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

Peter Lammich
Markus Müller-Olm

Institut für Informatik, Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster
peter.lammich@uni-muenster.de and mmo@math.uni-muenster.de

February 23, 2021

Abstract

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

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

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

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

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

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

2 Monitor Consistent Interleaving

theory ConsInterleave imports Interleave Misc begin

The monitor consistent interleaving operator is defined on two lists of arbitrary elements, provided an abstraction function \(\alpha\) 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: \(\text{mon-pl} \ (\text{map} \ \alpha \ w)\) 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. \(\text{fst} \ e \cup \text{snd} \ e\)) w)

lemma mon-pl-empty[simp]: mon-pl [] = {\}

\textsuperscript{1}Reentrant locks can always be simulated by non-reentrant ones, at the cost of a worst-case exponential blowup of the program size
by (unfold mon-pl-def, auto)

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

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

**Lemma** mon-pl-ileq: \( w \preceq w' \implies \text{mon-pl } w \subseteq \text{mon-pl } w' \)
by (induct rule: less-eq-list-induct) auto

**Lemma** mon-pl-set: \( \text{mon-pl } w = \bigcup \left\{ \text{fst } e \cup \text{snd } e \mid e \in \text{set } w \right\} \)
by (auto simp add: mon-pl-def foldl-set) blast+

**fun**

\[
\text{cil} :: 'a list \Rightarrow ('a \Rightarrow ('m set \times 'm set)) \Rightarrow 'a list \Rightarrow 'a list set
\]

- 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

\[
| w \odot \alpha \cdot [\cdot [w] = \{w\}
| w \odot \alpha ] = \{w\}
\]
- If both words are not empty, we can take the first step of one word, interleave the rest with the other word and then append the first step to all result set elements, provided it does not allocate a monitor that is used by the other word

\[
| e1 \# w1 \odot \alpha \cdot e2 \# w2 = \{
  \text{if } \text{fst}(\alpha \cdot e1) \cap \text{mon-pl}(\text{map } \alpha (e2 \# w2)) = \{\} \text{ then } e1 \cdot (w1 \odot \alpha \cdot e2 \# w2)
\]
\[
\text{else } \{}
\}
| \}
  \text{if } \text{fst}(\alpha \cdot e2) \cap \text{mon-pl}(\text{map } \alpha (e1 \# w1)) = \{\} \text{ then } e2 \cdot (e1 \# w1 \odot \alpha \cdot w2)
\]
\[
\text{else } \{}
\}
\]

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]:

\[
| w \in e1 \# w1 \odot \alpha \cdot e2 \# w2;
| \}
| \}
\]

- 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

\[
| !!w' \cdot [w = e1 \# w' \wedge w' \in (w1 \odot \alpha \cdot e2 \# w2);
  \text{fst}(\alpha \cdot e1) \cap \text{mon-pl}(\text{map } \alpha (e2 \# w2)) = \{\} ] \implies P;
| \}
| \}
\]

- 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

\[
| !!w' \cdot [w = e2 \# w' \wedge w' \in (e1 \# w1 \odot \alpha \cdot w2);
  \text{fst}(\alpha \cdot e2) \cap \text{mon-pl}(\text{map } \alpha (e1 \# w1)) = \{\} ] \implies P
\]

by (auto elim: list-set-cons-cases split: if-split-asn)

**Lemma** cil-cases [cases set, case-names both-empty left-empty right-empty app-left app-right]:
\[
\begin{align*}
\text{lemma} & \quad \text{cil-induct-fix} \\
\text{lemma} & \quad \text{cil-induct} \\
\text{qed} & \\
\text{simp} & \\
\text{next} & \\
\text{proof} & \begin{cases}
\text{case-1} & \text{by simp next} \\
\text{case-2} & \text{by simp next} \\
\text{var-3} & \text{by simp next}
\end{cases}
\end{align*}
\]

\[
\text{proof} (\text{induct } wa \alpha wb \text{ rule: cil.induct}) \\
\text{case-1} & \text{by simp next} \\
\text{case-2} & \text{by simp next} \\
\text{case-3} & \text{by simp next}
\]

\[
\text{lemma} \quad \text{cil-induct}[\text{case-names both-empty left-empty right-empty append}]: \\
\begin{align*}
\forall \alpha. & \quad P \alpha [] [] \\
\forall \alpha \text{ ad ac}. & \quad P \alpha [] (\text{ad} \# \text{ ac}) \\
\forall \alpha \text{ z aa}. & \quad P \alpha (\text{z} \# \text{ aa}) [] \\
\forall \alpha \text{ e1 w1 e2 w2}. & \quad \\
\end{align*}
\]

\[
\begin{align*}
\forall \alpha \text{ e1 w1 e2 w2}. & \quad \\
\end{align*}
\]

\[
\begin{align*}
\text{apply} & \quad (\text{induct } wa \alpha wb \text{ rule: cil.induct}) \\
\text{apply} & \quad (\text{case-tac } w) \\
\text{apply} & \quad \text{auto} \\
\text{done}
\end{align*}
\]

\[
\begin{align*}
\text{lemma} \quad \text{cil-induct-fixa}: \\
\begin{align*}
\forall \alpha. & \quad P \alpha [] [] \\
\forall \alpha \text{ ad ac}. & \quad P \alpha [] (\text{ad} \# \text{ ac}) \\
\forall \alpha \text{ z aa}. & \quad P \alpha (\text{z} \# \text{ aa}) [] \\
\forall \alpha \text{ e1 w1 e2 w2}. & \quad \\
\text{apply} & \quad (\text{induct } v \alpha w \text{ rule: cil.induct}) \\
\text{apply} & \quad (\text{case-tac } w) \\
\text{apply} & \quad \text{auto} \\
\text{done}
\end{align*}
\]

7
lemma cil-induct-fixα(case-names both-empty left-empty right-empty append): 
\[ P \alpha [] [] \]
\[ \\wedge \alpha. P \alpha [] (\alpha \neq \emptyset) ; \]
\[ \wedge \alpha. P \alpha (\emptyset \neq \emptyset) [] ; \]
\[ \Rightarrow \alpha \triangleq \emptyset \wedge P \alpha w1 (e2 \neq w2) ; \]
\[ \Rightarrow \alpha \triangleq \emptyset \wedge P \alpha (e1 \neq w1) (e2 \neq w2) \]
\[ \Rightarrow \alpha \triangleq \emptyset \wedge P \alpha wa wb \]
apply (induct wa α wb rule: cil.induct)
apply (case-tac w)
apply auto
done

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

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

lemma cil-cons-cases[cases set, case-names left right]: 
\[ \Rightarrow \alpha \triangledown \emptyset \]
\[ \Rightarrow \alpha \triangledown w \]
\[ \Rightarrow \alpha \triangledown w' \]
\[ \Rightarrow \alpha \triangledown w' \]
\[ \Rightarrow \alpha \triangledown w' \]
\[ \Rightarrow \alpha \triangledown w' \]
by (induct w) (auto intro!: cil-contains-empty elim: cil-cons-cases)

lemma cil-set-induct[induct set, case-names empty left right]: 
\[ \Rightarrow \alpha \triangledown \emptyset \]
\[ \Rightarrow \alpha \triangledown w \]
\[ \Rightarrow \alpha \triangledown w' \]
\[ \Rightarrow \alpha \triangledown w' \]
\[ \Rightarrow \alpha \triangledown w' \]
by (induct w) (auto intro!: cil-contains-empty elim: cil-cons-cases)

lemma cil-cons1[cases set, case-names empty left right]: 
\[ \Rightarrow \alpha \triangledown \emptyset \]
\[ \Rightarrow \alpha \triangledown w \]
\[ \Rightarrow \alpha \triangledown w' \]
\[ \Rightarrow \alpha \triangledown w' \]
\[ \Rightarrow \alpha \triangledown w' \]
by (cases wb) auto
lemma \( \text{cil-cons2} \): \[ w \in w_1 \otimes \alpha w_2 \implies e \# w \in w_1 \otimes \alpha e \# w_2 \]

by (cases \( w_1 \)) auto

2.2 Properties of consistent interleaving

— Consistent interleaving is a restriction of interleaving

lemma \( \text{cil-subset-il} \): \( w \otimes \alpha w' \subseteq w \otimes w' \)

apply (induct \( w \alpha w' \) rule: \( \text{cil-induct} \))
apply simp-all
apply safe
apply auto
done

lemma \( \text{cil-subset-il} \)′: \( w \in w_1 \otimes \alpha w_2 \implies w \in w_1 \otimes w_2 \)

using \( \text{cil-subset-il} \) by (auto)

— Consistent interleaving preserves the set of letters of both operands

lemma \( \text{cil-set} \): \( w \in w_1 \otimes \alpha w_2 \implies \text{set } w = \text{set } w_1 \cup \text{set } w_2 \)

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

corollary \( \text{cil-mon-pl} \): \( w \in w_1 \otimes \alpha w_2 \)

\implies \text{mon-pl (map } \alpha w) = \text{mon-pl (map } \alpha w_1) \cup \text{mon-pl (map } \alpha w_2) \)

by (subst \( \text{mon-pl-unconc[symmetric]} \)) (simp add: \( \text{mon-pl-set cil-set} \), blast 20)

— Consistent interleaving preserves the length of both operands

lemma \( \text{cil-length} \)\([\text{rule-format}]\): \( \forall w \in w_1 \otimes \alpha w_2. \text{length } w = \text{length } w_1 + \text{length } w_2 \)

by (induct rule: \( \text{cil-induct} \)) auto

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

lemma \( \text{cil-ileq} \): \( w \in w_1 \otimes \alpha w_2 \implies w_1 \preceq w \land w_2 \preceq w \)

by (intro conjI \( \text{cil-subset-il'} \) \( \text{ileq-interleave} \))

— Consistent interleaving is commutative and associative

lemma \( \text{cil-commute} \): \( w \otimes \alpha w' = w' \otimes \alpha w \)

by (induct rule: \( \text{cil-induct} \)) auto

lemma \( \text{cil-assoc1} \): \( \forall w_1 w_2 w_3. [ w \in w_1 \otimes \alpha w_2 ; w \in w_1 \otimes \alpha w_3 ] \)

\implies \exists wr. w \in w_1 \otimes \alpha wr \land wr \in w_2 \otimes \alpha w_3 \)

proof (induct \( w \) rule: \( \text{length-compl-induct} \))
case Nil thus ?case by auto

next
case (Cons \( e \) \( w \)) from Cons.prems(1) show ?case
proof (cases rule: \( \text{cil-cons-cases} \))
case (left \( w' \)) with Cons.prems(2) have \( e \# w' \in w_1 \otimes \alpha w_2 \) by simp
thus ?thesis
proof (cases rule: \( \text{cil-cons-cases[case-names left' right']} \))
case (left' \( w' \))
from Cons.hyps(OF - left(2) left'(2)) obtain wr where \( \text{IHAPP} : w \in w_1 \otimes \alpha w' \)
wr wr \in w_2 \otimes \alpha w_3 \) by blast
have \( e \# w \in e \# w_1 \otimes \alpha w' \) w proof (rule \( \text{cil-cons1[OF IHAPP(1)]} \))
from left left' \( \text{cil-mon-pl[OF IHAPP(2)]} \) show \( \text{fst (} \alpha \ e \) \( \cap \text{mon-pl (map } \alpha \)

9
proof (cil-map lemma — Parts of the abstraction can be moved to the operands

qed

thus ?thesis using IHAPP(2) left' by blast

next
case (right' w2') from Cons.hyps[OF - left(2) right'(2)] obtain wr where
IHAPP: w ∈ w1 ⊗α wr wr ∈ w2' ⊗α w3 by blast
from IHAPP(2) left have e#wr ∈ e#w2' ⊗α w3 by (auto intro: cil-cons1)
moreover from right' IHAPP(1) have e#w ∈ w1 ⊗α e#wr by (auto intro: cil-cons2)
ultimately show ?thesis using right' by blast
qed

next
case (right w3') from Cons.hyps[OF - right(2) Cons.prems(2)] obtain wr where
IHAPP: w ∈ w1 ⊗α wr wr ∈ w2 ⊗α w3' by blast
from IHAPP(2) right cil-mon-pl[OF Cons.prems(2)] have e#wr ∈ w2 ⊗α e#w3' by (auto intro: cil-cons2)
moreover from IHAPP(1) right cil-mon-pl[OF Cons.prems(2)] have e#w ∈ w1 ⊗α e#wr by (auto intro: cil-cons2)
ultimately show ?thesis using right by blast
qed

lemma cil-assoc2:
assumes A: w∈w1⊗α wr and B: wr∈w2⊗α w3
shows ∃wl. w∈wl⊗α w3 ∧ wl∈w1⊗α w2
proof —
from A have A': w∈wr⊗α w1 by (simp add: cil-commute)
from B have B': wr∈w3⊗α w2 by (simp add: cil-commute)
from cil-assoc1[OF A' B'] obtain wl where w ∈ w3 ⊗α wl ∧ wl ∈ w2 ⊗α w1
by blast
thus ?thesis by (auto simp add: cil-commute)
qed

— Parts of the abstraction can be moved to the operands

lemma cil-map: w∈w1 ⊗(αof) w2 ⟹ map f w ∈ map f w1 ⊗α map f w2
proof (induct rule: cil-scl-induct-fix)
case empty thus ?case by auto
next
case (left e w' w1' w2)
have f e # map f w' ∈ f e # map f w1' ⊗α map f w2 proof (rule cil-cons1)
  from left(2) have fst ((αof) e) ⊂ mon-pl (map α (map f w2)) = {} by (simp only: map-map[symmetric])
  thus fst (α (f e)) ⊂ mon-pl (map α (map f w2)) = {} by (simp only: o-apply)
  thus ?case by simp
next
case (right e w' w2' w1)
have \( f \neq \# \operatorname{map} f w' \in \operatorname{map} f w1 \otimes_{\alpha} f \neq \# \operatorname{map} f w2' \) proof (rule cil-cons2) 

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

thus \( \operatorname{fst} (\alpha (f e)) \cap \operatorname{mon-pl} (\operatorname{map} \alpha (\operatorname{map} f w1)) = \{\} \) by (simp only; o-apply)

qed (rule right(3))

thus ?case by simp

qed

end

3 Acquisition Histories

theory AcquisitionHistory

imports ConsInterleave

begin

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

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

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

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

3.1 Definitions

Acquisition histories are modeled as functions from monitors to sets of mon-
itors. 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 \( h m = {} \).

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

\[
\text{lemma } ah-cases[cases set]: \[ h \in ah; h m = \{ \} \implies P ; m \in h m \implies P ] \implies P
\]

by (unfold ah-def) blast

3.2 Interleavability

Two acquisition histories \( h_1 \) and \( h_2 \) are considered interleavable, iff there
is no conflicting pair of monitors \( m_1 \) and \( m_2 \), where a pair of monitors \( m_1 \)
and \( m_2 \) is called conflicting iff \( m_1 \) is used in \( h_2 \) after entering \( m_2 \) and, vice
versa, \( m_2 \) is used in \( h_1 \) after entering \( m_1 \).

\[
\text{definition } ah-il :: ('m \Rightarrow 'm set) \Rightarrow ('m \Rightarrow 'm set) \Rightarrow bool \ (\text{infix} \ [\ast] \ 65)
\]

where

\[
h_1 [\ast] h_2 == \neg(\exists m_1 m_2. m_1 \in h_2 \land m_2 \in h_1)
\]

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

\[
\text{lemma } ah-il-lockset-disjoint: \[ h_1 \in ah; h_2 \in ah; h_1 [\ast] h_2 \] \implies h_1 m = \{ \} \lor h_2 m = \{ \}
\]

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

Of course, acquisition history interleavability is commutative

\[
\text{lemma } ah-il-commute: h_1 [\ast] h_2 \implies h_2 [\ast] h_1
\]

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

\[
\text{definition } mon-ah :: ('m \Rightarrow 'm set) \Rightarrow 'm set
\]

where

\[
mon-ah h == \bigcup \{ h(m) \mid m. \text{True} \}
\]
3.4 Ordering

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

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

— The ordering is compatible with interleavability, i.e. smaller acquisition histories are more likely to be interleavable.

\begin{enumerate}
\item \textbf{Lemma 3.4.1 (ah-leq-il):} \( h_1 \ [\ast] h_2; h_1' \leq h_1; h_2' \leq h_2 \implies h_1' [\ast] h_2' \)
\end{enumerate}

\textbf{Proof:} (unfold ah-il-def le-fun-def \( \text{where } b = \{a\} \)) blast+

\begin{enumerate}
\item \textbf{Lemma 3.4.2 (ah-leq-il-left):} \( h_1 [\ast] h_2; h_1' \leq h_1 \implies h_1' [\ast] h_2 \)
\item \textbf{Lemma 3.4.3 (ah-leq-il-right):} \( h_1 [\ast] h_2; h_2' \leq h_2 \implies h_1 [\ast] h_2' \)
\end{enumerate}

\textbf{Proof:} (unfold ah-il-def le-fun-def \( \text{where } b = \{a\} \)) blast+

3.5 Acquisition histories of executions

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

\begin{enumerate}
\item \textbf{Primrec ah-ah:} \( (\mapsto m \times \mapsto m) \rightarrow \mapsto m \rightarrow \mapsto m \)
\end{enumerate}

\item \textbf{Lemma 3.5.1 (ah-ah):} \( ah \ (\ast w) \ m = (if \ m \in \text{fst } e \ then \text{fst } e \cup \text{snd } e \cup \text{mon-pl } w \ else \ ah \ w \ m) \)

\item \textbf{Lemma 3.5.2 (ah-hd):} \( x \in \text{fst } e; x \in \text{fst } e \cup \text{snd } e \cup \text{mon-pl } w \implies x \in ah \ (\# e w) \)
\end{enumerate}

\textbf{Proof:} (induct \( w \)) (simp-all split: if-split-asm)

\begin{enumerate}
\item \textbf{Lemma 3.5.3 (ah-tl):} \( m \notin \text{fst } e; x \in ah \ w \ m \implies x \in ah \ (\# e w) \)
\end{enumerate}

\textbf{Proof:} (cases \( w \)) (simp-all split: if-split-asm)

\begin{enumerate}
\item \textbf{Lemma 3.5.4 (ah-cases):} \( x \in ah \ w \ m; \)
\item \textbf{Lemma 3.5.5 (ah-cons-cases):} \( m \in \text{fst } e; x \in ah \ w \ m \implies x \in ah \ (\# e w) \)
\end{enumerate}

\textbf{Proof:} (cases \( w \)) (simp-all split: if-split-asm)

13
\[ P \implies \]
by (simp-all split: if-split-asm)

lemma mon-ah-subset: mon-ah (α ah w) ⊆ mon-pl w
by (induct w) (auto simp add: mon-ah-def)

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

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

lemma ah-interleavable1:
  w ∈ w1 ⊗ ah (map α w1) [∗] ah (map α w2)
  — The lemma is shown by induction on the structure of the monitor consistent
  interleaving operator
proof (induct w α w1 w2 rule: cil-set-induct-fixa)
case empty show ?case by (simp add: ah-il-def) — The base case is trivial by
the definition of ([∗])
next
— Case: First step comes from the left word
case (left e w' w1' w2) show \( ?\text{case} \)

proof (rule ccontra) — We do a proof by contradiction

— Assume there is a conflicting pair in the acquisition histories

\[ \text{assume } \neg \alpha \# (\text{map } \alpha (e \# w1')) \mapsto \alpha \# (\text{map } \alpha w2) \]

then obtain \( m1 m2 \) where CPAIR: \( m1 \in \alpha \# (\text{map } \alpha (e\#w1')) m2 \in \alpha \# (\text{map } \alpha w2) \) by (unfold ah-il-def, blast)

— It comes either from the first step or not

from CPAIR(1) have \((m2\#\text{fst } (\alpha e) \land m1 \in \text{fst } (\alpha e) \cup \text{snd } (\alpha e) \cup \text{mon-pl} (\text{map } \alpha w2)) \lor (m2\#\text{fst } (\alpha e) \land m1 \in \alpha \# (\text{map } \alpha w1') m2) \) (is ?\text{CASE1} \lor ?\text{CASE2})

by (auto split: if-split-asm)

moreover {
— Case: One monitor of the conflicting pair is entered in the first step of the left path

\[ \text{assume } ?\text{CASE1 hence } C: m2\in\text{fst } (\alpha e) .. \]

— Because the paths are consistently interleavable, the monitors entered in the first step must not occur in the other path

from \( \text{left(2) mon-ah-subset[of map } \alpha w2 \) have } \text{fst } (\alpha e) \cap \text{mon-ah } (\alpha \# (\text{map } \alpha w2)) = \{ \} \) by auto

— But this is a contradiction to being a conflicting pair

with \( C \) CPAIR(2) have False by (unfold mon-ah-def, blast)

} moreover {
— Case: The first monitor of the conflicting pair is entered after the first step of the left path

\[ \text{assume } ?\text{CASE2 hence } C: m1 \in \alpha \# (\text{map } \alpha w1') m2 .. \]

— But this is a contradiction to the induction hypothesis, that says that the acquisition histories of the tail of the left path and the right path are interleavable

with \( \text{left(3) CPAIR(2) have } \text{False by (unfold ah-il-def, blast)} \)

} ultimately show False ..

qed

next

— Case: First step comes from the right word. This case is shown completely analogous

case (right e w' w2' w1) show \( ?\text{case} \)

proof (rule ccontra)

assume \( \neg \alpha \# (\text{map } \alpha w1) \mapsto \alpha \# (\text{map } \alpha (e\#w2')) \)

then obtain \( m1 m2 \) where CPAIR: \( m1 \in \alpha \# (\text{map } \alpha w1) m2 \in \alpha \# (\text{map } \alpha (e\#w2')) m1 \) by (unfold ah-il-def, blast)

from CPAIR(2) have \((m1\#\text{fst } (\alpha e) \land m2 \in \text{fst } (\alpha e) \cup \text{snd } (\alpha e) \cup \text{mon-pl} (\text{map } \alpha w2)) \lor (m1\#\text{fst } (\alpha e) \land m2 \in \alpha \# (\text{map } \alpha w2') m1) \) (is ?\text{CASE1} \lor ?\text{CASE2})

by (auto split: if-split-asm)

moreover {
— Case: One monitor of the conflicting pair is entered after the first step of the right path

\[ \text{assume } ?\text{CASE1 hence } C: m1\in\text{fst } (\alpha e) .. \]

from \( \text{right(2) mon-ah-subset[of map } \alpha w1 \) have } \text{fst } (\alpha e) \cap \text{mon-ah } (\alpha \# (\text{map } \alpha w1)) = \{ \} \) by auto

with \( C \) CPAIR(1) have False by (unfold mon-ah-def, blast)

} moreover {
— Case: The first monitor of the conflicting pair is entered in the first step of the right path

\[ \text{assume } ?\text{CASE2 hence } C: m2 \in \alpha \# (\text{map } \alpha w2') m1 .. \]
with right(3) CPAIR(1) have False by (unfold ah-il-def, blast)
} ultimately show False ..

qed

lemma ah-interleavable2:
assumes A: \( \alpha \text{ah} (\text{map } \alpha \ w1) \neq \{\} \)
shows \( w1 \otimes_\alpha w2 \neq \{\} \)
  — This lemma is shown by induction on the sum of the word lengths

proof —
  — To apply this induction in Isabelle, we have to rewrite the lemma a bit
  \{ fix \ n \ have \ !!w1 w2. \[ [ \alpha \text{ah} (\text{map } \alpha \ w1) \neq \{\} \&\& \alpha \text{ah} (\text{map } \alpha \ w2); \ n=\text{length } w1 + \text{length } w2 ] \] \} = \Rightarrow w1 \otimes_\alpha w2 \neq \{\}

proof (induct n rule: nat-less-induct [case-names I])
  — We first rule out the cases that one of the words is empty
  case (I \ n \ w1 \ w2)
  show ?case
    proof (cases w1)
      — If the first word is empty, the lemma is trivial
      case Nil
      with I.prems show ?thesis by simp
    next
      case (Cons e1 w1')
      note CONS1 = this
      show ?thesis
        proof (cases w2)
          — If the second word is empty, the lemma is also trivial
          case Nil
          with I.prems show ?thesis by simp
        next
          — The interesting case is if both words are not empty
          case (Cons e2 w2')
          note CONS2 = this
          — In this case, we check whether the first step of one of the words can safely be executed without blocking any steps of the other word
          show ?thesis
            proof (cases \fst (\alpha \ e1) \cap \text{mon-pl} (\text{map } \alpha \ w2) = \{\})
              case True — The first step of the first word can safely be executed
              — From the induction hypothesis, we get that there is a consistent interleaving of the rest of the first word and the second word
              have w1'' \otimes_\alpha w2 \neq \{\}
              proof
                from I.prems(1) CONS1 \text{ah-leq-il-left}[OF - \alpha \text{-ileq}[OF \text{le-list-map}, \text{OF less-eq-list-drop}[OF \text{order-refl]}] have \alpha \text{ah} (\text{map } \alpha \ w1') \neq \{\} \alpha \text{ah} (\text{map } \alpha 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
                proof (cases \fst (\alpha \ e2) \cap \text{mon-pl} (\text{map } \alpha \ w1) = \{\})
                case True — The first step of the second word can safely be executed
                — This case is shown analogously to the latter one

next
  case False
    note C1 = this
    show ?thesis
      proof (cases \fst (\alpha \ e2) \cap \text{mon-pl} (\text{map } \alpha \ w1) = \{\})
    case True — The first step of the second word can safely be executed
      — This case is shown analogously to the latter one

