text{trcl } (\text{ntrp } fg)\).

from RU-sound[OF R-CONV(3), of U] REVSPILT(4) R-CONV(1) obtain \(\text{MI' } Me' \ h' \) where \(\text{RU: } (\text{entry } fg \ q', \text{MI'}, Me', h') \in \text{RU-cs } fg \ U \ \text{MI'} \subseteq \text{mon-loc } fg \ w2w' \ Me' \ \subseteq \text{mon-env } fg \ w2w' h' \leq ah (\text{map } (\text{onl } fg) \ w2w')\) by auto

moreover have \(\text{mon-loc } fg \ w2w' \ \subseteq \text{mon-ww } fg \ w2w' \text{mon-ww } fg \ w2w' \ \subseteq \text{mon-ww } fg \ w2\)

moreover have \(\alpha ah \ (\text{map } (\text{onl } fg) \ w2w') \leq ah \ (\text{map } (\text{onl } fg) \ w2)\) using REVSPILT(2) R-CONV(2) by (auto simp add: \alpha ah-ileq del: predicate2I)

ultimately show \(\text{thesis}\) using that REVSPILT(1) by (blast intro: order-trans)

qed

from S-ENTRY-PAT[of \{\# q'\}, simplified] RU(1) have S-ENTRY: \((v, \text{mon-w } fg \ w, \{\# q'\}) \in S-cs \ fg \ t\) by simp

have \((u, \text{mon } fg \ p \cup \text{mon-w } fg \ w \cup \text{MI}, Me \cup \text{MI'} \cup Me') \in \text{RUV-cs } fg \ U \ V\)

**proof** (rule RUV-split-el)(OF FS-FMT(2,4) S-ENTRY - RU(1) RU(2))]

from MON-PU MON1-LEQ MON2-LEQ RV(3) RU(3,4) show \((\text{mon-n } fg\))
\[ u \cup \text{mon fg p} \cap (\text{Me} \cup \text{ML'} \cup \text{Me'}) = \{\} \text{ by blast} \]

next

from \text{ah-interleavable}! [\text{OF LSEQ\textunderscore I}] have \text{\alpha ah} (\text{map (ahl fg w1) [\(\ast\)]})
\text{\alpha hh} (\text{map (ahl fg w2) by simp})

thus \text{\h h' using RUV(4) RUV(5) by (auto elim: ah-req-\text{il})}

\text{qed (simp)}

moreover have \text{mon fg p} \cap \text{mon-w fg w} \cup \text{ML} \subseteq \text{mon-loc fg (ee \# ww) using}
\text{FS-FMT(1) MON1-LEQ RUV(2) by (simp) blast}

moreover have \text{Me} \cup \text{ML'} \cup \text{Me'} \subseteq \text{mon-env fg (ee \# ww) using MON1-LEQ}
\text{MON2-LEQ RUV(3) RUV(3,4) by (simp) blast}

ultimately show \text{?thesis by blast}

\text{qed}

\text{lemma (in flowgraph)} \text{ RUV-precise: (u,ML,Me)\in RUV-cs fg U V}

\[ \exists w \ s' \ c' \ (\{[[u],\{\#\}],w,(s',c')\}\in\text{trcl (ntrp fg)} \land \text{atUV U V (\{\#\} + c') \land \text{mon-loc fg w} = \text{ML} \land \text{mon-env fg w} = \text{Me}} \text{ by blast} \]

from \text{S-precise-ntrp[OF RUV-call(3,2,1), simplified]} obtain \text{w} \text{ ch where FS:}
\[ (\{[[u],\{\#\}],w,(s',c')\}\in\text{trcl (ntrp fg)} \text{ atUV U V (\{\#\} + c') \text{ mon-loc fg w} = \text{ML} \text{ mon-env fg w} = \text{Me}} \text{ by blast} \]

\text{note FS(1)}

also have \( (\{[[u],\{\#\}],w,(s',c')\}\in\text{trcl (ntrp fg)} \]

\text{using ntrp-add-context[OF ntrp-stack-comp[OF IH(1), of \{[[u],\{\#\}],[ch,simplified]
\text{FS(5) IH(4) RUV-call.hyps(6) mon-n-same-proc[OF edges-part[OF RUV-call.hyps(1)]]