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

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

theorem ah-interleavable:
\[ (\alpha \text{ah} (\text{map} \alpha \ w1) [s] \alpha \text{ah} (\text{map} \alpha \ w2)) \iff (\mu \otimes \alpha \nu \neq \{\}) \]
using ah-interleavable1 ah-interleavable2 by blast

3.6 Acquisition history backward update

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

definition ah-update :: ('m \Rightarrow 'm set) \Rightarrow ('m set * 'm set) \Rightarrow 'm set \Rightarrow ('m \Rightarrow 'm set)
where
ah-update \ h \ F \ M \ m == \text{if } m \in \text{fst} \ F \text{ then } \text{fst} \ F \cup \text{snd} \ F \cup M \text{ else } h \ m

Intuitively, \( \text{ah-update} \ h \ (E, U) \) \( M \ m \) 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
Lemma **ah-update-cons**: \( \alpha ah (e\#w) = ah-update (\alpha ah w) e (\text{mon-pl} w) \)

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 [where 'b= 'a set])

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 [where 'b= 'a set])

end

4 Labeled transition systems

theory LTS imports Main begin

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

4.1 Definitions

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

**Type-synonym** \((\text{'c,'a})\text{ LTS} = (\text{'c } \times \text{'a } \times \text{'c})\text{ set}\)

Transitive reflexive closure

**Inductive-set**

\( \text{trcl} :: (\text{'c,'a})\text{ LTS} \Rightarrow (\text{'c,'a list})\text{ LTS} \)

for \( t \)

where

| empty[simp]: \( (c,[])\text{,}c) \in \text{trcl} t \)
| cons[simp]: \( (\text{c,a,c'}) \in t; (c',w,c'') \in \text{trcl} t \] \( \Rightarrow \) \( (c,a\#w,c'') \in \text{trcl} t \)

4.2 Basic properties of transitive reflexive closure

Lemma **trcl-empty-cons**: \( (c,[]\text{,}c')\in \text{trcl} t \) \( \Rightarrow \) \( (c=c') \)

by (auto elim: trcl_cases)

Lemma **trcl-empty-simp[simp]**: \( (c,[]\text{,}c')\in \text{trcl} t = (c=c') \)

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

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

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

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

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

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

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

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

lemma trcl-rev-cons: \[[ (c, w, ch) \in \text{trcl } T \; \& \; (ch, e, c') \in T \] \] \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, w, c') \in \text{trcl } S ; !! c. P c[\] c; !! c. w c' e c''. [ (c, w, c') \in \text{trcl } S ; (c', e, c'') \in S ; P c w c'] \implies P c (w@\[ e \], e'') ] 

\implies P c w c'

by (induct w rule: rev-induct) (auto dest: trcl-rev-uncons)

lemma trcl-cons2: \[[ (c, e, ch) \in T ; (ch, f, c') \in T \] \] 

\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 B. 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 \land 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 S] show x \in \text{trcl } S by auto

next

assume x \in \text{trcl } (S \cap R)

with trcl-mono[of S \cap R 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]:
(c,w,c')∈trcl S;
[c=c'; w=[]] ⇒ P;
[c∈Domain S; c'∈Range (Range S)]⇒P
⇒ P
apply (erule-tac trcl-rev-cases)
apply auto
apply (erule trcl.cases)
apply auto
done
end

5 Thread Tracking

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

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

5.1 Semantic on multiset configuration

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

definition gtr gtrs == { (add-mset s c,e,add-mset s' c') | s c e s' c'. ((s,c),e,(s',c'))∈gtrs }

lemma gtrI-s: ((s,c),e,(s',c'))∈gtrs ⇒ (add-mset s c,e,add-mset s' c')∈trcl gtrs
by (unfold gtr-def, auto)

lemma gtrI: ((s,c),w,(s',c'))∈trcl gtrs
⇒ (add-mset s c,w,add-mset s' c')∈trcl (gtr gtrs)
by (induct rule: trcl-pair-induct) (auto dest: gtrI-s)
lemma $\text{gtrE}$: 
$$(c,e,c') \in \text{gtr} T;$$
$$\n s \in c \leftarrow c' \in \text{add-mset} s \; c' = \text{add-mset} s' \; c'; \; ((s,c,e),(s',c')) \in T \implies P$$
\begin{align*}
\| & \implies P \\
& \text{by (unfold gtr-def) auto}
\end{align*}

lemma $\text{gtr-empty-conf-s}$:
$$(\{\#\},w,c') \notin \text{gtr} S$$
$$(c,w,\{\#\}) \notin \text{gtr} S$$
\begin{align*}
& \text{by (auto elim: gtrE)}
\end{align*}

lemma $\text{gtr-empty-conf-1}$:
$$(\{\#\},w,c') \in \text{trcl} \text{(gtr} S)$$
\begin{align*}
(w=\| \land c'=\{\#\}) & \leftrightarrow \implies P \\
& \text{by (induct w) (auto dest: trcl-uncons)}
\end{align*}

lemma $\text{gtr-empty-conf-2}$:
$$(c,w,\{\#\}) \in \text{trcl} \text{(gtr} S)$$
\begin{align*}
(w=\| \land c=\{\#\}) & \leftrightarrow \implies P \\
& \text{by (induct w rule: rev-induct) (auto dest: trcl-rev-uncons)}
\end{align*}

lemma $\text{gtr-find-thread}$:
$$(c,e,c') \in \text{gtr gtrs};$$
$$\n s \in c \leftarrow c' \in \text{add-mset} s \; c' = \text{add-mset} s' \; c'; \; ((s,c,e),(s',c')) \in \text{gtrs} \implies P$$
\begin{align*}
\| & \implies P \\
& \text{by (unfold gtr-def) auto}
\end{align*}

lemma $\text{gtr-step-cases}$:
$$(\text{add-mset} s \; c,e,e') \in \text{gtr gtrs};$$
$$\n s' \in c' \leftarrow c' = \text{add-mset} s' \; c'; \; ((s,c,e),(s',c')) \in \text{gtrs} \implies P;$$
$$\n c' \in \text{add-mset} s s' \; c' = \text{add-mset} ss' \; c'; \; ((ss,add-mset ss \; c'),e,(ss',add-mset ss' \; c')) \in \text{gtrs} \implies P$$
\begin{align*}
\| & \implies P \\
& \text{by (auto elim!: gtr-find-thread mset-single-cases)}
\end{align*}

lemma $\text{gtr-rev-cases}$:
$$(\text{add-mset} s' \; c') \in \text{gtr gtrs};$$
$$\n s \in c \leftarrow c = \text{add-mset} s \; c; \; ((s,c,e),(s',c')) \in \text{gtrs} \implies P;$$
$$\n c' \in \text{add-mset} ss' \; c' = \text{add-mset} ss' \; c; \; ((ss,add-mset s \; c),e,(ss',add-mset s' \; c')) \in \text{gtrs} \implies P$$
\begin{align*}
\| & \implies P \\
& \text{by (auto elim!: gtr-find-thread mset-single-cases)}
\end{align*}

5.2 Invariants

lemma $\text{gtr-preserve-s}$:
$$(c,e,c') \in \text{gtr} T;$$
$$\n s \in c \leftarrow c = \text{add-mset} s \; c; \; ((s,c,e),(s',c')) \in \text{gtrs} \implies P \; (\text{add-mset} s' \; c')$$
\begin{align*}
\| & \implies P \; c' \\
& \text{by (unfold gtr-def) blast}
\end{align*}

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

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 :: \[\left((s' \times s \text{ multiset}), l\right) \text{LTS}\]
  assumes env-no-step-s [cases set, case-names csp]:
  \[\left((s,c),e,(s',c')) \in \text{gtrs}; \quad \text{!! }\text{csp}. \quad \text{c}'=\text{csp}+\text{c} \implies P \right] \implies P\]

— The property of not changing existing threads in the context transfers to paths

lemma (in env-no-step) env-no-step-s[cases set, case-names csp]:
  \[\left((s,c),w,(s',c')) \in \text{trcl } \text{gtrs};\]
  \[\quad \text{!! csp. } \text{c}'=\text{csp}+\text{c} \implies P\]
  \[\] \implies P\]

proof —
  have \[\left((s,c),w,(s',c')) \in \text{trcl } \text{gtrs} \implies \exists \text{csp. } c'=\text{csp}+c\] proof (induct rule: trcl-pair-induct)
  case empty thus \[?\text{case by (auto intro: exI[of } \text{-}\{\#}\text{])})\]
  next
  case \[\left(\text{cons } s c e sh w s' c'\right)\] note IHP=\text{this}
  from env-no-step-s[\text{OF IHP(1)}] obtain \text{cspb} where \[\text{ch}=\text{cspb}+\text{c} \text{ by auto}\]
  moreover from IHP(3) obtain \text{csp'} where \text{c}'=\text{csp}'+\text{ch} \text{ by auto}\n  ultimately have \text{c}'=\text{csp}'+\text{cspb}+\text{c} \text{ by (simp add: union-ascoc}\)
  thus \[?\text{case by blast}\]
  qed

moreover assume \[\left((s,c),w,(s',c')) \in \text{trcl } \text{gtrs} \quad \text{!! csp. } c'=\text{csp}+c \implies P\]

ultimately show \[?\text{thesis by blast}\]
  qed

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

loc The thread made the step

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

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

5.4 Explicit local context

In the multiset semantics, a single thread has no identity. This may become
a problem when reasoning about a fixed thread during an execution. For
example, in our constraint-system-based approach the operational charac-
terization 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
eqv w == filter (λe. case e of LOC a ⇒ False | ENV a ⇒ True) w

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

Standard simplification lemmas

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

lemma loc-single[simp]: loc [a] = (case a of LOC e ⇒ [a] | ENV e ⇒ [])
  by (unfold loc-def) (auto split: el-step.split)

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

lemma loc-unconc[simp]: loc (a@b) = loc a @ loc b
  by (unfold loc-def, simp)

lemma env-single[simp]: env [a] = (case a of LOC e ⇒ [] | ENV e ⇒ [a])
  by (unfold env-def) (auto split: el-step.split)

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

lemma env-unconc[simp]: env (a@b) = env a @ env b
  by (unfold env-def, simp)
The following simplification lemmas are for converting between paths of the multiset- and loc/env-semantics

**Lemma le-rem-simps [simp]:**
- le-rem-s (LOC a) = a
- le-rem-s (ENV a) = a
  by (unfold le-rem-s-def, auto)

**Lemma le-rem-id-simps [simp]:**
- le-rem-s ◦ LOC = id
- le-rem-s ◦ ENV = id
  by (auto intro: ext)

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

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

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

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 S ⇒ (add-mset s c,le-rem-s e,add-mset s' c')∈gtr S
  by (cases rule: gtrp_cases, auto intro: gtrI-s)

  **Proof**
  - fix c c' e ss ss' assume ((ss,add-mset s c),e,(ss',add-mset s c'))∈T
  - hence (add-mset ss (add-mset s c),e,add-mset ss' (add-mset s c')) ∈ gtr T by (auto intro!: gtrI-s)
  - thus (add-mset s (add-mset ss c), e, add-mset s (add-mset ss' c')) ∈ gtr T by (auto simp add: add-mset-commute)

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

lemma (in env-no-step) \( gtrp2gtr-s[\text{cases set, case-names } gtrp] \):
\begin{enumerate}
\item \textbf{assumes} \( A : (\text{add-mset } s, c, c') \in gtr gtrs \)
\item \textbf{and} \textbf{CASE:} \((s,s') \in e e \) \[ c' = (\text{add-mset } s', c') \; e = \text{le-rem-s } e e; \]
\item \[(s,c),ee,(s',ce') \in gtrp gtrs \]
\end{enumerate}
\[
\implies P
\]
\begin{enumerate}
\item \textbf{using} \( A \)
\item \textbf{proof} (cases rule: gtr-step-cases)
\end{enumerate}
\begin{enumerate}
\item \textbf{case} \((\text{loc } s', e e) \) \textbf{hence} \[(s,c),\text{LOC } e, (s', ce') \in gtrp gtrs \text{ by (blast intro: gtrp-loc)} \]
\item \textbf{with} \textbf{loc(1) show} \( ?\text{thesis by (rule-tac CASE) auto} \)
\end{enumerate}
\begin{enumerate}
\item \textbf{next}
\end{enumerate}
\begin{enumerate}
\end{enumerate}
\begin{enumerate}
\item \textbf{from} \textbf{env-no-step-s[}OF other(3)\text{]} \textbf{obtain} \( \text{csp where CE'FMT: } c' = \text{csp + (add-mset } s, cc) \).
\end{enumerate}
\begin{enumerate}
\item \textbf{with} \textbf{other(3) have} \[(s,s',\text{add-mset } s, cc), (s', \text{add-mset } s, (\text{csp + cc})) \in gtrp gtrs \text{ by auto} \]
\end{enumerate}
\begin{enumerate}
\item \textbf{from} \textbf{gtrp-env[}OF this\text{]} \textbf{other(1) have} \[(s, c), \text{ENV } e, s, \{\text{#s' #}\} + (\text{csp + cc}) \in gtrp gtrs \text{ by simp} \)
\end{enumerate}
\begin{enumerate}
\item \textbf{moreover from} \textbf{other CE'FMT have} \( c' = \{\text{#s' #}\} + (\{\text{#s' #}\} + (\text{csp + cc}) \)
\end{enumerate}
\begin{enumerate}
\item \textbf{by} (simp add: union-ac)
\end{enumerate}
\begin{enumerate}
\item \textbf{ultimately show} \( ?\text{thesis by (rule-tac CASE) auto} \)
\end{enumerate}
\textbf{qed}

lemma (in env-no-step) \( gtrp2gtr[\text{cases set, case-names } gtrp] \):
\begin{enumerate}
\item \textbf{assumes} \( A : (\text{add-mset } s, c, c') \in \text{trcl (} gtrp gtrs) \)
\item \textbf{and} \textbf{CASE:} \((s,s') \in e e \) \[ c' = \text{add-mset } s', c'; \]
\item \[w = \text{map le-rem-s } w w; \]
\item \[
((s,c),ww,(s',ce')) \in \text{trcl (} gtrp gtrs) \]
\end{enumerate}
\begin{enumerate}
\item \textbf{shows} \( P \)
\item \textbf{proof --}
\end{enumerate}
\begin{enumerate}
\item \textbf{have} \((s,c), (\text{add-mset } s, c, c') \in \text{trcl (} gtrp gtrs) \implies \exists s' ce' w w'. c' = \text{add-mset } s', ce' \land w = \text{map le-rem-s } w w \land ((s,c),ww,(s',ce')) \in \text{trcl (} gtrp gtrs) \text{ proof (induct w)} \)
\begin{enumerate}
\item \textbf{case} \textbf{Nil} \textbf{thus} \( ?\text{case by auto} \)
\end{enumerate}
\item \textbf{next}
\begin{enumerate}
\item \textbf{case} \( (\text{Cons } e, w) \) \textbf{then obtain} \( \text{ch where SPLIT: } (\text{add-mset } s, c, ch) \in gtrp gtrs \text{ (ch,w,c') \in trcl (} gtrp gtrs) \text{ by (auto dest: trcl-uncons)} \)
\item \textbf{from} \textbf{gtrp2gtr-s[}OF SPLIT(1)\text{]} \textbf{obtain} \( \text{sh ceh ee where FS: ch = add-mset sh ceh e = le-rem-s } ee ((s, c), ee, sh, ceh) \in gtrp gtrs \text{ by blast} \)
\item \textbf{moreover from} \textbf{FS(1) SPLIT(2) Cons.hyps obtain} \( s' ce' w w \) \textbf{where IH:}\( c' = \text{add-mset } s', ce' w = \text{map le-rem-s } w w ((sh,ceh),ww,(s',ce')) \in \text{trcl (} gtrp gtrs) \text{ by blast} \)
\item \textbf{ultimately have} \((s,c),ee#ww,(s',ce')) \in \text{trcl (} gtrp gtrs) \text{ e#w = map le-rem-s (ee#ww) by auto} \)
\item \textbf{with} \textbf{IH(1) show} \( ?\text{case by iprove} \)
\end{enumerate}
\textbf{qed}
with A CASE show ?thesis by blast
qed

5.4.4 Invariants

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

lemma gtrp-preserve:
  assumes A: ((s,c),w,(s',c'))∈trcl (gtrp T)
  and INIT: P (add-mset s c)
  and PRES: ![s c s' c' e. [P (add-mset s c); ((s,c),e,(s',c'))∈T]
  shows P (add-mset s' c')
proof –
  from gtr-preserve[OF gtrp2gtr[OF A], where P=P, OF INIT] PRES show P (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**

\[
\text{edgeAnn} = \text{Base } 'b\text{a} \mid \text{Call } 'p \mid \text{Spawn } 'p
\]

**type-synonym**

\[
\text{edge } = ('n' \times (p,ba) \text{edgeAnn} \times 'n)
\]

**record**

\[
\text{flowgraph-rec} =
\begin{align*}
\text{edges} &:: (n,p,ba) \text{ edge set} \quad \text{Set of annotated edges} \\
\text{main} &:: 'p \rightarrow 'n \quad \text{Main procedure} \\
\text{entry} &:: 'p \Rightarrow 'n \quad \text{Maps a procedure to its entry point} \\
\text{return} &:: 'p \Rightarrow 'n \quad \text{Maps a procedure to its return point} \\
\text{mon} &:: 'p \Rightarrow 'm \text{ set} \quad \text{Maps procedures to the set of monitors they allocate} \\
\text{proc-of} &:: 'n \Rightarrow 'p \quad \text{Maps a node to the procedure it is contained in}
\end{align*}
\]

**definition**

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

**lemma**

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

**using**

\text{unfold initialproc-def, blast}

### 6.2 Basic properties

**lemma**

\[
\text{spawn-no-mon} : (u, \text{Spawn } p, v) \in \text{edges } fg \Rightarrow \text{mon } fg \ p = \{}
\]

**using**

\text{initial-no-mon, by (unfold initialproc-def, blast)}

**lemma**

\[
\text{main-no-mon} : \text{mon } (\text{main } fg) = \{}
\]

**using**

\text{initial-no-mon, by (unfold initialproc-def, blast)}

**lemma**

\[
\text{entry-return-same-proc} : \text{proc-of } (\text{entry } (\text{main } fg) p) = p
\]

### 29
entry fg p = return fg p' \implies 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' \implies 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' \implies 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 \implies \text{entry} fg p \neq \text{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{initialproc} fg p; (\text{entry} fg p,l,v)\in\text{edges} fg
\implies \exists p', l=\text{Call} p' \land \text{isolated-ret} fg p'
6.4 Example Flowgraph

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

**definition**

definition example-fg == {}  
  edges = {[(0::nat,0::nat),Call 1,(0,1)), ((1,0),Spawn 0,(1,0)), 
          ((1,0),Call 0, (1,0))},
  main = 0,
  entry = \lambda p. (p,0),
  return = \lambda p. (p,1),
  mon = \lambda p. if p=1 then {0} else {},
  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 add: initialproc-def)
apply (simp add: initialproc-def)
apply (simp add: initialproc-def isolated-ret-def)
done

end

7 Operational Semantics

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

7.1 Configurations and labels

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

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

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

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

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

### 7.2 Monitors

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

**definition**

— The monitors of a node are the monitors the procedure of the node synchronizes on

\[ mon-n fg n \equiv mon fg (proc-of fg n) \]

**definition**

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

\[ mon-s fg s \equiv \bigcup \{ mon-n fg n \mid n . n \in set s \} \]

**definition**

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

\[ mon-c fg c \equiv \bigcup \{ mon-s fg s \mid s . s \in \# c \} \]

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

**definition** mon-e :: ('b, 'c, 'd, 'a, 'e) flowgraph-rec-scheme \( \Rightarrow \) ('c, 'f) label \( \Rightarrow \) 'a set where

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

**lemma** mon-e-simps [simp]:

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

— The monitors of a path are the monitors of all procedures that are called on the path

**definition**

\[ mon-w fg w \equiv \bigcup \{ mon-e fg e \mid e . e \in set w \} \]

**lemma** mon-s-alt: mon-s fg s \( \Rightarrow \) (mon fg \( \cdot \) proc-of fg \( \cdot \) set s)

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

**lemma** mon-c-alt: mon-c fg c \( \Rightarrow \) (mon-s fg \( \cdot \) set-mset c)

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

**lemma** mon-w-alt: mon-w fg w \( \Rightarrow \) (mon-e fg \( \cdot \) set w)

- by (unfold mon-w-def) (auto intro: eq-reflection)

**lemma** mon-sI: \( \exists n \in set s ; m \in mon-n fg n \Rightarrow m \in mon-s fg s \)

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

**lemma** mon-sD: \( m \in mon-s fg s \Rightarrow \exists n \in set s . m \in mon-n fg n \)
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-n-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[simp]:
  \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 }\{} = \{\}
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)

33
lemma mon-w-single(simp): mon-w fg [e] = mon-e fg e
  by (unfold mon-w-def, auto)
lemma mon-w-unconc: mon-w fg (wa@wb) = mon-w fg wa ∪ mon-w fg wb
  by (unfold mon-w-def) auto
lemma mon-w-uncons(simp): mon-w fg (e#w) = mon-e fg e ∪ mon-w fg w
  by (rule mon-w-unconc[where wa=[e], simplified])
lemma mon-w-ileq: w ≺ w' ⇒ mon-w fg w ⊆ mon-w fg w'
  by (induct rule: less-eq-list-induct) auto

7.3 Valid configurations

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

definition valid fg c == ∀ s s'. {#s, s'} ⊆ # c → mon-s fg s ∩ mon-s fg s' = { }
lemma valid-empty[simp, intro!]: valid fg {#}
  by (unfold valid-def, auto)
lemma valid-single[simp, intro!]: valid fg {#s#}
  by (unfold valid-def subset-mset-def) auto
lemma valid-split1: valid fg (c+c') =⇒ valid fg c ∧ valid fg c' ∧ mon-c fg c ∩ mon-c fg c' = { }
  apply (unfold valid-def)
  apply (auto simp add: mset-le-incr-right)
  apply (drule mon-cD)+
  apply auto
  apply (subgoal-tac {#s#}+{#sa#} ⊆ # c+c')
  apply (auto dest!: multi-member-split)
  done
lemma valid-split2: [valid fg c; valid fg c'; mon-c fg c ∩ mon-c fg c' = { } ] =⇒ valid fg (c+c')
  apply (unfold valid-def)
  apply (intro impI allI)
  apply (erule mset-2dist2-cases)
  apply simp-all
  apply (blast intro: mon-cI)+
  done
lemma valid-union-conc: valid fg (c+c') ↔ (valid fg c ∧ valid fg c' ∧ mon-c fg c ∩ mon-c fg c' = { })
  by (blast dest: valid-split1 valid-split2)
lemma valid-add-mset-conc: valid fg (add-mset x c') ↔ (valid fg c' ∧ mon-s fg x ∩ mon-c fg c' = { })
  unfolding add-mset-add-single[of x c'] valid-union-conc by (auto simp: mon-s-def)
lemmas valid-unconc = valid-union-conc valid-add-mset-conc
Lemma valid-no-mon: mon-c fg c = {} → valid fg c
Proof (unfold valid-def, intro allI impI)
  fix s s'
  assume A: mon-c fg c = {} and B: {#s, s'#} ⊆# c
  from mon-c-mono[OF B, of fg] have mon-s fg s = {} mon-s fg s' = {} by (auto simp add: mon-c-unconc)
  thus mon-s fg s ∩ mon-s fg s' = {} by blast
qed

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

Lemma atU-s-decomp[simp]: atU-s U (s@s') = (atU-s U s ∨ (s=[] ∧ atU-s U s'))
  by (induct s) auto
— A configuration is at U if it contains a stack that is at U
Definition
  atU U c == ∃s. s ∈# c ∧ atU-s U s

Lemma atU-fmt: [atU U c; !ui r. [ui#r ∈# c; ui ∈ U] → P] → P
  apply (unfold atU-def)
  apply auto
  apply (case-tac s)
  apply auto
  done

Lemma atU-union-cases[case-names left right, consumes l]: []
  atU U (c1+c2);
  atU U c1 ⇒ P;
  atU U c2 ⇒ P
  [] ⇒ P
  by (unfold atU-def) (blast elim: mset-un-cases)

Lemma atU-add: atU U c ⇒ atU U (c+ce) ∧ atU U (ce+c)
  by (unfold atU-def) (auto simp add: union-ac)

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

Lemma atU-empty[simp]: ¬atU U {#}
  by (unfold atU-def, auto)

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

Lemma atU-single-top[simp]: atU U {#u#r#} = (u ∈ U)

by (auto)

**Lemma atU-add-mset[simp]:** \( atU\ U \ (add\-mset\ c\ c2) = (atU\-s\ U\ c \lor atU\ U\ c2) \)

**Unfolding** add-mset-add-single[of\ c\ c2] atU-union by auto

**Lemma atU-xchange-stack: atU\ U\ (add\-mset\ (u\#r)\ c) \implies atU\ U\ (add\-mset\ (u\#r')\ c) **

by (simp)

— A configuration is simultaneously at \( U \) and \( V \) if it contains a stack at \( U \) and another one at \( V \)

**Definition**
\[
\text{atUV} \ U \ V \ c \equiv \exists su sv. \{\#su\#\} + \{\#sv\#\} \subseteq \#c \land atU\-s\ U\ su \land atU\-s\ V\ sv
\]

**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]:**
\[
\text{atUV} \ U \ V\ (c1 + c2) \iff
\]
\[
(\text{atUV} \ U \ V\ c1) \lor
(\text{atUV} \ U \ V\ c2) \lor
(\text{atU} \ U\ c1 \land \text{atU} \ V\ c2) \lor
(\text{atU} \ V\ c1 \land \text{atU} \ U\ c2)
\]

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]:**
\[
\text{atUV} \ U \ V\ (add\-mset\ c\ c2) \iff
\]
\[
(\text{atUV} \ U \ V\ c2) \lor
(\text{atU} \ U\ \{\#c\#\} \land \text{atU} \ V\ c2) \lor
(\text{atU} \ V\ \{\#c\#\} \land \text{atU} \ U\ c2)
\]

unfolding add-mset-add-single[of\ c\ c2]
unfolding atUV-union
by auto

**Lemma atUV-union-cases[case-names left right lr rl, consumes I]:** [ 
\[
\text{atUV} \ U \ V\ (c1 + c2);\]
\[
\text{atUV} \ U \ V\ c1 \implies P;\]
\[
\text{atUV} \ U \ V\ c2 \implies P;\]
\[
[\text{atU} \ U\ c1; \text{atU} \ V\ c2] \implies P;
\]

36
\[ \text{atU } V \ c1; \text{ atU } U \ c2 \Rightarrow P \]
\[ \quad \Rightarrow P \]
\text{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.