\text{by simp}

finally have \( (\{[[u],\{\#\}],w,(s',c')\}\in\text{trcl (ntrp fg)} \)

moreover from \text{IH(2) have atUV U V (\{\#\} + (c'+ch)) by auto}

moreover have \text{mon-loc fg (LOC (LCall p \# w) \# ww) = mon fg p \cup M \cup ML}

using \text{IH(3) FS(3) by auto}

moreover have \text{mon-env fg (LOC (LCall p \# w) \# ww) = Me using IH(4)}

by auto

ultimately show \text{?case by blast}

next

\text{case (RUV-spawn u p u' v M P q Ml Me) then obtain w s' c' where IH:}
\[ (\{[[entry fg q],[\{\#\}],w,(s',c')\}\in\text{trcl (ntrp fg)} \text{ atUV U V (\{\#\} + c') \text{ mon-loc fg w} = \text{ML} \text{ mon-env fg w} = \text{Me}} \text{ by blast} \]

from \text{S-precise-ntrp[OF RUV-spawn(3,2,1), simplified]} \text{ mset-size1elem[OF - RUV-spawn(4)]}

obtain \text{w} \text{ c' where FS:}
\[ (\{[[u],\{\#\}],w,(s',c')\}\in\text{trcl (ntrp fg)} \text{ P=} (\{\#\} + \text{che}) \in \text{ntrp fg P=} (\{\#\} + \text{che}) \text{ M} = \text{mon-w fg w} \text{ mon-n fg v} = \text{mon fg p \cup M \cup ML} \]

by auto

ultimately show \text{?thesis by blast}

109
moreover
have \(((v, u'), \text{ che} + \{\# \text{entry fg q}\#\})\), map \(\text{ENV} (\text{map le-rem-s w})\), \(((v, u'), \text{che} + (\{\#s'\#\} + c'))\)\)\in trcl \((\text{ntrp fg})\)

using \(\text{ntrp2ntrp[OF gtrp2gtr[OF IH(1)], of [v, u'] che] IH(3,4)}\) \(\text{RUv-spawn(7)}\)
\(\text{FS(4,5)}\) \(\text{mon-n-same-proc[OF edges-part[OF RUV-spawn(1)]]}\)

by \((\text{auto simp add: mon-c-unconc mon-ww-of-le-rem})\)

ultimately have \(((\{v\}, \{\#\})\), \(\text{LOC (LCall p # w)}\#\map \text{ENV} (\text{map le-rem-s w})\), \(((v, u'), \text{che} + (\{\#s'\#\} + c'))\)\in trcl \((\text{ntrp fg})\)\) by \((\text{auto simp add: union-ac})\)

moreover have \(\text{atUV U V} (\{\# [v, u']\#\} + (\text{che} + (\{\#s'\#\} + c'))\)\) using \(\text{IH(2)}\)
by auto

moreover have \(\text{mon-loc fg} (\text{LOC (LCall p # w)}\#\map \text{ENV} (\text{map le-rem-s w})) = \text{mon fg p} \cup \text{M}\) using \(\text{FS(3)}\) by \((\text{simp del: map-map})\)

moreover have \(\text{mon-env fg} (\text{LOC (LCall p # w)}\#\map \text{ENV} (\text{map le-rem-s w})) = \text{Ml} \cup \text{Me}\) using \(\text{IH(3,4)}\) by \((\text{auto simp add: mon-ww-of-le-rem simp del: map-map})\)

ultimately show \(\text{?case by blast}\)

next
\text{case (RU-\text{split-le} u p u' v M P q Ml Me h Ml' Me' h')}

— Get paths from precision results

from \(\text{S-precise-ntrp[OF RUV-split-le(3,2,1), simplified]}\) \(\text{mset-size1elem[OF -RUv-split-le(4)]}\) obtain \(\text{ww1 s1' c1'}\) where \(\text{P1: (([v], \{\#\}), ww1, s1', c1' \in trcl (ntrp fg)}\) \(\text{atUV U}\) \((\{\#s'\#\} + c')\) \(\text{mon-loc fg}\) \(\text{ww1 = Ml}\) \(\text{mon-env fg}\) \(\text{ww1 = Me cah (map (cnil fg) ww1)}\) = \(h\) by \(\text{blast}\)

from \(\text{RU-precise[OF RUV-split-le(5)]}\) obtain \(\text{ww2 s2' c2'}\) where \(\text{P2: ([entry fg q], \{\#\}), ww2, s2', c2'}\in trcl (ntrp fg)\) \(\text{atUV V}\) \((\{\#s'\#\} + c')\) \(\text{mon-loc fg}\) \(\text{ww2 = Ml'\ mon-env fg}\) \(\text{ww2 = Me' cah (map (cnil fg) ww2)}\) = \(h'\) by \(\text{blast}\)