\text{inductive-set}
\begin{align*}
\text{refpoint} :: (\text{'n}, \text{'p}, \text{'ba}, \text{'m}, \text{'more}) \text{ flowgraph-rec-scheme} & \Rightarrow \\
\text{\quad } (\text{'n conf} \times (\text{'p,ba} \ label \times 'n conf) \ set)
\end{align*}
\text{for fg}
\text{where}

- A base edge transforms the top node of one stack and leaves the other stacks untouched.

\text{refpoint-base:} \ \llbracket (u, \text{Base } a, v) \in \text{edges } fg; \ \text{valid } fg (\{\# u\# r\# \} + c) \rrbracket
\quad \Rightarrow (\text{add-mset } (u\# r) \ c, \text{LBase } a, \text{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.

\text{refpoint-call:} \ \llbracket (u, \text{Call } p, v) \in \text{edges } fg; \ \text{valid } fg (\{\# u\# r\# \} + c); \ \text{mon } fg \ p \ \cap \ \text{mon-c } fg \ c = \{\} \rrbracket
\quad \Rightarrow (\text{add-mset } (u\# r) \ c, \text{LCall } p, \text{add-mset } (\text{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.

\text{refpoint-ret:} \ \llbracket \ \text{valid } fg (\{\# return \ fg \ p\# r\# \} + c) \rrbracket
\quad \Rightarrow (\text{add-mset } (\text{return } fg \ p\# r) \ c, \text{LRet}, (\text{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.

\text{refpoint-spawn:} \ \llbracket (u, \text{Spawn } p, v) \in \text{edges } fg; \ \text{valid } fg (\text{add-mset } (u\# r) \ c) \rrbracket
\quad \Rightarrow (\text{add-mset } (u\# r) \ c, \text{LSpawn } p, \text{add-mset } (v\# r) \ (\text{add-mset } (\text{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.

\text{inductive-set}
\begin{align*}
\text{trss} :: (\text{'n}, \text{'p}, \text{'ba}, \text{'m}, \text{'more}) \text{ flowgraph-rec-scheme} & \Rightarrow \\
\end{align*}
(((’n list * ’n conf) * (’p,’ba) label * (’n list * ’n conf)) set
for fg
where
\[\text{trss-base: } \{(u, \text{Base } a,v) \in \text{edges } fg \} \implies \]
\[\{(u \# r, c), \text{LBase } a, (v \# r, c) \} \in \text{trss } fg\]
| trss-call: \[\{(u, \text{Call } p,v) \in \text{edges } fg; \text{mon } fg p \cap \text{mon-c } fg c = \{\}\} \implies \]
\[\{(u \# r, c), \text{LCall } p, (\text{entry } fg p)\# v \# r, e, c) \in \text{trss } fg\]
| trss-ret: \[\{(\text{return } fg p)\# r, c), \text{LRet}, (r, c) \in \text{trss } fg\]
| trss-spawn: \[\{(u, \text{Spawn } p,v) \in \text{edges } fg \} \implies \]
\[\{(u \# r, c), \text{LSpawn } p, (v \# r, \text{add-mset } \text{entry } fg p|c) \} \in \text{trss } fg\]

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

### 7.6 Basic properties

#### 7.6.1 Validity

**Lemma (in flowgraph)** trss-valid-preserve-s:
\[\text{valid } fg \text{ (add-mset s } c); ((s,c),e,(s',c'))\in\text{trss } fg\] \implies \text{valid } fg \text{ (add-mset } s' c')

apply (erule trss-base)
apply (simp-all add: valid-unconc mon-c-unconc)
by (blast dest: mon-nsame-proc edges-part)+

**Lemma (in flowgraph)** trss-valid-preserve:
\[\{(s,c),u,(s',c')\in\text{trcl } \text{trss } fg; \text{valid } fg \{\# s'\} + c\} \implies \text{valid } fg \{\# s\} + c'

by (induct rule: trcl-pair-induct) (auto intro: trss-valid-preserve-s)

**Lemma (in flowgraph)** tr-valid-preserve-s:
\[\{(c,e,c')\in \text{tr } fg; \text{valid } fg c\} \implies \text{valid } fg c'

by (rule gtrp-preserve-s[where P=valid fg]) (auto dest: trss-valid-preserve-s)

**Lemma (in flowgraph)** tr-valid-preserve:
\[\{(c,w,c')\in \text{trcl } \text{tr } fg; \text{valid } fg c\} \implies \text{valid } fg c'

by (rule gtrp-preserve[where P=valid fg]) (auto dest: trss-valid-preserve-s)

**Lemma (in flowgraph)** trp-valid-preserve-s:
\[\{(s,c),u,(s',c')\in \text{trp } fg; \text{valid } fg \text{ (add-mset s } c)\] \implies \text{valid } fg \text{ (add-mset } s' c')

by (rule gtrp-preserve-s[where P=valid fg]) (auto dest: trss-valid-preserve-s)

**Lemma (in flowgraph)** trp-valid-preserve:
\[\{(s,c),u,(s',c')\in \text{trcl } \text{tr } fg; \text{valid } fg \{\# s\} + c\} \implies \text{valid } fg \text{ (add-mset } s' c')

by (rule gtrp-preserve[where P=valid fg]) (auto dest: trss-valid-preserve-s)
7.6.2 Equivalence to reference point

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

lemma refpoint-eq-s: valid fg c \implies ((c, w, e') \in trcl (refpoint fg)) \iff ((c, w, e') \in tr (tr fg))

proof

have ((c, w, e') \in trcl (refpoint fg)) \implies valid fg c \implies ((c, w, e') \in trcl (tr fg))

by (induct rule: trcl.induct) (auto simp add: refpoint-eq-s tr-valid-preserve-s)

moreover have ((c, w, e') \in trcl (tr fg)) \implies valid fg c \implies ((c, w, e') \in trcl (refpoint fg))

by (induct rule: trcl.induct) (auto simp add: refpoint-eq-s tr-valid-preserve-s)

ultimately show valid fg c \implies ((c, w, e') \in trcl (refpoint fg)) = ((c, w, e') \in trcl (tr fg)) ..

qed

7.6.3 Case distinctions

lemma trss-c-cases-s[cases set, case-names no-spawn spawn]: [

((s, e, (s', e')) \in trss fg; 
[ c' = c ] \implies P;

!!p u v. [ e=LSpawn p; (u, Spawn p, v) \in edges fg; 
hd s = u; hd s' = v; c' = \{ \# [ entry fg p ] \#] + c \}] \implies P

] \implies P

by (auto elim!: trss.cases)

lemma trss-c-fmt-s: [((s, e, (s', e')) \in trss fg]

\implies \exists csp. c' = csp + c \land

(csp = {}) \lor (\exists p. e=LSpawn p \land csp = \{ \# [ entry fg p ] \#\})

by (force elim!: trss.c-cases-s)

lemma (in flowgraph) trss-c'-split-s: [

((s, e, (s', e')) \in trss fg;

!!csp. [ c' = csp + c; mon-c fg csp = {} ] \implies P

] \implies P

apply (erule trss-c-cases-s)

apply (subgoal-tac c' = \{ \# \} + c)

apply (fastforce)

apply auto

done

lemma trss-c-cases[cases set, case-names c-case]: !!s c. [

((s, c), w, (s', c')) \in trcl (trss fg);

]
!!csp. \[ c' = csp + c \; \text{!!s} \; s \in csp \implies \exists p \; u \; v. \; s = \text{entry fg p} \land (u, \text{spawn p, v}) \in \text{edges fg} \land \text{initialproc fg p} \]

\[ \implies P \]

proof (induct w)

next

case (Cons c w)

then obtain sh ch where SPLIT1: \((s, c), c \in (s, ch) \in \text{trss fg} \land \text{SPLIT2:} ((s, ch), w, (s', c')) \in \text{trcl (trss fg)} \by \text{fastforce dest: trcl-uncs})

from SPLIT2 show ?case proof (rule IHP(1))

fix csp

assume C’FMT: \( c' = csp + c \) and CSPFMT: \( \text{!!s. s \in csp} \implies \exists p \; u \; v. \; s = \text{entry fg p} \land (u, \text{spawn p, v}) \in \text{edges fg} \land \text{initialproc fg p} \)

from SPLIT1 show ?thesis

proof (rule trss-c-cases-s)

assume ch=c with C’FMT CSPFMT IHP(3) show ?case by blast

next

fix p

assume EFMT: \( c = \text{Lspawn p} \) and CHFMT: \( c = (\text{entry fg p}) + c \)

with C’FMT have \( c' = (\text{entry fg p}) + c + c \) by (simp add: union-ac)

moreover

from EFMT SPLIT1 have \( \exists u \; v. \; (u, \text{spawn p, v}) \in \text{edges fg} \) by (blast elim!: trss_cases)

hence \( \text{!!s. s \in (\text{entry fg p}) + csp} \implies \exists p \; u \; v. \; s = \text{entry fg p} \land (u, \text{spawn p, v}) \in \text{edges fg} \land \text{initialproc fg p} \) using CSPFMT by (unfold initialproc-def, erule-tac met-un-cases) (auto)

ultimately show ?case using IHP(3) by blast

qed

qed

lemma (in flowgraph) c-of-initial-no-mon:

assumes A: \( \text{!!s. s \in csp} \implies \exists p. \; s = \text{entry fg p} \land \text{initialproc fg p} \)

shows mon-c fg csp = \{\}

by (unfold mon-c-def) (auto dest: A initial-no-mon)

lemma (in flowgraph) trss-c-no-mon-s:

assumes A: \((s, c), c \in (s', c') \in \text{trss fg}\)

shows mon-c fg c' = mon-c fg c

using A

proof (erule-tac trss-c-cases-s)

assume c'=c thus ?thesis by simp

next

40
fix p assume EFMT: c=LSpawn p and C’FMT: c’=#{entry fg p}# + c from EFMT obtain u v where (u,Spawn p,v)∈edges fg using A by (auto elim: trss.cases)
with spawn-no-mon have mon-c fg {#{entry fg p}#} = {} by simp
with C’FMT show thesis by (simp add: mon-c-unconc)
qed

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

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

lemma trss-no-empty-s[simp]: ([],c,w,(s’,c’))∈trss fg = False by (auto elim!: trss.cases)

lemma trss-no-empty[simp]:
assumes A: ([],c,w,(s’,c’))∈trcl (trss fg)
shows w=[] ∧ s’=[] ∧ c=c’
proof –
note A
moreover { fix s have (s,c,w,(s’,c’))∈trcl (trss fg) ⇒ s=[ ] ⇒ w=[] ∧ s’=[] ∧ c=c’
by (induct rule: trcl-pair-induct) auto }
ultimately show thesis by blast
qed

lemma trs-step-cases[cases set, case-names NO-SPAWN SPAWN]:
assumes A: (c,c’,c’)∈tr fg
assumes A-NO-SPAWN: !!s ce s’ csp. []
\((s,ce),(s',ce')\) ∈ trss fg;
\(c = \{\#s\} + ce; c' = \{\#s'\} + ce\)
\[\rightarrow P\]

assumes A-SPAWN: \(!! \text{s ce s}'\ p. \]
\((s,ce), L\text{Spawn } p,(s',\{\#\text{entry fg p}\}\#) + ce)) \in trss fg;
\(c = \{\#s\} + ce; c' = \{\#s'\} + \{\#\text{entry fg p}\}\# + ce;\)
\(e = L\text{Spawn } p\)
\[\rightarrow P\]

shows \(P\)

proof –
from A show \(?thesis\) proof (erule-tac gtr-find-thread)
fix s ce s' ce'
assume FMT: \(c = \text{add-mset s ce c'} = \text{add-mset s' ce}'\)
assume B: \((s, ce), e, s', ce') \in trss fg\) thus \(?thesis\) proof (cases rule: trss-c-cases-s)

case no-spawn thus \(?thesis\) using FMT B by (－) (rule A-NO-SPA WN, auto)

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

7.7 Advanced properties

7.7.1 Stack composition / decomposition

lemma trss-stack-comp-s:
\((s,c),e,(s',c')\) ∈ trss fg \(\rightarrow (s@r,c),e,(s'@r,c'))\) ∈ trss fg
by (auto elim!: trss.cases intro: trss.intro)

lemma trss-stack-comp:
\((s,c),w,(s',c')\) ∈ trcl (trss fg) \(\rightarrow (s@r,c),w,(s'@r,c')\) ∈ 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 \oplus r, c), e, sh \oplus r, ch) \in trss fg\ .
also note IHP(3)
finally show \(?case\) .
qed

lemma trss-stack-decomp-s: \[ (s@r,c),e,(s',c')\) ∈ trss fg; s\#\]
\(\Rightarrow \exists s'. s' = s@r \land ((s,c),e,(s',c'))\) ∈ trss fg
by (cases s, simp) (auto intro: trss.intros elim!: trss.cases)

lemma trss-find-return:
\((s@r,c),w,(r,c')\) ∈ trcl (trss fg);
\[ \forall w, wb, ch. \ [ w = wa \cdot wb; ((s, c), w, ([], ch)) \in \text{trcl} (\text{trss} fg); (r, ch), wb, (r', c') \in \text{trcl} (\text{trss} fg) ] \implies P \]

If \( s = [], \) the proposition follows trivially
apply \( \text{cases } s=[] \)
apply fastforce
proof  
  — For \( s \neq [], \) we use induction by \( w \)
have \( !s. \ [ ((s@r, c), w, (r, c')) \in \text{trcl} (\text{trss} fg); s \neq [] ] \implies \exists wa, wb, ch. w = wa \cdot wb \wedge ((s, c), w, ([], ch)) \in \text{trcl} (\text{trss} fg) \wedge ((r, ch), wb, (r', c')) \in \text{trcl} (\text{trss} fg) \)
proo\( f (\text{induct } w) \)
case \( \text{Nil} \) thus \( ?\text{case by (auto)} \)
next
case \( \text{Cons } c \ w \) note \( \text{IHP=this} \)
then obtain \( sb, ch \) where \( \text{SPLIT1} \): \( ((s@r, c), e, (sh, ch)) \in \text{trss} fg \) and \( \text{SPLIT2} \): \( ((sh, ch), w, (r, c')) \in \text{trcl} (\text{trss} fg) \) by \( \text{fast force: trcl-uncons} \)
  \{ assume \( \text{CASE: } c = \text{LR} \)
  with \( \text{SPLIT1} \) obtain \( p \) where \( \text{EDGE: } s@r = \text{return } fg \) \( p \neq \) \( sh = \text{ch} \) by \( \text{auto elim!: trss\_cases} \)
  with \( \text{IHP(3)} \) obtain \( ss \) where \( \text{SHFMT: } s = \text{return } fg \) \( p \neq \) \( ss = ss@r \) by \( \text{cases } s, \text{auto} \)
  \{ assume \( \text{CC: } ss \neq [] \)
  with \( \text{SHFMT} \) have \( \exists ss. ss \neq [] \wedge sh = ss@r \) by \( \text{blast} \)
  } moreover \{
  assume \( \text{CC: } ss = [] \)
  with \( \text{CASE SHFMT EDGE} \) have \( ((s, c), [e], ([], ch)) \in \text{trcl} (\text{trss} fg) \) \( c\# w = [e]@w \)
  by \( \text{auto intro: trss\_ret} \)
  moreover from \( \text{SPLIT2 SHFMT CC} \) have \( ((r, ch), w, (r, c')) \in \text{trcl} (\text{trss} fg) \)
  by \( \text{simp} \)
  ultimately have \( ?\text{case by blast} \)
  } ultimately have \( ?\text{case } (\exists ss. ss \neq [] \wedge sh = ss@r) \) by \( \text{blast} \)
  } moreover \{
  assume \( e \neq \text{LR} \)
  with \( \text{SPLIT1 IHP(3)} \) have \( \exists ss. ss \neq [] \wedge sh = ss@r \) by \( \text{force elim!: trss\_cases simp add: append-eq-Cons-cone} \)
  \}
  moreover \{
  assume \( \exists ss. ss \neq [] \wedge sh = ss@r \)
  then obtain \( ss \) where \( \text{CASE: } ss \neq [] \) \( sh = ss@r \) by \( \text{blast} \)
  with \( \text{SPLIT2} \) have \( ((ss@r, c), w, r, c') \in \text{trcl} (\text{trss} fg) \) by \( \text{simp} \)
  from \( \text{IHP(1)}(\text{OF this CASE(1)}) \) obtain \( wa, wb, ch' \) where \( \text{IHAPP: } w = wa \cdot wb \)
  with \( \text{CASE SPLIT1} \) have \( ((s@r, c), e, ss@r, ch) \in \text{trss} fg \) by \( \text{simp} \)
  from \( \text{trss\_stack\_decomp-s[OF this IHP(3)]} \) have \( ((s, c), e, ss, ch) \in \text{trss} fg \)
  by \( \text{auto} \)
  with \( \text{IHAPP} \) have \( ((s, c), \# e @ wa, ([], ch')) \in \text{trcl} (\text{trss} fg) \) by \( \text{rule-tac trcl\_cons} \)
  moreover from \( \text{IHAPP} \) have \( e \# w = (e\# wa)@wb \) by \( \text{auto} \)
  ultimately have \( ?\text{case by blast} \)
  } ultimately show \( ?\text{case by blast} \)

43
qed

assume \((s \circ r, c), w, r, c') \in \text{trcl} (\text{trss } fg) s \neq [] \Rightarrow \text{wa \ wb \ ch}. w = wa \circ wb; ((s,c),wa,[[],ch)] \in \text{trcl} (\text{trss } fg); ((r,ch),wb,(r,c')) \in \text{trcl} (\text{trss } fg) \Rightarrow P \text{ thus } P \text{ by (blast dest: IM)}

qed

lemma \text{trss-return-cases[cases set]}: \Rightarrow \text{cases intro: trss.intro}

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

next

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

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

\{ fix ba q

assume CASE: \(e=\text{LBase} ba \lor e=\text{LSpawn} q

with SPLIT1 obtain v where E: sh=v\#r \(((u],c),e,(v],ch)] \in \text{trss } fg \text{ by (auto elim!: trss.cases intro: trss.intro)}

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

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

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

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

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

next

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

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

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

qed

moreover \{

assume CASE: \(e=\text{LRet)

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

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

moreover \{

fix q

assume CASE: \(e=\text{LCall} q

with SPLIT1 obtain u' where SHFMT: sh=\text{entry } fg q \# u' \# r \(((u],c),e,(\text{entry } fg q \# [u'],ch)] \in \text{trss } fg \text{ by (auto elim!: trss.cases intro: trss.intro)

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

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

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

from \text{trss-stack-comp[OF CC(2), where r=[u']] have \(((entry } fg q\#[u'], ch),

qed
\[ w, (st \circ [ul]) \circ [u', c'] \in \text{trcl}(\text{trss } fg) \text{ by auto} \]

with \text{SHFMT}(2) have \(((\{u\}, c), e \# w, (st \circ [ul]) \circ [u', c'] \in \text{trcl}(\text{trss } fg) \text{ by auto})

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

next

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

\text{from trss-stack-comp}[OF CC(2), \text{where } r=[u']] \text{ have } ((\text{entry } fg \ # \ [w'], ch), wa, [u'], ct) \in \text{trcl}(\text{trss } fg) \text{ by simp}

\text{with } \text{SHFMT} \text{ have PREPATH: } ((\{u\}, c), e \# wa, [u', ct] \in \text{trcl}(\text{trss } fg) \text{ by simp})

\text{from } CC \text{ have } L: \text{ length } wb \leq \text{ length } w \text{ by simp}

\text{from } CC(3) \text{ show } ?\text{case proof (cases rule: IHP(1)[OF L, cases set])}

\text{case (1 s'' u') note } CCC=\text{this from } \text{trcl-concat}[OF PREPATH CCC(2)]

\text{CC(1) have } ((\{u\}, c), e \# w = (e \# wa \circ wb) \circ [u'] (\{u\}, c \ # wa \ @ wb, [\], c') \in \text{trcl}(\text{trss } fg) \text{ by simp})

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

\text{next}

\text{case (2 wba wbb c') note } CCC=\text{this from } \text{trcl-concat}[OF PREPATH CCC(2)] \text{ CC(1) CCC(1) have } e \# w = (e \# wa \circ wb) \circ [u'] \text{ (\{u\}, c \ # wa \ @ wba, [\], c') \in \text{trcl}(\text{trss } fg) \text{ by auto})}

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

\text{qed}

\text{qed}

\text{lemma (in flowgraph) trss-find-call:}

\[ \forall v \ r \ c'. \ [ (([sp], c), w,(v \# r',cE)) \in \text{trcl}(\text{trss } fg); \ r' \neq [] ] \]

\[ \Rightarrow \exists rh \ ch \ p \ wa \ wb. \]

\[ w=wa \circ (\text{LCall } p) \# wb \land \]

\[ \text{proc-of } fg \ v = p \land \]

\[ (([sp], c), wa,(rh, ch)) \in \text{trcl}(\text{trss } fg) \land \]

\[ ((rh, ch), \text{LCall } p,((\text{entry } fg \ p) \# r',ch)) \in \text{trss } fg \land \]

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

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

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

\text{next}

\text{case (snoc } w \ c \text{) note } \text{IHP=\text{this}}

\text{then obtain } rh \ ch \text{ where } SPLIT1: (([sp], c),w,(rh, ch)) \in \text{trcl}(\text{trss } fg) \text{ and SPLIT2:} ((rh, ch),c,(v \# r',c')) \in \text{trss } fg \text{ by (fast dest: trcl-rev-uncons)}

\{

\text{assume } \exists u. \ rh=u \# r'

\text{then obtain } u \text{ where } RHFMT[simp]: rh=u \# r' \text{ by blast}

\text{with SPLIT2 have } \text{proc-of } fg \ u = \text{proc-of } fg \ v \text{ by (auto elim: trss.cases intro: edges-part)}

\text{moreover from IHP(1)[of } w \ u \ r' \ ch, \text{ OF - SPLIT1[simplified] IHP(3)] obtain} rt ct p \ wa \ wb \text{ where}

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

\]
(\{(entry \text{fg} p), ct\}, \text{wb}, ([u], ch)) \in \text{trcl} (\text{trss f}g) \text{ by (blast)}

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

note \text{IHAPP'(5)}

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

finally show \text{thesis} .

qed

moreover from \text{IHAPP} have \text{wa}@[e] = \text{wa} \oslash \text{LCall} p \# (\text{wb}@[e]) \text{ by auto}

ultimately have \text{?case by auto}

}

moreover have (\exists u. \text{rh}=u\#r') \lor \text{?case}

proof (rule trss_cases[OF \text{SPLIT2}], simp-all, goal-cases) — Cases for base- and spawn edge are discharged automatically

— Case: call-edge

case \{1 ca \ p \ r \ u \ vv\} \text{ with \text{SPLIT1 SPLIT2} show \text{?case by fastforce} next}

— Case: return edge

case \text{CC}: \{2 q \ r \ ca\}

hence [simp]: \text{rh}=(\text{return f}g \ q)\#v\#r' \text{ by simp}

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

\text{IHPP: w = wa @ LCall p \# wb (([sp], c), wa, rt, ct) \in \text{trcl (trss f}g) ((rt, ct), LCall q, entry f}g q \# v \# r', ct) \in \text{trss f}g

(((entry f}g q), ct), \text{wb}, (\text{return f}g q), ch) \in \text{trcl (trss f}g) \text{ by force}

then obtain u \text{ where RTFMT [simp]: rt}=u\#r' \text{ and PROC-OF-U: proc-of f}g u = \text{proc-of f}g v \text{ by (auto elim: trss_cases intro: edges-part)}

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

from \text{IHPP(1)}\{OF LENWA \text{ IHAPP}(2)\text{simplified} \text{ IHPP(3)] obtain rhh chh p waa wab where}

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

(((entry f}g p),chh),wa,([u],ct))\in \text{trcl (trss f}g)

by \text{blast}

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

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

note \text{IHAPP'(5)}

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

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

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

finally show \text{?thesis by simp}

qed

moreover note \text{IHAPP' CC}

ultimately show \text{?case by auto}
— 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) 
  ([(rh, ch), LCall p, [entry fg p, u'], ch)] ∈ trss fg 
  ([(entry fg p], ch), wb, [return fg p], c') ∈ trcl (trss fg) 
  by auto

mor eover 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) 

implies proc-of fg u = proc-of fg u'

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

**case Nil thus ?case by auto**

next

**case** (snoc v u) note IHP=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-uncons)

from SPLIT2 have sh#[] by (auto elim!: trss.cases)

then obtain ssh uh where SHFMT: ssh=ssh@[uh] by (blast dest: list-rev-decomp)

with IHP(1)[of ssh uh ch] SPLIT1 have proc-of fg u = proc-of fg uh by auto

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)+

finally show ?case .