— Get combined path from the acquisition history interleavability, need to remap loc/evn-steps in second path

from \(\text{P2(5)}\) have \(\text{cah (map (cnil fg) (map ENV (map le-rem-s w)) = h')}\) by \(\text{simp add: mset-mset-op assoc}\)

with \(\text{P1(5) RUV-split-le(8)}\) obtain \(\text{ww where IL: ww \in ww1@\text{cnil fg}(map ENV (map le-rem-s w))}\) using \(\text{ah-interleaveable2 by (force)}\)

— Use the \(\text{ntrp-unsplit}\)-theorem to combine the executions

from \(\text{ntrp-unsplit[where ca\{-\{\#\},OF IL P1(1) gtrp2gtr[OF P2(1)], simplified]}\)

\text{have \(((\{v\}, \{\# \text{entry fg q}\#\}), \text{ww, s1', c1' + (\{\#s'\#\} + c2')}\) \in trcl (ntrp fg)\)

using \(\text{FS(4,5) RUV-split-le(7)}\)

by \((\text{auto simp add: mon-c-unconc mon-ww-of-le-rem P2(3,4)})\)

from \(\text{ntrp-add-context[OF ntrp-stack-comp[OF this, of [u'], of che]}\) have \(((v) \at [u'], \{\# \text{entry fg q}\#\} + \text{che}, \text{ww, s1' @ [u'], c1' + (\{\#s'\#\} + c2') + \text{che}})\) \(\in \text{trcl (ntrp fg)}\)

using \(\text{mon-n-same-proc[OF edges-part[OF RUV-split-le(1)]]}\) \(\text{mon-loc-cil[OF IL, of fg]}\) \(\text{mon-env-cil[OF IL, of fg]}\) \(\text{FS(4,5) RUV-split-le(7)}\) by \((\text{auto simp add: mon-c-unconc P1(3,4) P2(3,4) mon-ww-of-le-rem simp del: map-map})\)

with \(\text{FS(1)}\) have \(((\{v\}, \{\#\}), \text{LOC (LCall p # w)}\#\map \text{ww, (s1' @ [u'], c1' + (\{\#s'\#\} + c2') + \text{che}})\in \text{trcl (ntrp fg)}\) by \(\text{simp}\)
moreover have $atUVUV\{\#s1^\prime @ [u']\#\}+(c1^\prime +\{\#s2^\prime\#\} + c2^\prime + che))$

using $P1(2)$ $P2(2)$ by auto

moreover have $mon-loc fg\ (LOC\ (LCall\ p\ #\ w)\ #\ ww) = mon\ fg\ p\ M\ U\ Ml$

using $FS(3)\ P1(3)$ $mon-loc-cil[OF\ IL,\ of\ fg]$ by (auto simp del: map-map)

moreover have $mon-env fg\ (LOC\ (LCall\ p\ #\ w)\ #\ ww) = Me\ U\ Ml'\ Me'$

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 $\textit{case by blast}$

next
case $RU-precise\ split\ IL\ p\ u'\ v\ M\ P\ q\ Ml\ Me\ h\ Ml'\ Me'\ h'$ — This is the symmetric case to $RU-precise\ split\ IL$, it is proved completely analogously, just need to swap $U$ and $V$

— Get paths from precision results

from $RU-precise\ split\ el(3,2,1)$, simplified

mset-size1elem[OF $RU-precise\ split\ el(4)$] obtain $w\ che$ where

FS: $(((u, \{\#\}, LOC\ (LCall\ p\ #\ w), [v, a'], \{\#\}entry\ fg\ q\#\} + che) \in ntrp\ fg\ P=\{\#q\#\}\ M = mon-w\ fg\ w\ mon-n\ q\ v = mon\ fg\ p\ mon-c\ (\{\#\}entry\ fg\ q\#\} + che) = \{} \by (auto\ elim: mset-le-addE)

from $RU-precise[OF\ RU-precise\ split\ el(5)]$ obtain $ww1\ s1^\prime\ c1'$ where $P1: (([v], \{\#\}, ww1, s1', c1', c\#' ) \in trcl\ (ntrp\ fg)\ atU\ U\ ([\#s1^\prime\#\} + c1')\ mon-loc\ fg\ ww1 = Ml\ mon-env\ fg\ ww1 = Me\ \alpha\ ah\ (map\ (\alpha\ nl\ fg)\ ww1) = h'$ by blast

— Get combined path from the acquisition history interleavability, need to remap loc/env-steps in second path

from $P2(5)$ have $\alpha\ ah\ (map\ (\alpha\ nl\ fg)\ (map\ ENV\ (map\ le-rem-s\ wu2))) = h'$ by (simp add: mon-cnvl $\alpha$-assoc)

with $P1(5)\ RU-precise\ split\ el(8)$ obtain $ww\ where\ IL: \ww\in\ww1\otimes\alpha\nl\fg\ (map\ ENV\ (map\ le-rem-s\ wu2))$ using $ah$-interleave2 $\textit{by (force)}$

— Use the $\textit{ntrp-unsplit}$-theorem to combine the executions

from $ntrp-unsplit[where\ ca=\{\#\},\ OF\ IL\ P1(1)$ $\otimes\trp\operatorname{tr}\ [OF\ P2(1)],\ simplified]$ have $(([v], \{\#\}entry\ fg\ q\#\},\ ww, s1', c1'+\{\#s2^\prime\#\} + c\#) \in trcl\ (ntrp\ fg)$

using $FS(4, 5)$ $RU-precise\ split\ el(7)$

by (auto simp add: mon-c-uncon\ mon-ww-of-le-rem $P2(3,4)$)

from $ntrp-unsplit[where\ ca=\{\#\},\ OF\ ntrp\ stack\ comp[OF\ this,\ of\ [u'],\ of\ che]\ have\ (((v)\ @\ [u'],\ \{\#\}entry\ fg\ q\#\} + che),\ $ww, s1' \in [u'], c1'+\{\#s2^\prime\} + c\#\} + che) \in trcl\ (ntrp\ fg)$

using $\textit{mon-n-same-proc[OF\ edges-part[OF\ RU-precise\ split\ el(1)]]}$ $\textit{mon-loc-cil[OF\ IL,\ of\ fg]}$ $\textit{mon-env-cil[OF\ IL,\ of\ fg]}$ $\textit{FS(4, 5)\ RU-precise\ split\ el(7)}$ by (auto simp add: mon-c-uncon $P1(3,4)$ $P2(3,4)$ $\textit{mon-ww-of-le-rem\ simp\ del: map-map}$)

with $FS(1)$ have $(([v], \{\#\}, LOC\ (LCall\ p\ #\ w)\ #\ ww, (s1' \in [u'], c1'+\{\#s2^\prime\} + c\#\} + che) \in trcl\ (ntrp\ fg)\ by\ simp$

moreover have $atUVUV\{\#s1^\prime @ [u']\#}\ +(c1' +\{\#s2^\prime\} + c\#\} + che)$

using $P1(2)\ P2(2)$ by auto

moreover have $mon-loc\ fg\ (LOC\ (LCall\ p\ #\ w)\ #\ ww) = mon\ fg\ p\ M\ U\ Ml$

using $FS(3)\ P1(3)$ $mon-loc-cil[OF\ IL,\ of\ fg]$ by (auto simp del: map-map)

moreover have $mon-env\ fg\ (LOC\ (LCall\ p\ #\ w)\ #\ ww) = Me\ U\ Ml'\ Me'$

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-u 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 V ((\{\#s1'\#\} + c1') mon-loc fg ww1 = Ml mon-env fg ww1 = Me oah (map (env 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' oah (map (env fg) ww2) = h' by blast

  — Get interleaved paths, project away loc/env information first

  from P1(5) P2(5) have oah (map (on fg) (map le-rem-s ww1)) = h oah (map (on fg) (map le-rem-s ww2)) = h' by (auto simp add: on-onl o-assoc)

  with RUV-split-ee(8) obtain ww where IL: ww ∈ (map le-rem-s ww1) ⊗_onl fg (map le-rem-s ww2) using ah-interleaveable2 by (force simp del: map-map)

  — Use the ntr-absplit 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 (ntrp 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
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:
\[ \exists w \ c'. (\{\#entry fg (main fg)\}\#), w, c') \in trcl (refpoint fg) \land atUV U V c' \]
\[ \longleftrightarrow (\exists Ml Me. (entry fg (main fg), Ml, Me) \in RUV-cs fg U V) \]

— 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-mset s' ce' ww = map le-rem-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-cs fg U V by blast
from RUV-precise[OF C] obtain \( \text{wwl} \ s' \ c' \) where \( P: ((\text{entry fg (main fg)}), \{\#\}), \text{wwl}, s', c') \in \text{trcl (ntrp fg) atUV U V (\{\#s'\#\} + c')} \) by blast

from gtrp2gtr[OF P(1)] have \((\{\# [\text{entry fg (main fg)}]\} \#), \map \text{le-rem-s} \text{wwl}, \{\#s'\#\} + c') \in \text{trcl (ntr fg)} \) by (auto)

from ntr-is-tr[OF this] P(2) have \( \exists \ w \ c'. ((\{\# [\text{entry fg (main fg)}]\} \#), \ w, c') \in \text{trcl (tr fg) \land atUV U V c'} \) by blast

thus \( \text{lhs by (simp add: refpoint-eq)} \)

qed

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