**qed**

— Specialized version of flowgraph.trss-bot-proc-const that comes in handy for precision proofs of constraint systems

**lemma** (in flowgraph) trss-er-path-proc-const:

((entry fg p],c),w,[(return fg q],c'))∈trcl (trss fg) ⟹ p=q

using trss-bot-proc-const[of [] entry fg p - [] return fg q, simplified].
lemma \texttt{trss-2empty-to-2return}: \[ ((s, c), w, ([], c')) \in \text{trcl} (\text{trss} \ fg) \; ; \; s \neq [] \implies \exists w' \; p. \; w = w' @ [LRet] \wedge ((s, c), w', ([\text{return} \ fg \ p], c')) \in \text{trcl} (\text{trss} \ fg) \]

\textbf{proof} –

\begin{itemize}
  \item assume \( A: ((s, c), w, ([], c')) \in \text{trcl} (\text{trss} \ fg) \; ; \; s \neq [] \)
  \item hence \( w \neq [] \) by \text{auto}
  \item then obtain \( w' \in e \) where \( WD: w = w' @ [e] \) by (\text{blast dest: list-decomp})
\end{itemize}

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

\textbf{thus} \( (s', c) \in ch=c' \) by (\text{cases rule: trss.cases, auto})

\textbf{with} \( \text{SPLIT}(1) \) \( WD \) show \( ?\text{thesis} \) by \text{blast}

qed

\begin{itemize}
  \item lemma \texttt{trss-2return-to-2empty}: \[ ((s, c), w, ([\text{return} \ fg \ p], c')) \in \text{trcl} (\text{trss} \ fg) \]
  \item apply (\text{subgoal-tac} \; (([\text{return} \ fg \ p], c'), LRet, ([], c')) \in \text{trss} \ fg) \]
  \item by (\text{auto dest: trcl-rev-cons intro: trss.intros})
\end{itemize}

7.7.2 Adding threads

\begin{itemize}
  \item lemma \texttt{trss-env-increasing-s}: \[ ((s, c), w, ([\text{return} \ fg \ p], c')) \in \text{trss} \ fg \implies c \subseteq #\; c' \]
  \item by (\text{auto elim!: trss.cases})
\end{itemize}

\begin{itemize}
  \item lemma \texttt{trss-env-increasing}: \[ ((s, c), w, (s', c')) \in \text{trcl} (\text{trss} \ fg) \implies c \subseteq #\; c' \]
  \item by (\text{induct rule: trcl-pair-induct}) (\text{auto dest: trss-env-increasing-s order-trans})
\end{itemize}

7.7.3 Conversion between environment and monitor restrictions

\begin{itemize}
  \item lemma \texttt{trss-mon-e-no-ctx}:
    \[ ((s, c), e, (s', c')) \in \text{trss} \ fg \implies \text{mon-e} \; e \cap \text{mon-c} \; fg \; c = {} \]
  \item by (\text{erule trss.cases})
\end{itemize}

\begin{itemize}
  \item lemma \texttt{(in flowgraph) trss-mon-w-no-ctx}:
    \[ ((s, c), w, (s', c')) \in \text{trcl} (\text{trss} \ fg) \implies \text{mon-w} \; fg \; w \cap \text{mon-c} \; fg \; c = {} \]
  \item by (\text{induct rule: trcl-pair-induct}) (\text{auto dest: trss-mon-e-no-ctx simp add: trss-c-no-mon-s})
\end{itemize}

\begin{itemize}
  \item lemma \texttt{(in flowgraph) trss-modify-context-s}:
    \[ \forall cn. \; \forall c. (((s, c), e, (s', c')) \in \text{trss} \ fg; \; \text{mon-e} \; e \cap \text{mon-c} \; fg \; cn = {}) \]
    \[ \implies \exists \text{csp}. \; c' = \text{csp} + c \wedge \text{mon-c} \; fg \; csp = {} \wedge ((s, cn), e, (s', csp + cn)) \in \text{trss} \ fg \]
  \item by (\text{erule trss.cases}) (\text{auto intro!: trss.intros})
\end{itemize}

\begin{itemize}
  \item lemma \texttt{(in flowgraph) trss-modify-context} (\text{rule-formal}):
    \[ (((s, c), w, (s', c')) \in \text{trcl} (\text{trss} \ fg)) \]
    \[ \implies \forall cn. \; \forall \text{mon-w} \; fg \; w \cap \text{mon-c} \; fg \; cn = {} \]
    \[ \implies \exists \text{csp}. \; c' = \text{csp} + c \wedge \text{mon-c} \; fg \; csp = {} \wedge ((s, cn), w, (s', csp + cn)) \in \text{trcl} (\text{trss} \ fg)) \]
  \item proof (\text{induct rule: trcl-pair-induct})
  \item case empty thus \( ?\text{case} \) by \text{simp}
\end{itemize}

next

\begin{itemize}
  \item case \( (\text{cons s c e sh ch w s' c'}) \) \textbf{note} \( \text{IHP} = \text{this show} \; ?\text{case} \)
  \item proof (\text{intro allI impI})
\end{itemize}
fix cn
assume MON: mon-w fg (e ≠ w) ∧ mon-c fg cn = {}
from trss-modify-context-s[OF IHP(1)] MON obtain csph where S1: ch = csph + c mon-c fg csph = {} ((s, cn), e, sh, csph + cn) ∈ trss fg by auto
with MON have mon-w fg w ∩ mon-c fg (csph + cn) = {} by (auto simp add: mon-c-unconc)
with IHP(3)[rule-format] obtain esp where S2: e' = esp + ch mon-c fg esp = {} ((sh, csph + cn), w, (s', esp + (csph + cn))) ∈ trcl (trss fg) by blast
from S1 S2 have e' = (esp + csph) + c mon-c fg (esp + csph) = {} ((s, cn), e ≠ w, (s', esp + cn)) ∈ trcl (trss fg) by (auto simp add: union-assoc mon-c-unconc)
thus ∃ esp. e' = esp + c ∧ mon-c fg esp = {} ∧ ((s, cn), e ≠ w, s', esp + cn) ∈ trcl (trss fg) by blast
qed
lemma trss-add-context-s:
[(s, e, (s', c')) ∈ trss fg; mon-e fg e ∩ mon-c fg ec = {}]
⇒ ((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 ec = {} 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 .
show qed

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

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

moreover from CHFMT IHP(1) SPLIT(2) have ((sh,csp+c),w,(s',c'))∈trcl (trss fg) mon-w fg w ∩ mon-c fg ce = {} by blast+

ultimately show ?thesis by auto

done

lemma trss-xchange-context-s:
  assumes A: ((s,c),e,(s',csp+c))∈trcl (trss fg)
  and M:mon-c fg cn ⊆ mon-c fg c
  shows ((s,cn),e,(s',csp+cn))∈trss fg

proof –
  from trss-drop-context-s[of - {#}, simplified, OF A] have DC: ((s, {#}), e, s',
csp) ∈ trss fg mon-e fg e ∩ mon-c fg c = {} by simp-all
  with M have mon-e fg e ∩ mon-c fg cn = {} by auto
  from trss-add-context-s[OF DC(1)] this] show ?thesis by auto

done

lemma trss-xchange-context:
  assumes A: ((s,c),w,(s',csp+c))∈trcl (trss fg)
  and M:mon-c fg cn ⊆ mon-c fg c
  shows ((s,cn),w,(s',csp+cn))∈trcl (trss fg)

proof –
  from trss-drop-context[of - {#}, simplified, OF A] have DC: ((s, {#}), w, s',
csp) ∈ trcl (trss fg) mon-w fg w ∩ mon-c fg c = {} by simp-all
  with M have mon-w fg w ∩ mon-c fg cn = {} by auto
  from trss-add-context[of DC(1)] this] show ?thesis by auto

done

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

  done

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

  done
lemma \textbf{tr-add-context-s}:
\[
\begin{array}{l}
\forall (c, e, c') \in \text{tr \_ \_ f}_g; \ \text{mon \_ \_ e} \ f_g \cap \text{mon \_ \_ c} \ f_g \ c_e = \{\} \implies (c + c_e, e, c' + c_e) \in \text{tr \_ \_ f}_g \\
\text{by (erule gtrE) (auto simp add: mon-c-unconc union-assoc intro: gtrI-s dest: trss-add-context-s)}
\end{array}
\]

lemma \textbf{tr-add-context}:
\[
\begin{array}{l}
\forall (c, w, c') \in \text{trcl (tr \_ \_ f}_g); \ \text{mon \_ \_ w} \ f_g \ w \cap \text{mon \_ \_ c} \ f_g \ c_e = \{\} \\
\implies (c + c_e, w, c' + c_e) \in \text{trcl (tr \_ \_ f}_g)
\end{array}
\]

\textbf{proof (induct rule: trcl.induct)}
\begin{itemize}
\item \textbf{case empty thus ?case by auto}
\item \textbf{next}
\item \textbf{case (cons \_ \_ c} e c' w c'') \textbf{note IHP=this}
\item \textbf{from tr-add-context-s[OF IHP(1), of ce]} \textbf{IHP(4) have} \ (c + c_e, e, c' + c_e) \in \text{tr \_ \_ f}_g \text{ by auto}
\item \textbf{also from IHP(3,4) have} \ (c' + c_e, w, c'' + c_e) \in \text{trcl (tr \_ \_ f}_g) \text{ by auto}
\item \textbf{finally show ?case .}
\end{itemize}
\textbf{qed}

end

8 Normalized Paths

\textbf{theory Normalization}

\textbf{imports Main ThreadTracking Semantics ConsInterleave}

\textbf{begin}

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

8.1 Semantic properties of restricted flowgraphs

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

— Procedures with isolated return nodes will never return

\textbf{lemma (in eflowgraph) iso-ret-no-ret: !!u. c. [}
\begin{array}{l}
\text{isolated-ret f}_g p; \\
\text{proc-of f}_g u = p; \\
\text{u} \neq \text{return f}_g p; \\
((\{\{u, c\}, w, (\text{return f}_g p', c')\}) \in \text{trcl (trss f}_g)
\end{array}
\] 
\textbf{\implies False}

\textbf{proof (induct w rule: length-compl-induct)}
case Nil  thus \(\forall a. \text{trss}\) return-cases

next

\[\text{case (Cons } e\ w) \text{ note IHP} \Rightarrow \text{this} \]

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

\]

\[\text{show } \forall a. \text{trss}\) return-cases\]

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

next

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

\]

\[\text{with SPLIT2 show False by auto}\]

next

\[\text{case (LCall } q) \text{ with SPLIT1 IHP(2,3) obtain } uh\ \text{where } A: sh=\text{entry } fg\ q\#[uh]\ \text{proc-of } fg\ uh = p \ uh\text{\not return } fg p\ \text{by (force elim!)} \text{: trss.cases simp add: edges-part isolated-ret-def)}

\]

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

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

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

qed

qed

— The first step of an initial procedure is a call

\[\text{lemma (in eflowgraph) initial-starts-with-call:} \]

\[\exists p'\. \text{e=LCall } p'\ \text{\land isolated-ret } fg\ p'\]

\[\text{by (auto elim!)} \text{: trss.cases dest: initial-call-no-ret initial-no-ret entry-return-same-proc}\]

— There are no same-level paths starting from the entry node of an initial procedure

\[\text{lemma (in eflowgraph) no-sl-from-initial:} \]

\[\text{assumes } A: w\text{=}[]\ \text{\textsf{\textnot} initialproc } fg\ p\]

\[\text{shows } False \]

\[\text{proof} \]

\[\text{from A obtain } sh \ ch \ e \ w' \text{ where SPLIT: } (([e],c),e,(sh,ch)) \in \text{trss } fg\ ((sh,ch),w',([e],c')) \in \text{trcl (trss } fg) \text{ by (cases w, simp, fast dest: trcl-uncons)} \]

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

\[\text{with SPLIT(1) obtain } u' \text{ where sh=entry } fg\ p'\#[u']\ \text{by (auto elim!)} \text{: trss.cases} \]

\[\text{with SPLIT(2) have } (([e],c'),e',([e],c')) \in \text{trcl (trss } fg) \text{ by simp}\]

\[\text{then obtain } wa \ ct \text{ where } (([e],c'),wa,(\text{entry } fg\ p',ch),w1,(\text{entry } fg\ p',ct)) \in \text{trcl (trss } fg) \text{ by (erule-tac trss-return-cases, auto)} \]

\[\text{then obtain } wa' p'' \text{ where } (([e],c'),wa',([e],ct)) \in \text{trcl (trss} \]

52
fg) by (blast dest: trss-2empty-to-2return)

from iso-ret-no-ret[OF CE(2) - this] CE(2)[unfolded isolated-ret-def] show ?thesis by simp

qed

— There are no same-level or returning paths starting from the entry node of an initial procedure

lemma (in eflowgraph) no-retsl-from-initial:
  assumes A: w̸=[]
  initialproc fg p (((entry fg p),c),w,(r,c'))∈trcl (trss fg)
  length r' ≤ 1
  shows False

proof (cases r')
  case Nil with A(3) have (((entry fg p),c),w,([]),c'))∈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') ∈ trcl (trss fg) by (blast)
  show ?thesis proof (cases w')
    case Nil with B have p=q entry fg p = return fg p by (auto dest: trcl-empty-cons entry-return-same-proc)
    with A(2) initial-no-ret show False by blast
  next
    case Cons hence w'̸=[] by simp
    from no-sl-from-initial[OF this A(2) B(2)] show False .
  qed
  next
    case (Cons u rr) with A(4) have r'==[u] by auto
    with no-sl-from-initial[OF A(1,2)] A(3) show False by auto
  qed

8.2 Definition of normalized paths

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

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

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

\(^2\)Same-level paths are paths with balanced calls and returns. The stack-level at the beginning of their execution is the same as at the end, and during the execution, the stack never falls below the initial level.
to derive interleaving semantics on multisets and configurations with an explicit local thread (loc/env-semantics, cf. Section 5.4).

**inductive-set**

\[
ntrs :: (\langle n, p, ba, m, more \rangle) \Rightarrow (\langle n \text{ list } \times \langle n \text{ conf} \rangle \rangle \times \langle p, ba \rangle \text{ label list } \times (\langle n \text{ list } \times \langle n \text{ conf} \rangle \rangle) \text{ set for } fg\]

**where**

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

\[
ntrs-step: (\langle (u \# r, ce), LCall p, (entry fg p \# u' \# r, ce) \rangle) \in \text{trss } fg; \]

\[
\quad (\langle [entry fg p], ce, w, ([v], ce') \rangle) \in \text{trcl (trss } fg) \quad \Rightarrow \]

\[
\quad (\langle (u \# r, ce), LCall p \# w, (v' \# u \# r, ce') \rangle) \in \text{ntrs } fg\]

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

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

**interpretation** ntrs: env-no-step ntrs fg

**apply** (rule env-no-step.intra)

**apply** (erule ntrs.cases)

**apply** clarsimp

**apply** (erule trss-c-cases)

**apply** auto

**done**

### 8.3 Representation property for reachable configurations

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

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

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

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

**fix** p r u u' v w

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

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

**by** simp

**with** A(1) **show** (\langle u \# r, c \rangle, LCall p \# w, v \# u' \# r, c') \in \text{trcl (trss } fg)\]

**by** auto

**qed**

**lemma** ntrs-is-trss: (\langle s, c, w, (s', c') \rangle) \in \text{trcl (ntrs } fg) \Rightarrow (\langle s, c, foldl (@) \ [w, (s', c')] \rangle) \in \text{trcl (trss } fg)\]

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

**case** empty **thus** ?case by simp
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 than 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.
\[ \exists \, w\'. \{ \# \text{entry } fg \, p \} \#, \, w\wedge \#, \{ \# \, r \# \} + ce' \} \in \text{trcl} \left( \text{ntr } fg \right) \]

**proof** (induct \( wu \) rule: rev-induct)

- **case** \( \text{Nil} \) **note** \( CC=\text{this} \) **hence** \( u=\text{entry } fg \, p \) **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**[\( \text{OF } CC(5,3) - CC(4) \)] \( \text{have False by blast} \)

- **thus** ?case ..

- **next**

  - **case** \( \text{snoc } ee \, wu \) **note** \( \text{IHP=this} \)

    — In the induction step, we extract the last macrostep

    *then* **obtain** \( ch \) **where** \( \text{SPLIT} : \{ \#, \text{entry } fg \, p \} \#, wu, ch \} \in \text{trcl} \left( \text{ntr } fg \right) \left( ch, ee, \{ \#, u\#r \# \} + ce \right) \in \text{ntr } fg \) **by** (fast dest: trcl-rev-uncons)

    — The last macrostep first executes a call and then a same-level path

    *from** \( \text{SPLIT}(2) \) **obtain** \( q \, wws \) \& \( rh \) \& \( ch \) \& \( vt \) \& \( cet \) **where**

      \( \text{STEPFMT: } cc=\text{LCall } q \# wws \, ch=\text{add-mset} \left( u\#rh \right) \, ceh \text{ add-mset} \left( u\#r \right) \, ce=\text{add-mset} \left( vt\#uh\#rh \right) \, cet=\text{add-mset} \left( uh\#rh, ceh \right), \text{LCall } q, \text{(entry } fg \, q \# uh\#rh, ceh \right) \} \in \text{trss } fg \)

      \( \left[ \left( \text{entry } fg \, q \right), ceh, wws, \left( \left( vt, cet \right) \right) \right] \in \text{trcl} \left( \text{trss } fg \right) \)

      **by** (auto simp: trcl-concat)

      — Make a case distinction whether the last step was executed on the same thread as the sl/ret-path or not

    *from** \( \text{STEPFMT}(3) \) **show** \( \text{?thesis} \) **proof** (cases rule: mset-single-cases')

      — If the sl/ret path was executed on the same thread as the last macrostep

      *case** \( \text{loc} \) **note** \( \text{CASE=this} \) **hence** \( C' : u=vt \) \& \( r=uh\#rh \) \& \( ce=\text{cet} \) **by** auto

      — We append it to the last macrostep.

    *with** \( \text{STEPFMT}(5) \) **IHP(3) have** \( \text{NEWPATH: } \left[ \left( \text{entry } fg \, q \right), ceh, wws@w, \left( r', ce' \right) \right] \in \text{trcl} \left( \text{trss } fg \right) \)

      **by** (simp add: trcl-concat)

      — We then distinguish whether we appended a same-level or a returning path

    *show** \( \text{?thesis} \) **proof** (cases \( r' \))

      — If we appended a same-level path

      *case** \( \text{Cons } v' \) — Same-level path **with** \( \text{IHP(5)} \) **have** \( CC: r'=\left[ v' \right] \) **by** auto

      — The macrostep still ends with a same-level path

    *with** \( \text{NEWPATH have} \left( \left( \text{entry } fg \, q \right), ceh, wws@w, \left( \left[ v' \right], ce' \right) \right) \in \text{trcl} \left( \text{trss } fg \right) \)

      **by** simp

      — and thus remains a valid macrostep

    *from** \( \text{gtr-s[OF ntr-step[OF STEPFMT(4), simplified, OF this]] have} \)

      \( \left( \text{add-mset} \left( uh \# rh \right) \, ceh, \text{LCall } q \# wws@w, \text{add-mset} \left( v' \# uh' \# rh \right) \, ce \right) \in \text{ntr } fg \).

      — that we can append to the prefix of the normalized path to get our proposition

    *with** \( \text{STEPFMT(2) SPLIT(1) CC C'(2) have} \left( \#, \text{entry } fg \, p \right) \#, wu\#LCall q\#wus@w, \left( \#, r'\#r \# \right) \) \& \( ce' \) \& \( \text{trcl} \left( \text{ntr } fg \right) \)

      **by** (auto simp add: trcl-rev-cons)

      — **thus** ?thesis by blast

- **next**

  — If we appended a returning path

  *case** \( \text{Nil} \) **note** \( CC=\text{this} \)

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

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

  **proof**

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

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

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

qed

Hence we can apply the induction hypothesis and get the proposition

\[\text{from IHP(1)|OF - NEWSL} \text{ SPLIT STEPFM T(2) IHP(4) CC C'(2) show}\]

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

next

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

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

— we first look at the context after the last macrostep. It consists of the threads that already have been there and the threads that have been spawned by the last macrostep

\[\text{from STEP FM T(5) obtain cspt where CETFM T: cet=cspt+ceh !s, s} \in\#
\]

\[\text{cspt} \Rightarrow \exists p.\ s=[\text{entry}\ fg\ p] \land \text{initialproc}\ fg\ p\]

by (unfold initialproc-def) (erule trss-c-cases, blast)

— The spawned threads do not hold any monitors yet

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

— We now distinguish whether the sl/ret path is executed on a thread that was just spawned or on a thread that was already there

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

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

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

\[\text{case left — Thread was spawned with CETFM T obtain q where u}=\text{entry}\]

\[\text{fg}\ q\ r=\text{}\]\n
\[\text{by auto}\]

with IHP(3,5,6) no-retsl-from-initial have False by blast

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

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

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

proof —

from CEHFMT STEPFM T(2) have \(ch = \{# u\#rh #\} + (\{# u\#r #\}+\{ceh-{# u\#r #}\})\) by simp

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

qed

— There are not more monitors than after the last macrostep

have MON-CE: mon-c fg ((# u\#rh #)+\{ceh-{# u\#r #}\}) \subseteq mon-c fg cc proof —

have mon-n fg uh \subseteq mon-n fg uh' using STEPFM T(4) by (auto elim!: trss.cases dest: mon-n-same-proc edges-part)

moreover have mon-c fg \{ceh-{# u\#r #}\} \subseteq mon-c fg cc proof —
from CASE(3) CETFMT have cc=(cspt+ceh)−{#u#r#} by simp
cw CC have cc = cspt+(ceh−{#u#r#}) by auto
cw 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 esp' where CE’FMT: ce'=csp'+ce mon-c fg csp' = {} by (−) (crude trss-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-C] IHP(3) CE’FMT have NSL: (([u], {#uh # rh#}) + (ceh − {#u # r#})), w, r', esp' + ((#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 wu' where NNPATH: ([#entry fg p#], wu', (#r' @ r#) + (esp' + {#uh # rh#} + (ceh − {#u # r#}))) ∈ trcl (ntp fg) by blast
— We now show that the last macrostep can also be executed from the new configuration, after the sl/ret path has been executed (on another thread)
have {(#r' @ r#) + (esp' + {#uh # rh#} + (ceh − {#u # r#}))), ce, (#vt # uh' # rh#) + (cspt + {#r' @ r#} + (esp' + (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' @ r#) + (esp' + (ceh − {#u # r#}))) ⊆ mon-c fg ceh proof −
from IHP(3,5) trss-bot-proc-const[of [] u ce w [] - ce'] mon-n-same-proc
have mon-s fg r' ⊆ mon-n-same-proc fg 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-CE], where esp={#}, simplified, OF STEPFMT(4)
moreover from trss-xchange-context[OF - MON-CE, of [entry fg q] wuS
[vt] cspt] STEPFMT(3) CETFMT(1) have
(([entry fg q], {#r' @ r#} + (esp' + (ceh − {#u # r#}))), wus, [vt], cspt + {#r' @ r#} + (esp' + (ceh − {#u # r#}))) ∈ trcl (trss fg) by blast
moreover note STEPFMT(1)
ultimately have ((#u#rh, (#r' @ r#) + (esp' + (ceh − {#u # r#}))),wus,(vt)#uh#rh,cspt+(#r' @ r#) + (esp' + (ceh − {#u # r#}))))∈ntrS
fg by (auto intro: ntrs.intros)
from gtr1-s[OF this] show ?thesis by (simp add: add-mset-commute)
qed
— Finally we append the last macrostep to the normalized paths we obtained by the induction hypothesis.

From trcl-rev-cons[of NNPATH this] have \((\#[entry fg p]#, \{\#vt \# uh’ \# rh\#\} + (csp + (\#r’ @ r#\}) + (csp’ + (ceh - \{\#u \# r\#\}))}) \in \text{trcl}(ntr fg)\).

— And show that we got the right configuration

Moreover from CC CETFMT CASE(3)[symmetric] CASE(2) CE’FMT(1)

have \{\#vt \# uh’ \# rh\#\} + (csp + (\#r’ @ r#\}) + (csp’ + (ceh - \{\#u \# r\#\})) = \{\# r’@r \#\} + ce’ by \(\text{simp add: union-ac}\)

ultimately show \(\text{thesis by auto}\) qed

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

**Lemma (in eflowgraph) normalize:**

\(\text{normalize: } [\text{cstart},w,c] \in \text{trcl}(tr fg); \text{cstart} = \{\# \text{[entry fg p] \#}\}; \text{initialproc fg p} \]

\(\Rightarrow \exists w’. (\{\# \text{[entry fg p] \#}\}, w’,c) \in \text{trcl}(ntr fg) (c,e,c’) \in \text{tr fg} \) by blast

— 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 \(\text{thesis by auto}\) (auto intro: efl[of - []])**

Next

**Case (snoc cstart w v 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’ where \(\text{IHP’}: (\{\# \text{[entry fg p] \#}\},w’,c) \in \text{trcl}(ntr fg) (c,e,c’) \in \text{tr fg} \)**

by blast

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

From gtr-find-thread[of \(\text{IHP’(2)}\)] obtain \(s ce s’ ce’ \) where \(\text{TSTEP: } c = \text{add-mset} s ce c’ = \text{add-mset} s’ ce’ ((s, ce), e, (s’, ce’)) \in \text{trss fg} \) 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

**Fix q**

**Assume CASE: e=LCall q**

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

**Have (c,LCall q \# [], c’) \in ntr fg using TSTEP CASE by (auto elim!: trss.cases**
introl: ntrs.intros gtrI-s trss.intros
— That can be appended to the initial normalized path

from trcl-rev-cons[OF HLP(1) this] have "case by blast"

} moreover {
— Last step was no procedure call

fix q a

assume CASE: c=LLBase a ∨ c=LSpawn q ∨ c=LRet
— Then it is a same-level or returning path

with TSTEP(3) obtain u r r' where SLR: s=u#r s'=r#r length r' ≤ 1
(((u),c),(r',c'))∈trcl (trss fg) by (force elim!: trss.cases introl: trss.intros)
— That can be appended to the normalized path using the "[["##entry fg ?p#]],
??ww, {#?u # ?r#} + ?ce) ∈ trcl (ntr fg); (((u), ?ce), ?w, ?r', ?ce') ∈ trcl (trss fg);
initialproc fg ?p; length ?r' ≤ 1; ?w ≠ []] → ∃ ww'. "[["##entry fg ?p#]], ww',
{#?r' # ?r#} + ?ce') ∈ trcl (ntr fg) - lemma

from ntr-sl-move-lep[OF - SLR(4) HLP(5) SLR(3)] HLP(1) TSTEP(1) SLR(1)
obtain ww' where "[["##entry fg p#]], ww', {#?r # ?r#} + ce')) ∈ trcl (ntr fg) by auto

with SLR(2) TSTEP(2) have "case by auto"

} ultimately show "case by (cases e, auto)"

qed

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

theorem (in eflograph) ntr-repr:
(∃ w. ((#entry fg (main fg))#,w,c)∈trcl (tr fg))
iff (∃ w. ((#entry fg (main fg))#,w,c)∈trcl (ntr fg))

by (auto simp add: initialproc-def introl: normalize ntr-is-tr)

8.4 Properties of normalized path

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

lemma ntrs-c-cases-s[cases set]: [ (s,c),w,(s',c'))∈trcl (ntr fg);
!!esp. [ c'=esp+c; !!s. s ∈# esp ⇒ ∃ p u v. s=[entry fg p] ∧
(u,Spawn p,v)∈edges fg ∧
initialproc fg p ] ⇒ P

] ⇒ P

by (auto dest!: ntrs-is-trss-s elim!: trss-c-cases)

lemma ntrs-c-cases[cases set]: [ ((s,c),w,(s',c'))∈trcl (ntr fg);
!!esp. [ c'=esp+c; !!s. s ∈# esp ⇒ ∃ p u v. s=[entry fg p] ∧
(u,Spawn p,v)∈edges fg ∧
initialproc fg p ] ⇒ P

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

8.4.2 Monitors

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

definition mon-ww fg ww == foldl (∪) {} (map (mon-w fg) ww)
definition mon-loc fg ww == mon-ww fg (map le-rem-s (loc ww))
definition mon-env fg ww == mon-ww fg (map le-rem-s (env ww))

lemma mon-ww-empty[simp]: mon-ww fg [] = {} 
by (unfold mon-ww-def, auto)
lemma mon-ww-uncons[simp]: mon-ww fg (ee#ww) = mon-w fg ee ∪ mon-ww fg ww 
by (unfold mon-ww-def, auto simp add: foldl-un-empty-eq[of mon-w fg ee])
lemma mon-ww-unconc:
mon-ww fg (ww1@ww2) = mon-ww fg ww1 ∪ 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)
lemma mon-env-uncons\[simp\]:
\[
\text{mon-env} \ fg \ (e \# w) = \text{(case } e \text{ of LOC } a \Rightarrow \{\} \mid \text{ENV } a \Rightarrow \text{mon-w} \ fg \ a) \cup \text{mon-env} \ fg \ w
\]
by (unfold mon-env-def) (auto split: el-step.split)

lemma mon-env-unconc:
\[
\text{mon-env} \ fg \ (w1 @ w2) = \text{mon-env} \ fg \ w1 \cup \text{mon-env} \ fg \ w2
\]
by (unfold mon-env-def) (auto simp add: mon-ww-unconc)

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

lemma mon-loc-single\[simp\]:
\[
\text{mon-loc} \ fg \ [e] = \text{(case } e \text{ of ENV } a \Rightarrow \{\} \mid \text{LOC } a \Rightarrow \text{mon-w} \ fg \ a)
\]
by (unfold mon-loc-def) (auto split: el-step.split)

lemma mon-loc-uncons\[simp\]:
\[
\text{mon-loc} \ fg \ (e \# w) = \text{(case } e \text{ of ENV } a \Rightarrow \{\} \mid \text{LOC } a \Rightarrow \text{mon-w} \ fg \ a) \cup \text{mon-loc} \ fg \ w
\]
by (unfold mon-loc-def) (auto split: el-step.split)

lemma mon-loc-unconc:
\[
\text{mon-loc} \ fg \ (w1 @ w2) = \text{mon-loc} \ fg \ w1 \cup \text{mon-loc} \ fg \ w2
\]
by (unfold mon-loc-def) (auto simp add: mon-ww-unconc)

lemma mon-ww-of-foldl\[simp\]:
\[
\text{mon-w} \ fg \ (\text{foldl} \ (@) \ [] \ ww) = \text{mon-ww} \ fg \ ww
\]
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:
\[
w \leq w' \implies \text{mon-ww} \ fg \ w \subseteq \text{mon-ww} \ fg \ w'
\]
by (induct rule: less-eq-list-induct) auto

lemma mon-ww-of-le-rem:
\[
\text{mon-ww} \ fg \ (\text{map} \ \text{le-rem-s} \ w) = \text{mon-loc} \ fg \ w \cup \text{mon-env} \ fg \ w
\]
by (induct w) (auto split: el-step.split)
lemma mon-env: \( w \preceq w' \implies \text{mon-env} fg w \subseteq \text{mon-env} fg w' \)
by (induct rule: less-eq-list-induct) auto

lemma mon-loc: \( w \preceq w' \implies \text{mon-loc} fg w \subseteq \text{mon-loc} fg w' \)
by (induct rule: less-eq-list-induct) 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: \( ((s,c),(s',c')) \in ntrs fg \)
\( \Rightarrow \) mon-s fg s \( \subseteq \) mon-s fg s' \( \land \) 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) ntrp-mon-increasing-s:
\( ((s,c),(s',c')) \in ntrp fg \implies \text{mon-c} fg c \subseteq \text{mon-c} fg c' \)
by (erule gtrE) (auto dest: ntrs-mon-increasing-s simp add: mon-c-unconc)

lemma (in flowgraph) ntrp-mon-increasing-
\( ((s,c),(s',c')) \in \text{trcl} (ntrp fg) \)
\( \Rightarrow \) mon-s fg s \( \subseteq \) mon-s fg s' \( \land \) mon-c fg c \( \subseteq \) mon-c fg c'
by (induct rule: trcl-rev-pair-induct) (auto dest!: ntrp-mon-increasing-s)
8.4.3 Modifying the context

lemmas (in flowgraph) \( \text{ntrs-c-no-mon-s} = \text{trss-c-no-mon}[OF \text{ntrs-is-trss-s}] \)
lemmas (in flowgraph) \( \text{ntrs-c-no-mon} = \text{trss-c-no-mon}[OF \text{ntrs-is-trss}] \)

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

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

lemma (in flowgraph) \( \text{ntrp-mon-env-e-no-ctx}: \)
\begin{align*}
((s,c),\text{ENV } e,(s',c')) \in \text{entrp fg} & \Rightarrow \text{ mon-w fg } e \setminus \text{ mon-s fg } s = \{\} \\
\text{by } & (\text{auto elim!': grtrp.cases dest!: ntrs-mon-e-no-ctx simp add: mon-c-unconc})
\end{align*}

lemma (in flowgraph) \( \text{ntrp-mon-loc-e-no-ctx}: \)
\begin{align*}
((s,c),\text{LOC } e,(s',c')) \in \text{entrp fg} & \Rightarrow \text{ mon-w fg } e \setminus \text{ mon-c fg } c = \{\} \\
\text{by } & (\text{auto elim!': grtrp.cases dest!: ntrs-mon-e-no-ctx})
\end{align*}

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

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

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

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

lemma (in flowgraph) \( \text{ntrs-modify-context}[rule-format]: \)

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

\[\Rightarrow \forall \text{cn. mon-fw fg w} \cap \text{mon-c fg cn} = \{\}\]

\[\Rightarrow (\exists \text{csp. } c' = \text{csp} + c \land \text{mon-c fg} \text{ csp} = \{\} \land \text{((s, cn), w, (s', csp + cn))} \in \text{trcl (ntrs fg)}]\]

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

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

**next**

**case** (\(\text{cons s c e sh w s' c'}\)) \textbf{note} IHP\(=\text{this show} \ ?\)case

**proof** (intro allI \text{impl})

\(\text{fix cn}\)

**assume** \(\text{MON: mon-fw fg (e} \neq \text{ w)} \cap \text{mon-c fg cn} = \{\}\)

**from** ntrs-modify-context-s\[\text{OF IHP I}1] \text{MON obtain csp where S1: ch = csp} + c \land \text{mon-c fg csp} = \{\} \land \text{((s, cn), e, sh, csp + cn)} \in \text{ntrs fg by auto}\)

**with** \(\text{MON have mon-fw fg w} \cap \text{mon-c fg (csp + cn)} = \{\}\) by (auto simp add: mon-c-uncnc)

**with** IHP\(=\text{rule-format}\) \text{obtain csp where S2: c' = csp} + c \land \text{mon-c fg csp} = \{\}

\(\text{((s, cn), e, (s', csp + cn))} \in \text{trcl (ntrs fg)}\) by blast

**thus** \(\exists \text{csp. } c' = \text{csp} + c \land \text{mon-c fg csp} = \{\} \land \text{((s, cn), e} \neq \text{ w, s', csp} + \text{cn)} \in \text{trcl (ntrs fg)}\) by blast

**qed**

**qed**

**lemma** ntrs-xchange-context-s:

**assumes** \(A: ((s,c),ee,(s',csp+c)) \in \text{ntrs fg}\)

and \(B: \text{mon-c fg csp fn} \subseteq \text{mon-c fg e}\)

**shows** \(\text{((s, cn), ee, (s', csp + cn)} \in \text{ntrs fg}\)

**proof** –

**obtain** \(p r u u' v w\) \text{where S: s} = \text{u} \neq \text{ r} \Rightarrow \text{LCall p} \neq \text{ w s'} = \text{v} \neq \text{ u'} \neq \text{ r} \Rightarrow \text{LCall p, (entry fg p} \neq \text{ u'} \neq \text{ r, c)} \in \text{trss fg}\)

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

**proof** –

**from** ntrs.cases\[\text{OF A, simplified}\] \text{obtain ce ce' p r u u' v w where s} = \text{u} \neq \text{ r} \Rightarrow \text{ce ce ee = LCall p} \neq \text{ w s'} = \text{v} \neq \text{ u'} \neq \text{ r} \Rightarrow \text{ce ce'] ((u} \neq \text{ r, ce), LCall p, entry fg p} \neq \text{ u'} \neq \text{ r, ce)} \in \text{trss fg}\)

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

**hence** \(s = u \neq \text{ r ee = LCall p} \neq \text{ w s'} = \text{v} \neq \text{ u'} \neq \text{ r} \Rightarrow \text{(u} \neq \text{ r, c), LCall p, (entry fg p} \neq \text{ u'} \neq \text{ r, c)} \in \text{trss fg}\)

\(\text{((entry fg p, c), w, ([v], csp + c)} \in \text{trcl (trss fg)}\) by auto

**then show** \(\text{?thesis ..}\)

**qed**

**from** ntrs-step\[\text{simplified, OF trss-xchange-context-s|where csp} = \{\}\, \text{, simplified, OF S(4) B} \text{ trss-xchange-context|OF S(5) B]} \text{ S show} \ ?\text{thesis by simp}\]

**qed**

**lemma** ntrs-replace-context-s:

**assumes** \(A: ((s,c+cr),ee,(s',c'+cr)} \in \text{ntrs fg}\)

and \(B: \text{mon-c fg crn} \subseteq \text{mon-c fg cr}\)

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

**proof** –
from ntrs-cases-s[OF A] obtain csp where G: c'+cr = csp+(c+cr) . hence F: c' = csp+c by (auto simp add: union-assoc[symmetric])

from B have MON: mon-c fg (c+crn) ⊆ mon-c fg (c+cr) by (auto simp add: mon-c-unconc)

from ntrs-xchange-context-s[OF - MON] A G have ((s,c+crn),ee,(s',csp+(c+crn)))∈ntrs fg by auto

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

qed

lemma (in flowgraph) ntrs-xchange-context: !!s c c' cn. [I]

((s,c),ww,(s',c'))∈trcl (ntrs fg);

mon-c fg cn ⊆ mon-c fg c [I] ⟹ ∃csp.

c' = csp+c ∧ ((s,cn),ww,(s',csp+cn))∈trcl (ntrs fg)

proof (induct ww)

case Nil note CASE=this

thus ?case by (auto intro!: exI[of - ∅])

next

case (Cons ee ww) note IHP=this

then obtain sh ch where SPLIT: ((s,c),ee,(sh,ch))∈ntrs fg ((sh,ch),ww,(s',c'))∈trcl (ntrs fg) by (fast dest: trcl-uncons)

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

s ∈ # csph ⟹ ∃p u v. s = [entry fg p] ∧ (u, Spawn p, v) ∈ edges fg ∧ initialproc fg p by blast

with ntrs-xchange-context-s SPLIT(1) IHP(3) have ((s,cn),ee,(sh,csph+cn))∈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) ⊆ mon-c fg ch by (auto simp add: mon-c-unconc)

from IHP(1)(OF SPLIT(2)) this obtain csp where C'FMT: c' = csp+ch and SND: ((sh,csph+cn),ww,(s',csp+(csph+cn)))∈trcl (ntrs fg) by blast

note SND

finally have ((s,cn), ee ≠ # ww, s', (csp + csph) + cn) ∈ trcl (ntrs fg) by (simp add: union-assoc)

moreover from CHFMT(1) C'FMT have c' = (csp+csph)+c by (simp add: union-assoc)

ultimately show ?case by blast

qed

lemma (in flowgraph) ntrs-replace-context:

assumes A: ((s,c+cr),ww,(s',c'+cr))∈trcl (ntrs fg)

and B: mon-c fg crn ⊆ mon-c fg cr

shows ((s,c+crn),ww,(s',c'+crn))∈trcl (ntrs fg)

proof –

from ntrs-cases-s[OF A] obtain csp where G: c'+cr = csp+(c+cr) . hence F: c' = csp+c by (auto simp add: union-assoc[symmetric])
from B have MON: mon-c fg (c+crn) ⊆ mon-c fg (c+cr)
  by (auto simp add: mon-c-unconc)

lemma ntrs-stack-comp-s
  assumes s: "(s,c+crn) ∈ ntrs fg
  and w: "(w,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
  have \[ G[of MON]
  G have \((s,c+crn),w,(s',csp+(c+crn))\) ∈ trcl (ntrs fg) by auto
  with F show ?thesis by (simp add: union-assoc)
qed

lemma (in flowgraph) ntr-add-context:
  \[(c,w,c')∈trcl (ntr fg);\ mon-ww fg w ∩ mon-c fg cn = {}\]
  \[\implies (c+cn,w,c'+cn)∈trcl (ntr fg)\]
  by (induct rule: trcl.induct) (simp, force dest: ntr-add-context-s)

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

lemma (in flowgraph) ntrp-add-context:
  \[((s,c),w,(s',c'))\) ∈ trcl (ntrp fg);
  mon-ww fg (map le-rem-s w) ∩ mon-c fg cn = {}\]
  \[\implies ((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: \((s,c), \text{ww}, (s',c')\) \(\in\) trcl (ntrs fg)
\(\Longrightarrow\) \((s@r,c), \text{ww}, (s'@r,c')\) \(\in\) trcl (ntrs fg)
by (induct rule: trcl-pair-induct) (auto intro!: trcl.cons[OF ntrs-stack-comp-s])

lemma (in flowgraph) ntrp-stack-comp-s:
assumes A: \((s,c), ee,(s',c')\) \(\in\) ntrp fg
and B: mon-s fg r \(\cap\) mon-env fg \(\left[ee\right] = \{\}

shows \((s@r,c), ee,(s'@r,c')\) \(\in\) ntrp fg

using A
proof (cases rule: gtrp.cases)
case gtrp-loc then obtain e where CASE: ee = \(\text{LOC} e ((s,c),ee,(s',c'))\) \(\in\) ntrs fg
by auto
hence \((s@r,c), ee,(s'@r,c')\) \(\in\) ntrp 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: \(s' = s\) \(\Rightarrow\) \(\text{sm} = \{\#sm\}\) \(\cup\) ce ce' \(\Rightarrow\) \(\text{ce' = \{\#sm'\}} \cup\) ce' ee = \(\text{ENV e ((sm, \{\#sm\}} \cup\) ce), sm', ce'}\) \(\in\) ntrs fg by auto
from ntrs-modify-context-s[OF CASE(5), where cn=\{\#s @ r\}\} + ee] ntrs-mon-e-no-ctx[OF CASE(5)] B CASE(4) obtain esp where
\(ADD: \{\#s\} \cup ce = esp + \{\{\#s\}\} \cup ce\) mon-c fg esp = \{\} ((sm, \{\#s @ r\}\} + ce), e, sm', esp + \{\{\#s @ r\}\} + ce') \(\in\) ntrs fg by (auto simp add: mon-c-unconc mon-s-unconc
moreover from ADD(1) have \{\#s\} + ce' = \{\#s\} + (esp + ce) by (simp add: union-ac) hence ce' = esp + 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) ntrp-stack-comp:
\[ ((s,c), \text{ww}, (s',c')) \in \text{trcl (ntrp fg)}; \text{mon-s fg r \cap mon-env fg \ww} = \{\} \]
\(\Longrightarrow\) \((s@r,c), \text{ww}, (s'@r,c')\) \(\in\) trcl (ntrp 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 w p. \[ s' = v\#w\#r; (([u],c), ee, ([v,w],c')) \(\in\) ntrs fg;
(u,Call p, u') \(\in\) edges fg \]
\(\Longrightarrow\) \(P\)
shows \(P\)
using A
proof (cases rule: ntrs.cases)
case ntrs-step then obtain \( u' \) \( v \) \( p \) \( w \) where \( \text{CASE: } ec=\text{LCall } p\#w \ s'=v\#u'\#r \ ((u\#r,c), \text{LCall } p,(\text{entry } fg \ p\#u'\#r, c)) \in \text{trss } fg \) by (simp)

from trss-stack-decomp-s\[\text{where } s=[u], \text{ simplified, OF \ CASE(3) }\] have SDC:\(([[u], c), \text{LCall } p,(\text{entry } fg \ p, u'] , c)) \in \text{trss } fg \) by auto

with \( \text{CASE(1,4) }\) have \(([[u], c),ec,([v, u'], c')) \in \text{ntrs } fg \) by (auto intro: ntrs.ntrs-step)

moreover from SDC have \((u, \text{Call } p, u') \in edges \ fg \) by (auto elim!: trss.cases)

ultimately show \( ?\text{thesis using CASE(2) by (blast intro!: EX) }\)

qed

lemma ntrs-stack-decomp-s:
assumes A: \((u\#s\#@r, c),ec,(s',c')) \in \text{ntrs } fg 

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

\[] \implies P 

shows P

apply \( \text{rule ntrs-stack-top-decomp-s}[OF A] \)

apply \( \text{rule EX} \)

apply \( \text{(auto dest: ntrs-stack-comp-s) }\)

done

lemma ntrs-stack-decomp: \( !u \ s \ r \ c \ P. \ [] \)
\( ((u\#s\#@r, c),ww,(s',c')) \in \text{trecl } (\text{ntrs } fg); \)

\( !v \ rr. \ [s'=v\#rr\#r; \ ((u\#s,c),ww,(v\#rr,c')) \in \text{trecl } (\text{ntrs } fg)] \implies P \)

\[] \implies P 

proof (induct \( \text{ww} \))

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

next

case \( \text{Cons } e \ w \) from \( \text{Cons.-prems show } ?\text{case proof} \) (cases rule: trecl-pair-unconsE)

case \( \text{split } sh \ ch \)

from \( \text{ntrs-stack-decomp-s}[OF split(1)] \) obtain \( vh \ uh \ p \) where \( F; \ sh=vh\#uh\#s@r \)
\((u\#s, c), e, vh\#uh\#s, ch) \in \text{ntrs } fg \ (u, \text{Call } p, uh) \in \text{edges } fg \) by blast

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

from \( \text{trecl.cons}[OF F(2) S(2)] \ S(1) \) \( \text{Cons.-prems(2) show } ?\text{thesis by blast} \)

qed

qed

lemma ntrp-stack-decomp-s:
assumes A: \((u\#s\#@r, c),ec,(s',c')) \in \text{ntrp } fg 

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

shows P

using A

proof (cases rule: gtrp.cases)

case gtrp-loc thus \( ?\text{thesis using EX by (force elim!: ntrs-stack-decomp-s intro!: gtrp.intros) }\)

next

69
We first need to define an abstraction function that maps a macrostep on a pair of entered and passed monitors, as required by the $\otimes_{\alpha}$-operator:

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

**Definition**

\[
\text{an fg e }=\text{ if } e=\text{ then }\{\},\{\} \else (\text{mon-e fg (hd e)}, \text{mon-w fg (tl e)})\]

**Lemma** \text{an-simps[simp]}:

\[
\begin{align*}
\text{an fg []} &= \{\},\{\} \\
\text{an fg e#w} &= (\text{mon-e fg e}, \text{mon-w fg w}) \\
&\text{by (unfold an-def, auto)}
\end{align*}
\]

---

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

**Definition**
\( \alpha_{nl} \, fg \, e \equiv \alpha_{n} \, fg \, (le\text{-}rem\text{-}s \, e) \)

**Lemma** \( \alpha_{nl}\text{-def}^{'}: \alpha_{nl} \, fg \equiv \alpha_{n} \, fg \circ le\text{-}rem\text{-}s \)

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

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

**Lemma** \( \alpha_{nl}\text{-simps}[simp] \):

\( \alpha_{nl} \, fg \, (ENV \, x) = \alpha_{n} \, fg \, x \)

\( \alpha_{nl} \, fg \, (LOC \, x) = \alpha_{n} \, fg \, x \)

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

**Lemma** \( \alpha_{nl}\text{-simps1}[simp] \):

\( (\alpha_{nl} \, fg) \circ ENV = \alpha_{n} \, fg \)

\( (\alpha_{nl} \, fg) \circ LOC = \alpha_{n} \, fg \)

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

**Unfolding** \( \alpha_{nl}\text{-def}^{'}[symmetric] \) ..

**Lemma** \( \alpha_{n}\text{-an}: (\alpha_{n} \, fg) \circ le\text{-}rem\text{-}s = \alpha_{n} \, fg \)

**Unfolding** \( \alpha_{nl}\text{-def}^{'}[symmetric] \) ..

**Lemma** \( \alpha_{n}\text{-fst-snd}[simp] \):

\( \text{fst} (\alpha_{n} \, fg \, w) \cup \text{snd} (\alpha_{n} \, fg \, w) = \text{mon-w} \, fg \, w \)

by (induct \( w \)) auto

**Lemma** \( \text{mon-pl\text{-}of\text{-}an}: \text{mon-pl} (\text{map} (\alpha_{n} \, fg) \, w) = \text{mon-loc} \, fg \, w \cup \text{mon-env} \, fg \, w \)

by (induct \( w \)) (auto split: el-step.split)

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

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

apply (unfold mon-pl-def)

apply (induct wb)

apply simp-all

apply (unfold mon-ww-def)

apply (subst foldl-un-empty-eq)

apply (case-tac a)

apply simp-all

done

**Lemma** \( \text{cil-an-cons-helper}: \text{mon-pl} (\text{map} (\alpha_{nl} \, fg) \, wb) = \text{mon-ww} \, fg \, wb \)

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

**Lemma** \( \text{cil-an-cons1}[w\in\alpha_{n}\otimes\alpha_{n} \, f_{g} \, w_{b}]: \text{fst} (\alpha_{n} \, fg \, e) \cap \text{mon-ww} \, fg \, wb = \{\} \)

\( \Rightarrow e \# w \in e \# \alpha_{n} \otimes \alpha_{n} \, f_{g} \, w_{b} \)

apply (rule cil-cons1)

apply assumption

apply (subst cil-an-cons-helper)

apply assumption

done

**Lemma** \( \text{cil-an-cons2}[w\in\alpha_{n}\otimes\alpha_{n} \, f_{g} \, w_{b}]: \text{fst} (\alpha_{n} \, fg \, e) \cap \text{mon-ww} \, fg \, wa = \{\} \)

\( \Rightarrow e \# w \in \alpha_{n} \otimes \alpha_{n} \, f_{g} \, e \# w \)

apply (rule cil-cons2)
apply assumption
apply (subst cil-α-n-cons-helper)
apply assumption
done

8.5.2 Monitors

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

corollary (in flowgraph) ntrs-called-mon:
  assumes A: ((s,c),e,(s',c'))∈ntrs fg
  shows fst (αn fg e) ⊆ mon-s fg s'
using ntrs-mon-s[OF A] by auto

lemma (in flowgraph) ntr-mon-s:
  ((c,e,c')∈ntr fg ⇒ mon-c fg c' = mon-c fg c ∪ fst (αn fg e)
by (erule gtrE) (auto simp add: mon-c-unconc ntrs-c-no-mon-s ntrs-mon-s)
∃ ca' cb' wa wb.
c' = ca' + cb' ∧
w ∈ (wa ⊕ cf fg wb) ∧
mon-c fg ca ∩ (mon-c fg cb ∪ mon-wf fg wb) = {} ∧
mon-c fg cb ∩ (mon-c fg ca ∪ mon-wf fg wa) = {} ∧
(ca, wa, ca') ∈ trcl (ntr fg) ∧ (cb, wb, cb') ∈ trcl (ntr fg)

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

— If the path is empty, the lemma is trivial

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

**next**

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

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

then obtain ch where SPLIT: (ca + cb, e, ch) ∈ ntr fg (ch, w, c'}) ∈ 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)) ∈ ntr fg .

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

then obtain csp where CEHFMT: ceh = csp + ce mon-c fg csp = {} by (auto elim!: ntr-s-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 ntrrs-mon-c-no-ctx[OF NTRS(3)] have MONED: mon-wf fg e ∩ mon-c fg ce = {} by (auto simp add: mon-c-unconc)

— Needed later: The intermediate configuration is valid

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

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

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

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

**case left note CASE = this**

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

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

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

with IHP(1) SPLIT(2) CHVALID obtain ca' cb' wa wb where IHAPP:
c' = ca' + cb'
w ∈ wa ⊕ cf fg wb
mon-c fg (\{#sh\} + csp + (ca - \{#s\})) ∩ (mon-c fg cb ∪ mon-wf fg wb) = {}
mon-c fg cb ∩ (mon-c fg (\{#sh\} + csp + (ca - \{#s\})) ∪ mon-wf fg wa) = {}
(\{#sh\} + csp + (ca - \{#s\}), wa, ca') ∈ trcl (ntr fg)
(cb, wb, cb') ∈ trcl (ntr fg)

73
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#\}))) ∈ ntr fg by blast
  from gtrI-s[OF this] CASE(1) show ?thesis by (auto simp add: union-assoc)
  qed
moreover — and that we can prepend the first step to the interleaving
  have e#w ∈ e#wa ⊗ αn fg wb
proof —
  from ntrs-called-mon[OF NTRS(3)] have fst (αn fg e) ⊆ mon-s fg sh.
  with IHAPP(3) have fst (αn fg e) ∩ mon-ww fg wb = {} by (auto simp add: mon-c-unconc)
  from cil-an-consI[OF IHAPP(2) this] show ?thesis .
  qed
moreover — and that the monitors of the initial context does not interfere
  have mon-c fg ca ∩ (mon-c fg cb ∪ mon-ww fg wb) = {} mon-c fg cb ∩ (mon-c fg ca ∪ mon-ww fg e#wa)) = {}
proof —
  from ntr-mon-increasing-s[OF FIRSTSTEP] IHAPP(3) show mon-c fg ca ∩ (mon-c fg cb ∪ mon-ww fg wb) = {} by auto
  from MONED CASE have mon-c fg cb ∩ mon-w fg e = {} by (auto simp add: mon-c-unconc)
  with ntr-mon-increasing-s[OF FIRSTSTEP] IHAPP(4) show mon-c fg cb ∩ (mon-c fg ca ∪ mon-ww fg (e#wa)) = {} by auto
  qed
ultimately show ?thesis by blast
next — The other case, that is if the first step was made on a thread in the right part of the configuration, is shown completely analogously

case right note CASE=this
with CEHFMT NTRS have CHFMT: ch=ca+(\{#sh#\} + csp+(cb-\{#s#\})))
by (simp add: union-ac)
with IHP(1) SPLIT(2) CHVALID obtain ca ' cb' wa wb where IHAPP: c'=ca'+cb' w∈wa@αn fg wb mon-c fg ca ∩ (mon-c fg (\{#sh#\} + csp+(cb-\{#s#\}))) ∪ mon-ww fg wb)={} mon-c fg ((\{#sh#\} + csp+(cb-\{#s#\}))) ∩ (mon-c fg ca ∪ mon-ww fg wa)={}
(ca, wa, ca')∈trcl (ntr fg) ((\{#sh#\} + csp+(cb-\{#s#\})), wb, cb')∈trcl (ntr fg) by blast
moreover
have FIRSTSTEP: (cb, e, \{#sh#\} + csp+(cb-\{#s#\}))) ∈ ntr fg proof —
from CASE(2) have mon-c fg (cb−#s#) ⊆ mon-c fg ce by (auto simp add: mon-c-unconc)
with ntrs-exchange-context-s NTRS(3) CEHFMT CASE(2) have ((s,cb−#s#),e,(sh,csp+(cb−#s#))) ∈ ntrs cb
fg by blast
from gtrl-s[OF this] CASE(1) show ?thesis by (auto simp add: union-assoc)
qed

with IHAPP(6) have PA: (cb,e#wb,cb') ∈ trcl (ntr fg) by simp
moreover
have e#w ∈ wa ⊗ αn fg e#wb
proof –
from ntrs-called-mon[OF NTRS(3)] have fst (αn fg e) ⊆ mon-s fg sh .
with IHAPP(4) have fst (αn fg e) ∩ 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 ∩ (mon-c fg ca ∪ mon-ww fg wa) = {} mon-c fg ca ∩ (mon-c fg cb ∪ mon-ww fg (e#wb)) = {}
proof –
from ntr-mon-increasing-s[OF FIRSTSTEP] IHAPP(4) show mon-c fg cb ∩ (mon-c fg ca ∪ mon-ww fg wa) = {} by auto
from MONED CASE have mon-c fg ca ∩ mon-w fg e = {} by (auto simp add: mon-c-unconc)
with ntr-mon-increasing-s[OF FIRSTSTEP] IHAPP(3) show mon-c fg ca ∩ (mon-c fg cb ∪ mon-ww fg (e#wb)) = {} by auto
qed
ultimately show ?thesis by blast
qed
qed

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

lemma (in flowgraph) ntrp-split:
|[[s c₁ c₂ s' c'.|]
|((s,c₁+c₂),w,((s',c')))∈trcl (ntrp fg); valid fg {{#s#}+c₁+c₂}]
|⇒ ∃ w₁ w₂ c₁' c₂'.
|↓ w ∈ w₁ ⊗ αn fg (map ENV w₂) ∧
|c' = c₁'+c₂' ∧
|((s,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 {{#s#}+c₁} = {}
proof (induct w)
case Nil thus ?case by (auto intro: exI[of - []] exI[of - {#}])
next
case (Cons ee w) then obtain sh ch where SPLIT: ((s,c₁+c₂),ee,(sh, ch)) ∈ ntrp fg ((sh, ch),w,((s',c'))) ∈ trcl (ntrp fg) by (fast dest: trcl-ancons)
from SPLIT(1) show ?case proof (cases rule: gtrl.cases)
case gtrp-loc then obtain e where CASE: ee=LOC e ((s,c1+c2),e,(sh,ch))∈ntrs
fg by auto
from ntrs-c-cases-s[OF CASE(2)] obtain csp where CHFMT; ch:=(csp+c1)+c2
∩s. s ∈ csp ⇒ ∃ p u v. s = [entry fg p] ∧ (u, Spawn p, v) ∈ edges fg ∧ initialproc
fg p by (simp add: union-associative, blast)
with c-of-initial-no-mon have CSPNOMON: mon-c fg csp = {} by auto
from ntr-valid-preserves-s[OF gtrl-s, OF CASE(2)] Cons.prems(2) CHFMT
have VALID: valid fg ({}+(csp+c1)+c2) by (simp add: union-associative)
from Cons.hyps[OF - VALID, of s' c'] CHFMT(1) SPLIT(2) obtain w1 w2
c1' c2' where IHAPP: w ∈ w1 ⊗_{anl 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)) = {} by blast
have ee#w ∈ ee#w1 ⊗_{anl 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 w1)) \ mon-s fg sh = {} .
moreover have mon-ww fg w2 ≤ mon-ww fg (map le-rem-s (env w1)) 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 ntr-s-mon-s[OF CASE(2)] CASE(1) show fst (anl fg ee) \ mon-pl (map (anl fg) (map ENV w2)) = {} by (auto simp add: cil-\an-cons-helper)
qed (rule IHAPP(1))
moreover
have ((s,c1),ee#w1,(s',c1'))∈trcl (ntrp fg) proof –
from ntr-x-change-context-s[of s c1+c2 e sh csp fg c1] CASE(2) CHFMT(1)
have ((s, c1), e, sh, csp + c1) ∈ ntrs 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) ntr-s-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 ntr-s-mon-increasing-s[OF CASE(2)] CHFMT(1) IHAPP(6)
have mon-ww fg w2 \ mon-c fg ({}+(csp)) + 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

76
\(fg\) by auto

from ntrs-cases-s \((\text{OF CASE}[5])\) obtain \(csp\) where \(HFMT\): add-mset \(s\) ceh

\[
\text{csp} + (\text{add-mset} \ s \ ce) \land \ s \in\ csp \Rightarrow \exists u, v. \ s = [\text{entry} \ fg \ p] \land (u, \text{Spum} p, v) \in \text{edges} \ fg \ \& \ \text{initialproc} \ fg \ p \ \text{by (blast)}
\]

from union-left-cancel \([\{#s\}\] ceh csp+ce \) \(HFMT(1)\) have CEHFMT:

\[
\text{ceh} = \text{csp} + \text{ce} \ \text{by (auto simp add: union-ac)}
\]

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

from CASE\([2]\)\([\text{symmetric}]\) show ?thesis proof (cases rule: mset-unplusm-dist-cases)

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

case left

with \(HFMT(1)\) CASE\([4]\) CEHFMT have CHFMT' := (csp+\{#ssh\}+(c1-\{#ss\})))

\(c2\) by (simp add: union-ac)

have VALID: valid \(fg\) \(\{\{#s\}\} + (csp+\{#ssh\}+(c1-\{#ss\}))\) + \(c2\)

proof

from ntr-valid-preserve-s \([\text{OF gtrl-s, OF CASE}[5]]\) Cons.prems\([2]\) CASE\([2]\)

have valid \(fg\) \(\{\{#s\}\} + (\{#s\}) + \text{ceh}\) by (simp add: union-assoc add-msf-commute)

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

qed

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

\((s, \ csp + \{#ssh\} + (c1 - \{#ss\}))\), \(w1, s', c1') \in \text{trcl (ntrpg \(fg\)) \(c2, w2, c2'\)) \in \text{trcl (ntrfg)}\)

mon-ww \(fg\) (map le-rem-s \(w1\)) \(\cap\) mon-c \(fg\) \(c2\) = \(\{\}\) mon-ww \(fg\) \(w2\) \(\cap\) mon-c

\(fg\) \(\{\{#s\}\} + (csp + \{#ssh\} + (c1 - \{#ss\}))\) = \(\{\}\) blast

have ee : \(\text{w} \in (\text{ee}\#\#\#\#) \odot_{\alpha} fg\) map ENV \(w2\) proof (rule cil-consI)

from IHAPP\([6]\) have mon-ww \(fg\) \(w2\) \(\cap\) mon-s \(fg\) \(ssh\) = \(\{\}\) by (auto simp add: cil-mon-cons-helper)

moreover from ntrs-mon-s \([\text{OF CASE}[5]]\) CASE\([1]\) have \(\text{fst (axn} fg \ ee) \subseteq\)

mon-s \(fg\) \(ssh\) by auto

ultimately have \(\text{fst (axn} fg \ ee) \cap\) mon-ww \(fg\) \(w2\) = \(\{\}\) by auto

moreover have mon-pl (map (axn \(fg\) map ENV \(w2\))) = mon-ww \(fg\) \(w2\)

by (simp add: cil-on-cons-helper)

ultimately show \(\text{fst (axn} fg \ ee) \cap\) mon-pl (map (axn \(fg\) map ENV \(w2\)))

= \(\{\}\) by auto

qed (rule IHAPP\([1]\))

moreover

have SS: \((\text{s,c1),ee, (s,csp + \{#ssh\} + (c1 - \{#ss\}))}) \in\text{ntrp} \ fg\ proof -

from left HFMT\([1]\) have \(\{\#s\} + ce = \{\#ss\} + (c1 - \{#ss\})\) + \(c2\ \{\#s\} + ceh\)

= \(csp + (\{#\}) + (c1 - \{#ss\}) + c2\) by (simp-all add: union-ac)

with CASE\([5]\) ntrs-exchange-context-s \([\text{of ss} \ \{\#s\} + (c1 - \{#ss\}) + ceh\)

csp \(fg\) \((\{#s\} + (c1 - \{#ss\}))\) have

\((\text{ss, add-mset} \ s (c1 - \{#ss\}))\), \(e, ssh, add-mset \ s (csp + (c1 - \{#ss\}))\)

\(\in\ ntrs \ fg\) by (auto simp add: mon-c-unconc union-ac)

from gtrp.gtrp-env\([\text{OF this}]\) left\([1]\)\([\text{symmetric}]\) CASE\([1]\) show ?thesis by

(simp add: union-ac)

qed

from trcl.cons \([\text{OF this IHAPP\([3]\)}\]) have \(\{(s, c1), ee \# w1, s', c1'\} \in\text{trcl (ntrp} \)}
fg) ·

moreover 
from ntrs-mon-e-no-ctx[OF CASE(5)] left CASE(1) IHAPP(5) have mon-ww fg (map le-rem-s (ee#w1)) ∩ mon-c fg c2 = {} by (auto simp add: mon-c-unconc)
moreover 
from ntrp-mon-increasing-s[OF SS] IHAPP(6) have mon-ww fg w2 ∩ mon-c fg ({} + c1) = {} by (auto simp add: mon-c-unconc)
moreover note IHAPP(2,4)
ultimately show ?thesis by blast

next
— Made an env-step in c2. This is considered the right part. Induction hypothesis is applied with original local thread and the spawned threads on the right side

  case right
  with HFMT(1) CASE(4) CHFMT have CHFMT’: ch=c1 + (csp+{ssh#}+(c2-{ssh#}))
  by (simp add: union-ac)
  have VALID: valid fg ({} + c1 + (csp+{ssh#}+(c2-{ssh#})))
  proof –
  from ntr-valid-preserve-s[OF ctrl-s, OF CASE(5)] Cons.prems(2) CASE(2)
  have valid fg ({} + ({} + ceh)) by (auto simp add: union-ac add-mset-commute)
  with right CEHFMT show ?thesis by (auto simp add: union-ac add-mset-commute) –
  qed

  from Cons.hyps[OF VALID, OF s1 s’ c1’ CHFMT’ SPLIT(2) CASE(3) obtain
  w1 w2 c1’ c2’ where IHAPP: w ∈ w1 ⊗ctx fg map ENV w2 c’ = c1’ + c2’
  (s, c1), w1, s’, c1’) ∈ trcl (ntrp fg) (csp + {ssh#} + (c2 – {ssh#}))
  = {} mon-ww fg w2 ∩ mon-c fg ({} + c1) = {} by blast
  have ee w ∈ w1 ⊗ctx fg map ENV c#w2 proof (simp add: CASE(1), rule cil-cons2)
  from IHAPP(5) have mon-ww fg (map le-rem-s w1) ∩ mon-s fg ssh = {}
  by (auto simp add: mon-c-unconc)
  moreover from ntrs-mon-s[OF CASE(5)] CASE(1) have fst (xml fg ee) ⊆
  mon-s fg ssh by auto
  ultimately have fst (xml fg ee) ⊆ mon-ww fg (map le-rem-s w1) = {}
  by auto
  moreover have mon-pl (map (xml fg) w1) = mon-ww fg (map le-rem-s w1)
  by (unfold xml-def’) (simp add: cil-an-cons-helper[symmetric])
  ultimately show fst (xml fg (ENV e)) ⊆ mon-pl (map (xml fg) w1) = {}
  using CASE(1) by auto
  qed (rule IHAPP(1))

moreover
have SS: (c2,e,csp+{ssh#}+(c2-{ssh#})) ∈ ntr fg proof –
from right HFMT(1) have {#s#}+ce={#s#}+c1+(c2-{#s#}) {#s#} + ceh
= csp+{#s#}+c1+(c2-{#s#}) by (simp-all add: union-ac)
with CASE(5) ntrs-exchange-context-s[OF SS {#s#}+c1+(c2-{#s#}) e ssh
csp fg c2-{#s#}]
have
((ss, c2 – {#s#}), e, ssh, csp+ (c2 – {#s#})) ∈ ntrs fg by (auto simp add: mon-c-unconc union-ac)
from gtrI-s[OF this] right(1)[symmetric] show ?thesis by (simp add: union-ac)
qed
from trcl.cons[OF this IHAPP(4)] have (c2, e # w2, c2') \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 ntrs-mon-e-no-ctx[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

— Just a check that flowgraph.ntrp-split is really a generalization of flowgraph.ntr-split:

lemma (in flowgraph) ntrp-split':
assumes A: \( (ca+cb,w,c') \in \text{trcl} (ntr fg) \)
and VALID: valid fg (ca+cb)
shows \exists ca' cb' wa wb.\[\begin{array}{c}c' = ca' + cb' \\
w \in (wa \otimes_{\alpha n} fg) wb \land \\
\text{mon-c fg ca} \cap (\text{mon-c fg cb} \cup \text{mon-ww fg wb}) = {} \land \\
\text{mon-c fg cb} \cap (\text{mon-c fg ca} \cup \text{mon-ww fg wa}) = {} \land \\
(ca,wa,ca') \in \text{trcl} (ntr fg) \land \\
(cb,wb,cb') \in \text{trcl} (ntr fg)\end{array}\]using A VALID by (rule ntrp-split)

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

lemma (in flowgraph) ntr-unsplit:
assumes A: \( w \in wa \otimes_{\alpha n} fg wb \) and 
B: \( (ca,wa,ca') \in \text{trcl} (ntr fg) \)
\( (cb,wb,cb') \in \text{trcl} (ntr fg) \)
\( \text{mon-c fg ca} \cap (\text{mon-c fg cb} \cup \text{mon-ww fg wb}) = {} \) \land \\
\( \text{mon-c fg cb} \cap (\text{mon-c fg ca} \cup \text{mon-ww fg wa}) = {} \) \land \\
\( (ca,wa,ca') \in \text{trcl} (ntr fg) \land \\
(cb,wb,cb') \in \text{trcl} (ntr fg) \)
shows \((ca+cb,w,ca'+cb') \in \text{trcl} (ntr fg)\)

proof —
— We have to generalize and rewrite the goal, in order to apply Isabelle’s induction method
from A have \( \forall ca \ cb. \ (ca,wa,ca') \in \text{trcl} (ntr fg) \land (cb,wb,cb') \in \text{trcl} (ntr fg) \land \text{mon-c fg ca} \cap (\text{mon-c fg cb} \cup \text{mon-ww fg wb}) = {} \land \text{mon-c fg cb} \cap (\text{mon-c fg ca} \cup \text{mon-ww fg wa}) = {} \) \land \\
\( (ca+cb,w,ca'+cb') \in \text{trcl} (ntr fg) \)
— We prove the generalized goal by induction over the structure of consistent interleaving
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 proof (intro allI implI, goal-cases)}

\textbf{case (1 ca cb)}

\textbf{hence I: w’ \in w1’ \otimes_{\alpha n fg} w2 fst (\alpha n fg e) \cap mon-pl (map (\alpha n fg) w2) = \{}\}

\textbf{!!ca cb.}

\[ [(ca, w1’, ca’) \in \text{trcl (ntr fg)};\]

\[ (cb, w2, cb’) \in \text{trcl (ntr fg)};\]

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

\[ \text{mon-c fg cb} \cap (\text{mon-c fg ca} \cup \text{mon-ww fg w1’}) = \{} \quad \Rightarrow \]

\[ (ca + cb, w’, ca’ + cb’) \in \text{trcl (ntr fg)}\]

\[ (ca, e \# w1’, ca’) \in \text{trcl (ntr fg)} (cb, w2, cb’) \in \text{trcl (ntr fg)} \]

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

\[ \text{mon-c fg cb} \cap (\text{mon-c fg ca} \cup \text{mon-ww fg (e \# w1’)}) = \{} \quad \text{by blast+}\]

— Split the left path after the first step

\textbf{then obtain cah where SPLIT: (ca,e,cah)\in ntr fg (cah,w1’,ca’)\in trcl (ntr fg)}

\textbf{by (fast dest; trcl-uncons)}

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

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

\textbf{also}

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

\textbf{have (cah + cb, w’, ca’ + cb’) \in \text{trcl (ntr fg)} proof –}

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

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

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

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

\textbf{case (1 ca cb)}

\textbf{hence I: w’ \in w1 \otimes_{\alpha n fg} w2’ fst (\alpha n fg e) \cap mon-pl (map (\alpha n fg) w1) = \{}\]

\textbf{!!ca cb.}

\[ [(ca, w1, ca’) \in \text{trcl (ntr fg)};\]

\[ (cb, w2’, cb’) \in \text{trcl (ntr fg)};\]

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

\[ \text{mon-c fg cb} \cap (\text{mon-c fg ca} \cup \text{mon-ww fg w1}) = \{} \quad \Rightarrow \]

\[ (ca + cb, w’, ca’ + cb’) \in \text{trcl (ntr fg)}\]
lemma (in flowgraph) ntrp-unsplit:
assumes A: \( w \in \text{wa} \otimes \text{and} \ f g (\text{map ENV} \ w b) \) and
B: ((s',ca),wa,(s,ca'))\( \in \text{trcl (ntrp f g)} \)
\( (cb,wb,cb') \in \text{trcl (ntrp f g)} \)
\( \text{mon-c f g } (\{\#s\#\} + ca) \cap (\text{mon-c f g } cb \cup \text{mon-ww f g } wb) = \{\} \)
\( \text{mon-c f g } cb \cap (\text{mon-c f g } ca \cup \text{mon-ww f g } w1) = \{\} \) by blast+

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

next
"case" left e \( w' \) \( w1' \) \( w2' \)
thus \(?case\)
proof (intro allI impI, goal-cases)
case (1 s ca cb wb)
\text{hence} I: \( w' \in w1' \otimes \text{and} f g w2 \) \( \text{fst (} \text{and} f g e) \cap \text{mon-pl (} \text{map (} \text{and} f g) \) w2) = \{\} \)
\( \text{!!s ca cb wb}, \]
\( w2 = \text{map ENV} \ w b; \)
\( ((s, ca), w1', s', ca') \in \text{trcl (ntrp f g)}; \)
(cb, wb, cb') ∈ trcl (ntrp fg);
  mon-c fg (\{#s#\} + ca) ∩ (mon-c fg cb ∪ mon-ww fg wb) = \{
  mon-c fg cb ∩ (mon-c fg (\{#s#\} + ca) ∪ mon-ww fg (map le-rem-s w1'))

= \{
\]
\implies ((s, ca + cb), w', s', ca' + cb') ∈ trcl (ntrp fg)
   w2 = map ENV wb
((s, ca), e # w1', s', ca') ∈ trcl (ntrp fg)
   (cb, wb, cb') ∈ trcl (ntrp fg)
  mon-c fg (\{#s#\} + ca) ∩ (mon-c fg cb ∪ mon-ww fg wb) = \{
  mon-c fg cb ∩ (mon-c fg (\{#s#\} + ca) ∪ mon-ww fg (map le-rem-s (e # w1')))

= \{
\]
  by blast+
  then obtain sh ca h where SPLIT: ((s,ca),e,(sh,cah))\in ntrp fg ((sh,cah),w1',(s',ca'))\in trcl (ntrp fg) by (fast dest: trcl-uncons)
  from ntrp-add-context-s[of SPLIT(1), of cb] I(8) have ((s, ca + cb), e, sh, cah + cb) ∈ ntrp fg by auto
  also have ((sh,cah+cb),w',(s',ca'+cb'))\in trcl (ntrp fg) proof (rule I(3))
  from ntrp-mon-s[of SPLIT(1)] I(2,4,7,8) show I: mon-c fg (\{#sh#\} + cah) ∩ (mon-c fg cb ∪ mon-ww fg wb) = \{
  by (cases e) (rename-tac a, case-tac a, simp add: cilm-on-cons-helper, fastforce simp add: cilm-on-cons-helper)+
  from I(8) I show mon-c fg cb ∩ (mon-c fg (\{#sh#\} + cah) ∪ mon-ww fg (map le-rem-s w1')) = \{
  by auto
  qed (auto simp add: I(4,6) SPLIT(2))
  finally show ?case .
  qed

next
  case (right ee w' w2' w1)
  thus ?case
  proof (intro allI impI, goal-cases)
    case (1 s ca cb wb)
    hence I: w' ∈ w1 ∩ mon fg w2' fst (anl fg ee) ∩ mon-pl (map (anl fg) w1)

= \{
\]
  !!s ca cb wb. \[ w2' = map ENV wb;
  ((s, ca), w1, s', ca') ∈ trcl (ntrp fg);
  (cb, wb, cb') ∈ trcl (ntrp fg);
  mon-c fg (\{#s#\} + ca) ∩ (mon-c fg cb ∪ mon-ww fg wb) = \{
  mon-c fg cb ∩ (mon-c fg (\{#s#\} + ca) ∪ mon-ww fg (map le-rem-s w1'))

= \{
\]
\implies ((s, ca + cb), w', s', ca' + cb') ∈ trcl (ntrp fg)
   ee#w2' = map ENV wb
((s, ca), w1, s', ca') ∈ trcl (ntrp fg)
   (cb, wb, cb') ∈ trcl (ntrp fg)
  mon-c fg (\{#s#\} + ca) ∩ (mon-c fg cb ∪ mon-ww fg wb) = \{
  mon-c fg cb ∩ (mon-c fg (\{#s#\} + ca) ∪ mon-ww fg (map le-rem-s w1'))

= \{
\]
  by fastforce+
  from I(4) obtain ee wb' where EE: wb=e#wb' ee=ENV e w2'=map ENV

82
wb' by (cases wb, auto)

with I(6) obtain cbh where SPLIT: (cb,e,cbh)∈ntr fg (cbh,wb',cb')∈trcl (ntr fg)
by (fast dest: trcl-uncons)

have ((s, ca + cb, ee, (s, ca + cbh)) ∈ ntrp fg)
proof —
  from gtrp[OF SPLIT(I)] obtain sb ceb sbh cebh where NTRS: cb = add-mset sb ceb ccbh = add-mset sbh cebh ((sb, ceb), e, sbh, cebh) ∈ ntrp fg .
  from ntrs-add-context-s[OF NTRS(3), of {(#s#) + ca}] EE(1) I(7) have
  ((sb, add-mset s (ca+ceb)), e, sbh, add-mset s (ca+ceb)) ∈ ntrp fg by (auto simp add: union-ac)

  from gtrp-env[OF this] NTRS(1,2) EE(2) show ?thesis by (simp add: union-ac)
  qed

  also have 
  proof (rule I(3))
    from ntr-mon-s[OF SPLIT(1)] I(2,4,7,8) EE(2) show I: mon-c fg cbh \n    \{ mon-c fg (\{#s#\} + ca) \cup \{ mon-c fg cbh \cup mon-ww fg wb\} = \}
    by (cases e) (simp add: cil-\-cons-helper, fastforce simp add: cil-\-cons-helper)

    from I(7) 1 EE(1) show mon-c fg (\{#s#\} + ca) \cup \{ mon-c fg cbh \cup 
    mon-ww fg wb\} = \} by auto
    qed (auto simp add: EE(3) I(5) SPLIT(2))
  finally show ?case .
  qed

  qed

  }
with A B show ?thesis by blast
qed

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

theorem (in flowgraph) ntr-\-interleave: valid fg (ca+cb) \implies 
  \{(ca+cb,w,e')\in trcl (ntr fg) \iff 
  \exists ca' cb' wa wb.
  c'=ca'+cb' \land 
  w\in (wa\otimes_{an} fg\,wb) \land 
  mon-c fg ca \cup \{ mon-c fg cb \cup mon-ww fg wb\} = \}
  \land
  mon-c fg cb \cup \{ mon-c fg ca \cup mon-ww fg wa\} = \} \land
  \land
  (ca,wa,ca')\in trcl (ntr fg) \land \(cb,wb,cb')\in trcl (ntr fg) \land
by (blast intro: ntr-split ntr-\-unsplit)

— Here is the corresponding version for executions with an explicit local thread

theorem (in flowgraph) ntr-\-interleave:
  valid fg (\{#s#\}+c1+c2) \implies 
  \{(s,c1+c2),w,(s',e')\in trcl (ntrp fg) \iff 
  \exists w1 w2 c1' c2'.
  w \in w1 \otimes_{an} fg (map ENV w2) \land 
  c'=c1'+c2' \land 
  ((s,c1),w1,(s',c1'))\in trcl (ntrp fg) \land
(c_2, w_2, c_2') \in \text{trcl} (\text{ntr } f g) \land \\
\text{mon-ww } f g (\text{map } \text{le-rem-s } w_1) \cap \\
\text{mon-c } f g c_2 = \{} \land \\
\text{mon-ww } f g w_2 \cap \text{mon-c } f g ((\#s\#)+cl) = \{} \\
\text{apply (intro iffI)} \\
\text{apply (blast intro: ntrp-split)} \\
\text{apply (auto intro!: ntrp-unsplit simp add: valid-unconc mon-c-unconc)} \\
\text{done}

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

\begin{corollary}[in flowgraph] ntr2ntrp: \\
(c_2, w_2, c_2') \in \text{trcl} (\text{ntr } f g); \\
\text{mon-c } f g (\text{add-mset } s \text{ cl}) \cap (\text{mon-c } f g c \cup \text{mon-ww } f g w) = \{} \\
\text{using ntrp-unsplit}[\text{where } \text{wa=[]} \text{simplified}] \text{ by fast}
\end{corollary}

\section*{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.

\begin{lemma}[in flowgraph] ntr-reverse-split: \[
(c, w, s') c' \in \text{trcl} (\text{ntr } f g); \\
\text{valid } f g c \implies \exists s ce w_1 w_2 ce_1' ce_2'. \\
\text{c=}{\#s\#}+ce \land \\
\text{ce'=ce1'+ce2'} \land \\
w \in w_1 \otimes \alpha \text{fg} w_2 \land \\
\text{mon-s } f g s \cap (\text{mon-c } f g ce \cup \text{mon-ww } f g w_2) = \{} \land \\
\text{mon-c } f g ce \cap (\text{mon-s } f g s \cup \text{mon-ww } f g w_1) = \{} \land \\
(\{\#s\#\}, w_1, (\#s'\#)+ce_1') \in \text{trcl} (\text{ntr } f g) \land \\
(ce, w_2, ce_2') \in \text{trcl} (\text{ntr } f g)
\]
\end{lemma}

— 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

\textbf{proof (induct c rule: multiset-induct')}
— If the initial configuration is empty, we immediately get a contradiction

\textbf{case empty hence False by auto thus ?case ..}

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

\textbf{case (add ce s)}
— We split the path by this initial configuration

\textbf{from ntr-split[OF add.prems(1,2)] obtain ce1' ce2' w_1 w_2 where}

\text{SPLIT: add-mset } s' ce'=ce1'+ce2' \text{ w}\in w_1 \otimes \alpha \text{fg} w_2
mon-c fg ce ∩ (mon-s fg s ∩ mon-ww fg w1) = {}
mon-s fg s ∩ (mon-c fg ce ∪ mon-ww fg w2) = {}
(\{#s#\}, w1, ce) ∈ trcl (ntr fg)
(cet, w2, ce) ∈ trcl (ntr fg)

by auto
— And then check whether splitting off s was the right choice

from SPLIT(1) show \( \text{case proof (cases rule: mset-unplusm-dist-cases)} \)
— Our choice was correct, \( s' \) is generated by some descendant of \( s'' \)

\( \text{case left} \)

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

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

\( \text{case right with SPLIT(6) have C: } \{ ce, w2, \{#s\} + (ce' - \{#s\}) \} \in trcl (ntr fg) \)
by auto
— In this case we apply the induction hypothesis to the path from \( ce \)

from add.prems(2) have VALID: valid fg ce mon-s fg s ∩ mon-c fg ce = {} by
(auto simp_all add: valid-unconc)

from add.hyps[OF C VALID(1)] obtain st cet w21 w22 ce21 ce22 where
IHAPP:
cet = \{#st\} + cet
ce2' - \{#s\} = ce21' + ce22'
w2 ∈ w21 ∩ cet
mon-s fg st ∩ (mon-c fg cet ∪ mon-ww fg w22) = {}
mon-c fg cet ∩ (mon-s fg st ∩ mon-ww fg w21) = {}
(\{#s\}, w21, \{#s\} + ce21') ∈ trcl (ntr fg)
(cet, w22, ce22') ∈ trcl (ntr fg) by blast

— And finally we add the path from \( s \) again. This requires some monitor sorting
and the associativity of the consistent interleaving operator.

from cil-assoc2 [of w w1 - w2 w22 w21] SPLIT(2) IHAPP(3) obtain w1 where
CASSOC: w ∈ w22 ∩ cet
IHAPP(7) by (auto simp add: cil-commute)

from CASSOC IHAPP(1,3,4,5) SPLIT(3,4) have COMBINE: (add-mset s cet, w1, ce1' + ce22') ∈ trcl (ntr fg) using ntr-unsplit[OF CASSOC(2) SPLIT(5)
IHAPP(7)] by (auto simp add: mon-c-unconc mon-ww-cil Int-Un-distrib2)

moreover from CASSOC IHAPP(1,3,4,5) SPLIT(3,4) have mon-s fg st ∩
(mon-c fg \{#s\} + cet) ∪ mon-ww fg w1 = {}
mon-c fg \{#s\} + cet) ∩ (mon-s
fg st ∪ mon-ww fg w1) = {} by (auto simp add: mon-c-unconc mon-ww-cil)

moreover from right IHAPP(1,2) have \{#s\} + cet = \{#s\} + (\{#s\} + cet)
ce' = ce21' + (ce1' + ce22') by (simp_all add: union-ac)

moreover note IHAPP(6) CASSOC(1)

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

qed


ced

end

9 Constraint Systems

theory ConstraintSystems
imports Main AcquisitionHistory Normalization begin

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

Constraint systems are widely used in static program analysis. There least solution describes the desired analysis information. In its generic form, a constraint system \( R \) is a set of inequations over a complete lattice \((L, \sqsubseteq)\) and a set of variables \( V \). An inequation has the form \( R[v] \sqsubseteq \text{rhs} \), 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:

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

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 \((P(\mathbb{P}a), \subseteq)\) for some type ‘\( a \). This admits a convenient formalization in Isabelle/HOL using inductively defined sets. We inductively define a relation between variables\(^3\) and the elements of their values in the least solution, i.e. the set \( \{(v, x) \mid x \in R[v]\} \). For example, the constraint generator pattern from above would become the following introduction rule in the inductive definition of the set \( S\text{-cs } fg k \):

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

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

\(^3\)Variables are identified by control nodes here
9.1 Same-level paths

9.1.1 Definition

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

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

\[
\text{inductive-set}
\]

\[
S\text{-}cs : (\wedge n, \wedge p, \wedge ba, \wedge m, \wedge more) \text{ flowgraph-rec-scheme} \Rightarrow \text{nat} \Rightarrow (\wedge n \times \wedge m \times (\wedge p \text{ multiset}) \text{ set})
\]

for \( fg \ k \)

where

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

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

\[-\text{S-call:} : [(u, \text{Call} q, v) \in \text{edges} fg; (u, M, P) \in S\text{-}cs\ fg\ k;\]

\[-\text{return} fg q, Ms, Ps) \in S\text{-}cs\ fg\ k;\ P' \subseteq \#P + Ps;\ size\ P' \leq k]
\]

\[-\implies (v, \text{mon} fg q \cup M \cup Ms, P') \in S\text{-}cs\ fg\ k
\]

\[-\text{S-spawn:} : [(u, \text{Spawn} q, v) \in \text{edges} fg; (u, M, P) \in S\text{-}cs\ fg\ k;\]

\[-P' \subseteq \#\{q\#\} + P;\ size\ P' \leq k]\]

\[-\implies (v, M, P') \in S\text{-}cs\ fg\ k
\]

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: $! p v c' P$.

\[ (((\text{entry } fg p),\{\#\}),w,([v],[c'])) \in \text{trcl} (\text{trss } fg); \]
\[
    \text{size } P \leq k; (\lambda p. \text{[entry } fg p]) \# P \subseteq \# c'
\]
\[
    \quad \Rightarrow (v,\text{mon-w } fg w.P) \in S\text{-cs } fg k
\]

**proof** (induct $w$ rule: length-compl-rev-induct)

**case Nil** thus $?case$ by (auto intro: S-init)

**next**

**case** (snoe $w$ $c$) then obtain $sh ch$ where $SPLIT$: 

\[ (((\text{entry } fg p),\{\#\}),w,([sh, ch]) \in \text{trcl} (\text{trss } fg); \]

\[
    (\text{size } P \leq k; (\lambda p. \text{[entry } fg p]) \# P \subseteq \# c')
\]

\[
    \Rightarrow (v,\text{mon-w } fg w.P) \in S\text{-cs } fg k
\]

**from** SPLIT(2) **show** $?case$ proof (cases rule: trss.cases)

**case** trss-base then obtain $u a v$ where CASE: $e=\text{LBase}$ $a \; \text{sh}=\lfloor u \rfloor \; \text{ch}=c' \; (u,\text{Base} a, v) \in \text{edges } fg$ by auto

**with** snoe.hyps[of $w \; p \; u \; c'$, $OF$ - - snoe.prems(2,3)] **SPLIT(1) have** $(u,\text{mon-w } fg w.P) \in S\text{-cs } fg k$ by blast

**moreover from** CASE(1) **have** $\text{mon-e } fg \; e = \{ \}$ by simp

**ultimately show** $?thesis$ using $S$-base[OF CASE(4)] by (auto simp add: mon-e-uncanc)

**next**

**case** trss-ret then obtain $q$ where CASE: $e=\text{LRet}$ $sh=\text{return } fg \; q\#\lfloor v \rfloor \; \text{ch}=c'$ by auto

**with** SPLIT(1) **have** $(\lfloor \text{entry } fg p \rfloor,\{\#\}), w, \text{[return } fg q,v], c') \in \text{trcl} (\text{trss } fg)$ by simp

**from** trss-find-call'(OF this) **obtain** $at ct w1 w2$ where $FC$:

\[
    w=\text{w1} \odot \text{LCall } q#\odot w2
\]

\[ ((\lfloor \text{entry } fg p \rfloor,\{\#\}),w1,(\lfloor \text{ut},ct \rfloor) \in \text{trcl} (\text{trss } fg); \]

\[ ((\lfloor \text{ut},ct \rfloor,\text{LCall } q,\lfloor \text{entry } fg q,v \rfloor,ct) \in \text{trss } fg \]

\[ (\text{ut},\text{Call } q,v) \in \text{edges } fg \]

\[ ((\lfloor \text{entry } fg q \rfloor,ct,w2,(\text{[return } fg q],c') \in \text{trcl} (\text{trss } fg)). \]

**from** trss-drop-all-context[OF FC(5)] **obtain** $\text{csp'}$ where $SLP$: $c'=ct+csp'$

\[ (((\lfloor \text{entry } fg q \rfloor,\{\#\}), w2, (\text{[return } fg q],c') \in \text{trcl} (\text{trss } fg) \text{ by } (\text{auto simp add: union-ac}) \]

**from** FC(1) **have** $\text{LEN}$: $\text{length } w1 \leq \text{length } w \text{ length } w2 \leq \text{length } w$ by auto

**from** mset-map-split-originle SLP(1) **snoe.prems(3) obtain** $P1 \; P2$ where $\text{PSPLIT}$: $P=\lambda p1+\lambda p2$ ($\lambda p. \text{[entry } fg p]) \# P1 \subseteq \# ct$ ($\lambda p. \text{[entry } fg p]) \# P2 

\[
\subseteq \# csp' \text{ by blast}
\]

**with** snoe.prems(2) **have** PSIZE: $\text{size } P1 \leq k \text{ size } P2 \leq k$ by auto

**from** snoe.hyps[OF LEN(1) FC(2) PSIZE(1) PSPLIT(2)] **snoe.hyps[OF LEN(2)] SLP(2) PSIZE(2) PSPLIT(3)] **have** IHAPP: $(\text{ut},\text{mon-w } fg \; w1, P1) \in S\text{-cs } fg k$

88
(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@[c]), 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] e’={#|entry fg q|#}+ch (u,Spaun q;c)∈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,P2)∈S-cs fg k by blast
  have PCOND: P ⊆# {#q#}+P2 proof –
  from PSPLIT(2) have P1⊆#{#q#} by (auto elim!: mset-le-single-cases mset-map-single-rightE)
  with PSIZE(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@[c]), P) ∈ S-cs fg k by (auto simp add: mon-w-unconc)

qed

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

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

lemma (in flowgraph) S-precise: (v,M,P)∈S-cs fg k
  ⊢ ∃ p e’ w.
    (((|entry fg p|,{|#}|),w,([v],c'))∈trcl (trss fg) ∧
     size P≤k ∧
     (λp. [entry fg p]) ‘# P ⊆# e’ ∧
     M=mon-w fg w
  proof (induct rule: S-cs-induct)
    case (S-init p) have (((|entry fg p|,{|#}|),{[v]}),((|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 e’ w where IHAPP: (((|entry fg p|,{|#}|), w, [u], c’)∈trcl (trss fg) size P ≤ k (λp. [entry fg p]) ‘# P ⊆# e’ M = mon-w fg w by blast
  note IHAPP(1)
  also from S-base have ([w],c’),LBase a,[v],c’)∈trss fg by (auto intro: trss-base)
  finally have (((|entry fg p|,{|#}|), w @ [LBase a], [v], c’)∈trcl (trss fg) .

89
moreover from \textit{IHAPP(4)} have $M = \text{mon-w fg} (w \oplus [\text{LBase a}])$ by (simp add: mon-w-unconc)
ultimately show \textit{IHAPP(2,3,4)} by blast
next
case (\textit{S-call u q v M P Ms Ps P'}) then obtain $p \ csp1 w1$ where \textit{REACHING-PATH}: \(((\text{entry fg p}),\{\#\}), w1, [u], \ csp1) \in \text{trcl (trss fg)}$ size $P \leq k \ (\lambda p. \ [\text{entry fg p}])$ ‘# $P \subseteq\# \ csp1 M = \text{mon-w fg} w1$ by blast
from \textit{S-call} obtain $\ csp2 w2$ where \textit{SL-PATH}: \(((\text{entry fg q}),\{\#\}), w2, [\text{return fg q}], \ csp2) \in \text{trcl (trss fg)}$ size $Ps \leq k \ (\lambda p. \ [\text{entry fg p}])$ ‘# $Ps \subseteq\# \ csp2 Ms = \text{mon-w fg} w2$
by (blast dest: trss-er-path-proc-const)
from \textit{trss-c-no-mon[of REACHING-PATH(1)]} \textit{trss-c-no-mon[of SL-PATH(1)]}
have \textit{NOMON: mon-c fg csp1 = }\{\} \mon-c fg csp2 = \{\} by auto
have \textit{((entry fg p),\{\#\}), w1@\text{LCall q}#w2@[\text{LRet}, ([v], csp1+csp2)) \in \text{trcl (trss fg)}} proof
\note{REACHING-PATH(1)}
also from \textit{trss-call[of S-call(1)]} \textit{NOMON have (([u], csp1), \text{LCall q}, ([\text{entry fg q}, v], csp1)) \in \text{trss fg} by (auto)}
also from \textit{trss-add-context[of trss-stack-comp[of SL-PATH(1)]]} \textit{NOMON have (([\text{entry fg q}, v], csp1), w2, ([\text{return fg q}, v], csp1+csp2)) \in \text{trcl (trss fg)} by (simp add: union-ac)}
also have \textit{(([\text{return fg q}, v], csp1+csp2), LRet, ([v], csp1+csp2)) \in \text{trss fg} by (rule trss-ret)}
finally show \textit{?thesis by simp}\qed
moreover from \textit{REACHING-PATH(4)} \textit{SL-PATH(4)} have $\text{mon fg q U M U Ms = mon-w fg (w1@LCall q#w2@[LRet])}$ by (auto simp add: mon-w-unconc)
moreover have $(\lambda p. \ [\text{entry fg p}])$ ‘# $(P') \subseteq\# \ csp1+csp2$ (is ?f ‘# $P' \subseteq\#$ -)
proof –
from \textit{image-mset-subseteq-mono[of S-call(6)]} have ?f ‘# $P' \subseteq\#$ ?f ‘# $P + ?f ‘#$ Ps by auto
also from \textit{mset-subset-eq-mono-add[of REACHING-PATH(3)] \textit{SL-PATH(3)}}
have ... \subseteq\# \ csp1+csp2 .
finally show \textit{?thesis} .\qed
moreover note \textit{S-call(7)}
ultimately show \textit{?case by blast}
next
case (\textit{S-spawn u q v M P P'}) then obtain $p \ c' w$ where \textit{IHAPP}: \(((\text{entry fg p}),\{\#\}), w, [u], c') \in \text{trcl (trss fg)}$ size $P \leq k \ (\lambda p. \ [\text{entry fg p}])$ ‘# $P \subseteq\# \ c' M = \text{mon-w fg} w$ by blast
\note{IHAPP(1)}
also from \textit{S-spawn(1)} have \textit{(([u], c'), LSpawn q, ([v], add-mset [entry fg q] c')) \in \text{trss fg} by (rule trss-spawn)}
finally have \textit{(([entry fg p], \{\#\}), w @ [LSpawn q], [v], add-mset [entry fg q] c') \in \text{trcl (trss fg)} .}
moreover from \textit{IHAPP(4)} have $\text{M=mon-w fg (w @ [LSpawn q])}$ by (simp add: mon-w-unconc)
moreover have $(\lambda p. \ [\text{entry fg p}])$ ‘# $P' \subseteq\# \ \{# [\text{entry fg q}]\#\} + c'$ (is ?f ‘# -
\[ \subseteq \# \) proof \\
from \textbf{image-mset-subseteq-mono}[OF S-spawn(4)] \textbf{have} \ ?f \ ' \ P' \subseteq \# \{\text{entry fg q}\}\#\] + \ ?f + \ P by auto  \\
also from \textbf{mset-subset-eq-mono-add}[OF - IHAPP(3)] \textbf{have} \ldots \subseteq \# \{\text{entry fg q}\}\#\] + c' by (auto intro; IHAPP(3))  \\
finally show ?thesis .  \\
qed
moreover note S-spawn(5)  \\
ultimately show ?thesis by auto  \\
qed

— Finally we can state the soundness and precision as a single theorem

\textbf{theorem} \textbf{(in flowgraph)} \textbf{S-sound-precise}:

\((v,M)\in S-cs fg k \iff
\exists c' w. ([\text{entry fg p}],\#),w,([v],c')\in trcl (trss fg) \land
\text{size } P \leq k \land (\lambda p. [\text{entry fg p}]) \ ' \ P \subseteq \# c' \land M=\text{mon-w fg w})\]

\textbf{using} \textbf{S-sound S-precise by blast}

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

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

\textbf{assumes} A: \((([v],\#),ee,(sh,ch))\in ntrp fg\textbf{ and}
\textbf{CASE:} \exists p u v w. \[
\begin{aligned}
\& eel=LOC (LCall p#w); \\
\& (u,Call p,u')\in edges fg; \\
\& sh=[v,u']; \\
\& \text{proc-of fg v = p}; \\
\& \text{mon-c fg ch = \{\}}; \\
\& !.s. s \in\# ch \implies \exists p u v. s=[\text{entry fg p}] \land \\
\& (u,Spawn p,v)\in edges fg \land \\
\& \text{initialproc fg p}; \\
\& !.P. \exists p u v. s=[\text{entry fg p}] \ ' \ P \subseteq\# ch \implies \\
\& (v,\text{mon-w fg w,P})\in S-cs fg (size P)
\end{aligned}
\]

\textbf{shows} Q

\textbf{proof} —
from A obtain ee where EE: eel=LOC ee (([v],\#),ee,(sh,ch))\in ntrp fg by 
(auto elim: \texttt{ntrp.cases})

\textbf{have} CHFMT: \exists s. s \in\# ch \implies \exists p u v. s=[\text{entry fg p}] \land (u,Spawn p,v)\in edges fg \land \text{initialproc fg p by (auto intro: ntrp-cases-\textbf{OF} EE(2))}

\textbf{with} c-of-initial-no-mon \textbf{have} CHNOMON: mon-c fg ch = \{\} by blast

from EE(2) obtain p u v w where FIRSTSPLIT: eel=LCall p#w (([v],\#),LCall p,([\text{entry fg p},u'],\#))\in trss fg sh=[v,u'] (([\text{entry fg p}],\#),w,([v],ch))\in trcl (trss fg) by (auto elim!: ntrp.cases[simplified])

from FIRSTSPLIT have EDGE: (u,Call p,u')\in edges fg by (auto elim!: \texttt{trss.cases})

from \texttt{trss-bot-prec-const}[where \ s=[] \textbf{ and} \ s'=\[]], simplified, OF FIRSTSPLIT(4)]

\textbf{have} PROC-OF-V: proc-of fg v = p by simp

91
have \( \exists P. (\lambda p. \text{entry \( fg \)} p) \) ‘\# \( P \subseteq \# ch \implies (v, \text{mon-\( w \)} \ fg \ w, P) \in \text{S-cs} \ fg \) (size \( P \))

proof –

\( \exists P \) assume \( (\lambda p. \text{entry } \ fg \ p) \) ‘\# \( P \subseteq \# ch \)

from \( S\)-sound OF FIRSTSPLIT(\( 4 \)) - this, of size \( P \) show ?thesis \( P \) by simp

qed

with \( EE(1) \) FIRSTSPLIT(1,3) EDGE PROC-OF-V CHNOMON CHFMT show \( Q \) by (rule-tac \ CASE) auto

qed

lemma in flowgraph) \( S\)-precise-ntrp:

assumes ENTRY: \((v, M, P) \in \text{S-cs} \ fg \ k \) and

\( P: \text{proc-\( f g \)} \ \text{v} = p \) and

\( \text{EDGE: } (u, \text{Call} \ p, u') \in \text{edges} \ fg \)

shows \( \exists w c h. \)

\((\{(u), \{\#\}\}, \text{LOC} \ (\text{LCall} \ p\#w), \{(v, u'), ch\}) \in \text{ntrp} \ fg \wedge \)

\( \text{size } P \leq k \wedge \)

\( \text{M=} \text{mon-\( w \)} \ \text{fg} \ \text{w} \wedge \)

\( \text{mon-n } \text{fg} \ \text{v} = \text{mon } \text{fg} \ p \wedge \)

\( (\lambda p. \text{entry } \ fg \ p) \) ‘\# \( P \subseteq \# ch \wedge \)

\( \text{mon-c } \ fg \ ch=\{\} \)

proof –

from \( P \) \( S\)-precise[OF ENTRY, simplified] \( \text{trss-bot-proc-const}[\text{where } s=[] \) and \( s'=[], \) simplified] obtain \( \text{wsl ch where} \)

\( \text{SLPATH: } ((\{(u), \{\#\}\}), \text{wsl}, \{v\}, ch) \in \text{trel} \ (\text{trss } fg) \) size \( P \leq k \) \( (\lambda p. \text{entry } \ fg \ p) \) ‘\# \( P \subseteq \# ch \wedge \)

\( \text{M=} \text{mon-\( w \)} \ \text{fg} \ \text{w} \wedge \)

\( \text{mon-n } \text{fg} \ \text{v} = \text{mon } \text{fg} \ p \) by \( \text{fastforce} \)

from \( \text{mon-n-same-proc}[\text{OF trss-bot-proc-const}[\text{where } s=[] \) and \( s'=[], \) simplified, \( \text{OF SLPATH(1)}]) \) have \( \text{MON-V: } \text{mon-n } \text{fg} \ \text{v} = \text{mon } \text{fg} \ p \) by \( \text{(simp)} \)

from \( \text{trss-cases}[\text{OF SLPATH(1)}, \) simplified] have \( \text{CHFMT: } \forall s. s \in \# ch \implies \exists p. s = [\text{entry } \ fg \ p] \wedge (\exists u v. (u, \text{Spawn} \ p, v) \in \text{edges } fg) \wedge \text{initialproc } \text{fg} \ p \) by \( \text{blast} \)

with c-of-initial-no-mon have \( \text{CHNOMON: } \text{mon-c } \ fg \ ch=\{\} \) by \( \text{blast} \)

— From the constraints prerequisites, we can construct the first step

have \( \text{FS: } ((\{(u), \{\#\}\}), \text{LCall} \ p\#\text{wsl}, \{(v, u'), ch\}) \in \text{ntrs} \ fg \) proof \( \text{(rule ntrs-step[where } r=[][], \) simplified])

from \( \text{EDGE show } ((\{(u), \{\#\}\}), \text{LCall} \ p, \text{entry } \ fg \ p, u', \{\#\}) \in \text{trss } fg \) by (auto intro: \( \text{trss-call} \))

qed (rule SLPATH(1))

hence \( \text{FSP: } ((\{(u), \{\#\}\}), \text{LOC} \ (\text{LCall} \ p\#\text{wsl}, \{(v, u'), ch\}) \in \text{ntrp } fg \) by (blast intro: \( \text{gtrp-loc} \))

from \( \text{FSP SLPATH(2,3,4) CHNOMON MON-V show } \) ?thesis by \( \text{blast} \)

qed

9.2 Single reaching path

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

9.2.1 Constraint system

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

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

\[
\text{inductive-set}
\]

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

for \( fg \ U \)

where

\[
RU-init: u \in U \implies (u,\{}\{}),\lambda x.\{}\} \in RU-cs \text{ fg } U
\]

| RU-call: \[(u,\text{Call } p, u) \in \text{edges } fg ; \text{ proc-of } fg \text{ v } = \text{p}; (v,M,P) \in S-cs \text{ fg } 0; (v,Ml,Me,h) \in RU-cs \text{ fg } 0; \text{ mon-n } fg\text{ u } \cap \text{Me } = \{\} \]\n
\[\implies (u, \text{mon } fg\text{ p } \cup M \cup Ml, Me, \text{ah-update } h \text{ (mon } fg\text{ p,M) (Ml \cup Me)}) \\in RU-cs \text{ fg } U\]

| RU-spawn: \[(u,\text{Call } p,u) \in \text{edges } fg ; \text{ proc-of } fg \text{ v } = \text{p}; (v,M,P) \in S-cs \text{ fg } 1; q \in \# P; (\text{entry } fg\text{ q,Ml,Me,h}) \in RU-cs \text{ fg } U; (\text{mon-n } fg\text{ u } \cup \text{mon } fg\text{ p}) \cap (Ml \cup Me) = \{\} \]\n
\[\implies (u,\text{mon } fg\text{ p } \cup M, Ml \cup Me, \text{ah-update } h \text{ (mon } fg\text{ p,M) (Ml \cup Me)}) \\in RU-cs \text{ fg } U\]

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 \(\text{mon-n } fg\text{ u } \cap Me\)
As we only test disjointness with the set of monitors used by the environment, reentrant monitors can be handled. In the spawn case, we have to check disjointness with both, the monitors of local and environment steps of the reaching path from the spawned thread, because from the perspective of the initial thread, all these steps are environment steps ($(\text{mon-n fg } u \cup \text{mon fg } p) \cap (\text{Me}) = \{\}$. 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 $fg$ and $A$: at $U$ $c'$ and $B$: $(c, w, c') \in \text{trcl } (\text{ntr fg})$

**Shows** $\exists s w' c'$. $s \in \# c \wedge w' \preceq w \wedge c' \subseteq c' \wedge a t U \ w' \wedge ((\#s\#), w', c1') \in \text{trcl } (\text{ntr fg})$

**Proof**

- **Obtain** $ui \ r \ ce'$ where $c' = \{\#ui\#r\#\} + ce' \ \text{ui} \in U$ by (rule at $U$-fmt[OF $A$], simp only; mset-contains-eq) (blast dest: sym)

  - **With** ntr-reverse-split[OF - $V$] $B$ obtain $s \ c \ w1 \ c1' \ c2'$ where $\text{RSPLIT}$: $c = \{\#s\#\} + ce \ ce' = ce1' + ce2' \ w \in w1 \wedge (\text{\alpha fg } w2 \ \{\#s\#\}, w1, \{\#ui\#r\#\} + ce1') \in \text{trcl } (\text{ntr fg})$ by blast

  - **With** $\text{C'fmt}$ have $s \in \# c \ w1 \preceq w \{\#ui\#r\#\} + ce1' \subseteq c' \ \text{at } U \ (\{\#ui\#r\#\} + ce1')$

  - **By** (auto dest: cil-ileq)

  - **With** $\text{RSPLIT}(4)$ show $\text{thesis}$ by blast

**Qed**

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

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

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

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

**lemma** (in flowgraph) RU-sound:

$$\forall s' c'. \\{([u],\#), w, (s', c')\} \in \text{trcl (ntrp fg); atU U (add-mset s' c')}\] 

$$\implies \exists Ml Me h. ((u, Ml, Me, h) \in \text{RU-cs fg U} \land Ml \subseteq \text{mon-loc fg w} \land Me \subseteq \text{mon-env fg w} \land h \leq \alpha \text{ah (map (atn fg) w)}$$

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

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

— For a reaching path of length zero, the proposition follows immediately by the constraint RU-init

**case** Nil thus \text{?case by auto (auto intro!: RU-init)}

**next**

**case** (Cons eel wwl)

— For a non-empty path, we regard the first step and the rest of the path

then obtain sh ch where SPLIT:

$$\{([u],\#), eel, (sh, ch)\} \in \text{ntrp fg}$$

$$\{((sh, ch), wwl, (s', c'))\} \in \text{trcl (ntrp fg)}$$

by (fast dest: trcl-uncons)

obtain p u' v w where

— The first step consists of an initial call and a same-level path

**FS-FMT**: $eel = \text{LOC (LCall p \# w)} \quad (u, \text{Call p, u'}) \in \text{edges fg sh} = [v, u']$

$$\text{proc-of fg v} = p \text{ mon-c fg ch} = {}$$

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

and CHFMT: $\forall s. s \in \# ch \implies \exists p u v. s = [\text{entry fg p}] \land (u, \text{Spawn p, v}) \in \text{edges fg} \land \text{initialproc fg p}$

— For the same-level path, we find a corresponding entry in the S-cs-constraint system

and S-ENTRY-PAT: $\forall P. (\lambda p. [\text{entry fg p}]) \quad'\# P \subseteq \# ch \implies (v, \text{mon-w fg w}, P) \in S\text{-cs fg (size P)}$

by (rule S-sound-ntrp[OF SPLIT(1)]) blast

from ntrp-valid-preserve-s[OF SPLIT(1)] have HVALID: $\text{valid fg} \quad\{\#sh\#\} + ch$ by simp

— We split the remaining path by the local thread and the spawned threads,
getting two interleavable paths, one from the local thread and one from the spawned threads

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

LESLPLIT:

\[
\begin{align*}
\text{wwl} &\in w1\ominus\text{and fg} \map ENV w2 \\
c' & = c1' + c2' \\
((sh, \{\#\}), w1, s', c1') &\in \text{trcl (ntrp fg)} \\
(ch, w2, c2') &\in \text{trcl (ntrp fg)} \\
\text{mon-wu fg} & (\text{map le-rem-s w1}) \cap \text{mon-c fg} ch = \{\} \\
\text{mon-ww fg} & w2 \cap \text{mon-s fg} sh = \{\} \\
\end{align*}
\]

by blast

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

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

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

- case left — \( U \) was reached from the local thread

- case right — \( U \) was reached from the spawned thread

from cil-ileq[OF LESPLIT(1)] have ILEQ: \( w1\equiv\text{wwl} \) and \( \text{LEN: } \text{length } w1 \leq \text{length } wwl \) by (auto simp add: le-list-length)

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

from FS-FMT(\( ? \)) LESPLIT(3) ntrp-stack-decomp[of \( v \) \( \{\#\} \) \( w1 \ s' c1' fg, \) simplified] obtain \( v' \rr \) where DECOMP: \( s'=v'\rr\RR[\{\#\},w1,(v'\rr,c1')]\in\text{trcl (ntrp fg)} \) by auto

— This does not affect the configuration being at \( U \)

from atU-exchange-stack left DECOMP(\( 1 \)) have ATU: \( atU \ U \ (\text{add-mset } (v'\rr) \ c1') \) by fastforce

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

from Cons.hyps[OF LEN DECOMP(\( 2 \)) ATU] obtain \( Ml \ Me \ h \) where IHAPP: \( (v, Ml, Me, h) \in RU-cs fg \ U \ Ml \subseteq \text{mon-loc fg} \ w1 \ Me \subseteq \text{mon-env fg} \ w1 \ h \leq \text{aah} (\map (\text{anl fg}) w1) \) by blast

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

from S-ENTRY-PAT[of \( \{\#\} \), simplified] have S-ENTRY: \( (v, \text{mon-w fg} w, \{\#\}) \in S-\text{cs fg} 0 \)

- have MON-U-ME: \( \text{mon-n fg} u \cap Me = \{\} \) proof —

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

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

qed

from RU-call[OF FS-FMT(\( 2,4 \)) S-ENTRY IHAPP(\( 1 \)) MON-U-ME] have \( (u, \\text{mon fg} p \cup \\text{mon-w fg} w \cup Ml, Me, \text{ah-update} h \ (\text{mon fg} p, \text{mon-w fg} w) \ (Ml \cup \text{Me}) \in RU-cs fg U) \)

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

moreover have \( \text{mon fg} p \cup \text{mon-w fg} w \cup Ml \subseteq \text{mon-loc fg} \ (\text{eel}\#\text{wwl}) \) using mon-loc-ileq[OF ILEQ] IHAPP(\( 2 \)) FS-FMT(\( 1 \)) by fastforce

moreover have \( Me \subseteq \text{mon-env fg} \ (\text{eel}\#\text{wwl}) \) using mon-env-ileq[OF ILEQ],
of fg\ \text{IHAPP}(3)\ \text{by} \ \text{auto}
\begin{align*}
\text{moreover have} \ ah\text{-update}\ h\ (\text{mon}\ fg\ p,\ \text{mon-w}\ fg\ w)\ (\text{ML} \cup \text{Me}) \leq \text{aah} & (\text{map (n}\\ell\text{fg)}\ (\text{cel}\#\text{wwl}))\ \text{proof (simp add: ah-update-cons)}
\end{align*}
\begin{align*}
\text{show ah-update}\ h\ (\text{mon}\ fg\ p,\ \text{mon-w}\ fg\ w)\ (\text{ML} \cup \text{Me}) \leq \text{ah-update} & (\text{aah} \ (\text{map (n}\\ell\text{fg)\ wwl}))\ (\text{cel} fg\ \text{cel})\ (\text{mon-pl (map (n}\\ell\text{fg)\ wwl)})\ \text{proof (rule ah-update-mono)}
\end{align*}
\begin{align*}
\text{from IHAPP}(4) & \text{have } h \leq \text{aah} \ (\text{map (n}\\ell\text{fg)\ w1})\ .
\end{align*}
\begin{align*}
\text{also from } \text{aah-ileq}[\text{OF le-list-map}[\text{OF ILEQ}]] & \text{have } \text{aah} \ (\text{map (n}\\ell\text{fg)\ w1})
\text{\leq } \text{aah (map (n}\\ell\text{fg)\ wwl})\ .
\end{align*}
\begin{align*}
\text{finally show } h \leq \text{aah (map (n}\\ell\text{fg)\ wwl})\ .
\end{align*}
\begin{align*}
\text{next from FS-FMT(1) show (mon}\ fg\ p,\ \text{mon-w}\ fg\ w) & = \text{and}\ eel\ \text{by} \ \text{auto}
\end{align*}
\begin{align*}
\text{next from IHAPP}(2,3) & \text{have } (\text{ML} \cup \text{Me}) \leq \text{mon-pl (map (n}\\ell\text{fg)\ w1})\ \text{by (auto simp add: mon-pl-of-cval)}
\end{align*}
\begin{align*}
\text{also from mon-pl-ileq}[\text{OF le-list-map}[\text{OF ILEQ}]] & \text{have } \ldots \leq \text{mon-pl (map (n}\\ell\text{fg)\ wwl})\ .
\end{align*}
\begin{align*}
\text{finally show } (\text{ML} \cup \text{Me}) & \leq \text{mon-pl (map (n}\\ell\text{fg)\ wwl})\ .
\end{align*}
\begin{align*}
\text{qed}
\end{align*}
\begin{align*}
\text{qed}
\end{align*}
\begin{align*}
\text{ultimately show } \text{thesis by} \ \text{blast}
\end{align*}
\text{next}
\begin{align*}
\text{case right} & \text{ — U was reached from the spawned threads}
\end{align*}
\begin{align*}
\text{from cil-ileq}[\text{OF LE-SPLIT(1)] le-list-length[of map ENV w2 wwl} & \text{have ILEQ: map ENV w2\leq wwl and LENV: length w2 \leq length wwl by (auto)}
\end{align*}
\begin{align*}
\text{from HV ALID have CHVALID: valid fg ch mon-s fg sh \cap \\text{mon-c fg ch = \{}}
\end{align*}
\begin{align*}
\text{by (auto simp add: valid-unconc)}
\end{align*}
\begin{align*}
& \text{— We first identify the actual thread from that U was reached}
\end{align*}
\begin{align*}
\text{from ntr-reverse-split-at}\ U[\text{OF CHVALID(1)] right LENV(4)] & \text{obtain q wr cr' where RI: [entry fg q]} \in\#\ \text{ch wr\leq w2 cr'\leq c}\text{2' at U cr'} \text{ (\{[entry fg q}\#\},}\text{wr,cr')}\in\text{trcl (ntr fg) by (blast dest: CHFMT)}
\end{align*}
\begin{align*}
\text{— In order to apply the induction hypothesis, we have to convert the reaching path to loc/env semantics}
\end{align*}
\begin{align*}
\text{from ntrs.qtr\to qtrp[where c=}\text{\{\#\}, } \text{simplified, OF RI(5)] } & \text{obtain sr' cre' wwr where RI-NTRP: cr'=add-mset sr' cre' wwr=map le-rem-s wwr } \text{(([entry fg q]},\text{\{\#\}},}\text{wwr,(sr',cre')})\in\text{trcl (ntrp fg) by blast}
\end{align*}
\begin{align*}
\text{from LENV le-list-length[ORI(2)] } & \text{RI-NTRP(2) have LENV: length wwr \leq length wwl by simp}
\end{align*}
\begin{align*}
& \text{— The induction hypothesis yields a constraint system entry}
\end{align*}
\begin{align*}
\text{from Cons.hyps[ORI LENV'] RI-NTRP(3)] } & \text{RI-NTRP(1) RI(4) obtain } \text{ML Me h where IHAPP: (entry fg q, MI, Me, h)\in}\text{RU-cs fg U MI} \subseteq \text{mon-loc fg wwr Me} \subseteq \text{mon-env fg wwr h \leq aah (map (n}\\ell\text{fg) wwr) by auto}
\end{align*}
\begin{align*}
& \text{— We also have an entry in the same-level path constraint system that contains the thread from that U was reached}
\end{align*}
\begin{align*}
\text{from S-ENTRY-PAT[of } \text{\{\#q\#\}, } \text{simplified] RI(1) have S-ENTRY: (v, mon-w fg w, } \text{\{\#q\#\})} \in \text{S-cs fg I by auto}
\end{align*}
\begin{align*}
& \text{— Before we apply the RU-spawn-constraint, we have to analyze the monitors}
\end{align*}
\begin{align*}
\text{have MON-ML-ENV: MI } & \text{\cup Me} \subseteq \text{mon-loc fg wwr} \cup \text{mon-env fg wwr by auto}
\end{align*}
also from mon-ww-of-le-rem[symmetric] RI-NTRP(2) have \ldots = mon-ww
fg wr by fastforce
also from mon-env-ileq[OF ILEQ] mon-ww-ileq[OF RI(2)] have \ldots \subseteq mon-env
fg wwr by fastforce
finally show \?thesis .

qed
have MON-UP-MLE: (mon-n fg u \cup mon fg p) \cap (Ml \cup Me) = {} proof -
from ntrp-mon-env-w-no-ctz[OF SPLIT(2)] FS-FMT(3,4) edges-part[OF
FS-FMT(2)] have (mon-n fg u \cup mon fg p) \cap mon-env fg wwl = {}
by (auto simp add: mon-n-def)
with MON-MLE-ENV show \?thesis by auto

qed
— Finally we can apply the RU-spawn-constraint that yields us an entry for the
reaching path from a
from RU-spawn[OF FS-FMT(2,4) S-ENTRY - IHAPP(1) MON-UP-MLE]
have \langle 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) \rangle \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 oah (map
\langle anl fg \rangle (eel\#wwl)) — Only the proposition about the acquisition histories needs
some more work
proof (simp add: ah-update-cons)
   have MAP-HELPER: map (anl fg) wwr \leq map (anl fg) wwl proof -
   from RI-NTRP(2) have map (anl fg) wwr = map (anl fg) wr by (simp
add: an-anl)
also from le-list-map[OF RI(2)] have \ldots \leq map (anl fg) w2 .
also have \ldots = map (anl fg) (map ENV w2) by simp
also from le-list-map[OF ILEQ] have \ldots \leq map (anl fg) wwl .
finally show \?thesis .

qed
show ah-update h (mon fg p, mon-w fg w) (Ml \cup Me) \leq ah-update (oah (map
\langle anl fg \rangle wwl)) (anl fg eel) (mon-pl (map (anl fg) wwl)) proof (rule ah-update-mono)
from IHAPP(4) have h \leq oah (map (anl fg) wwr) .
also have \ldots \leq oah (map (anl fg) wwl) by (rule oah-ileq[OF MAP-HELPER])
finally show h \leq oah (map (anl fg) wwl) .

next
from FS-FMT(1) show (mon fg p, mon-w fg w) = anl fg eel by simp
next
from IHAPP(2,3) mon-pl-ileq[OF MAP-HELPER] show Ml \cup Me \subseteq mon-pl
(map (anl fg) wwl) by (auto simp add: mon-pl-of-anl)

qed
qed
ultimately show \?thesis by blast

qed

qed

98
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 (flowgraph) RU-precise**: \((u, Ml, Me, h)\) \(\in\) RU-cs \(\Longleftrightarrow\) \[
\exists w s' c'.
\]
\[
((\{u\}, \{\#\}), w, (s', c')) \in \text{trcl} (\text{ntrp} f g) \land
\]
a\(U\) \(\{\{s'\} \land c'\}\) \land

\[
\text{mon-loc} f g w = Ml \land
\]
\[
\text{mon-env} f g w = Me \land
\]
\[
\text{aah} (\text{map} (\text{anl} f g) w) = h
\]

**Proof** (induct rule: RU-cs.induct)

— The RU-init constraint is trivially covered by the empty path

**Case (RU-init u)** thus \(\text{?case by (auto intro: exI[of - []])}\)

**Next**

— Call constraint

**Case (RU-call u p w v M P Ml Me h)**

**Then obtain** \(w s' c'\) where IHAPP: \((\{v\}, \{\#\}), w, s', c'\) \(\in\) \(\text{trcl} (\text{ntrp} f g)\) at \(U\) \(\{\{s'\} \land c'\}\) \(\text{mon-loc} f g w = Ml\land \text{mon-env} f g w = Me\land \text{aah} (\text{map} (\text{anl} f g) w) = h\) by blast

**From** RU-call.hyps(2) S-precise\(\text{OF RU-call.hyps(3), simplified}\) trss-bot-prec-const\(\text{where s=[], and s'=[], simplified}\) obtain \(\text{wsl} ch\) where

SLPATH: \(\{\{\text{entry} f g p\}, \{\#\}\}, \text{wsl} [v], ch\) \(\in\) \(\text{trcl} (\text{trss} f g)\) \(M = \text{mon-w} f g \text{wsl}\) by fastforce

**From** trss-cases\(\text{OF SLPATH(1), simplified}\) have CHFMT: \(\exists s. s \in \# ch \Longrightarrow \exists u. s = [\text{entry} f g p] \land (\exists u v. (u, \text{Spawn} p, v) \in \text{edges} f g) \land \text{initialproc} f g p\) by blast

with c-of-initial-no-mon have CHNOMON: \(\text{mon-c} ch = \{\}\) by blast

— From the constraints prerequisites, we can construct the first step

**Have** FS: \((\{u\}, \{\#\}), L\text{Call} p\#wsl, ([v, u'], ch)\) \(\in\) ntrp f g **Proof** (rule ntrss-step\(\text{where r=[], simplified}\))

**From** RU-call.hyps(1) show \((\{u\}, \{\#\}), L\text{Call} p, [\text{entry} f g p, u'], \{\#\}) \(\in\) trss f g by (auto intro: trss-call)

**Qed** (rule SLPATH(1))

**Hence** FSP: \((\{u\}, \{\#\}), L\text{Call} p\#wsl, ([v, u'], ch)) \(\in\) ntrp f g by (blast intro: gtrlp-loc)

**Also**

— The rest of the path comes from the induction hypothesis, after adding the
rest of the threads to the context

**Have** \((\{v, u'\}, ch), w, s' \in [u], c' + ch\) \(\in\) \(\text{trcl} (\text{ntrp} f g)\) **Proof** (rule ntrp-add-context\(\text{OF ntrp-stack-comp}(\text{OF IHAPP(1)}, \text{where r=[u']}, \text{where cn=ch, simplified})\))

**From** RU-call.hyps(1,6) IHAPP(4) show \(\text{mon-n} f g u' \cap \text{mon-env} f g w = \{\}\) 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 \(((\{u\}, \{\#\}), LOC (LCall p \# wsl) \# w, s' @ [u']_w, c' + ch) \in trcl (ntrp $fg$)\).

— 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']_w\#\} + (c'+ch)) by auto

moreover have mon-loc $fg$ (LOC (LCall p \# wsl) \# w) = mon $fg$ $p$ \cup $M$ \cup $MI$ using SLPATH(2) IHAPP(3) by auto

moreover have mon-env $fg$ (LOC (LCall p \# wsl) \# w) = $Me$ using IHAPP(4) by auto

moreover have oah (map (oahl $fg$) (LOC (LCall p \# wsl) \# w)) = ah-update $h$ (mon $fg$ $p$, $M$) (MI \cup $Me$) proof —

have oah (map (oahl $fg$) (LOC (LCall p \# wsl) \# w)) = ah-update (oah (map (oahl $fg$) $w$)) (mon $fg$ $p$, mon-w $fg$ wsl) (mon-pl (map (oahl $fg$) $w$)) by (auto simp add: ah-update-cons)

also have \(=\) ah-update $h$ (mon $fg$ $p$, $M$) (MI \cup $Me$) proof —

from IHAPP(5) have oah (map (oahl $fg$) $w$) = $h$ .

moreover from SLPATH(2) have (mon $fg$ $p$, mon-w $fg$ wsl) = (mon $fg$ $p$, $M$) by (simp add: mon-pl-of-oval)

moreover from IHAPP(3,4) have mon-pl (map (oahl $fg$) $w$) = $MI$ \cup $Me$ by (auto simp add: mon-pl-of-oval)

ultimately show \(?thesis\) by simp

qed

finally show \(?thesis\) .

qed

ultimately show \(?case\) by blast

next

— Spawn constraint

case (RU-spawn $u p u' v$ $M P q$ $MI Me h)$ then obtain $w$ $s'$ $c'$ where IHAPP:

\(((\{entry $fg$ $q\}, \{\#\}), w, s', c') \in trcl (ntrp $fg$) \atU U (\{\#s'\#\} + c')\) mon-loc $fg$ $w$ = $MI$ mon-env $fg$ $w$ = $Me$ oah (map (oahl $fg$) $w$) = $h$ by blast

from RU-spawn-hyps(2) S-precise[OF RU-spawn-hyps(3), simplified] trss-bot-proc-const[where $s'[]=\$ and $s'=\$, simplified] obtain wsl $ch$ where

SLPATH: \(((\{entry $fg$ $p\}, \{\#\}), wsl, [v], ch) \in trcl (trss $fg$) M = mon-w $fg$ wsl size $P \leq 1 (\lambda p. [entry $fg$ $p$] (# q#)) ch = (# [entry $fg$ $q$] #) + che by (auto elim!: mset-size-le1-cases mset-le-addE)

from trss-cases[OF SLPATH(1), simplified] have CHFMT: \(\forall s. s \in # \ ch \Rightarrow \exists p. s = [entry $fg$ $p] \land (\exists u v. (u, Spawn p, v) \in edges $fg$) \land initialproc $fg$ $p$ by blast

with c-of-initial-no-mon have CHNOMON: mon-c $fg$ $ch$ = \{\} by blast

have FS: \(((\{u\}, \{\#\}), LCall p#wsl, ([v, u'], ch)) \in ntrp $fg$ proof (rule ntrs-step[where $r=[]$, simplified])

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

qed (rule SLPATH(1))

hence FSP: \(((\{u\}, \{\#\}), LOC (LCall p#wsl), ([v, u', ch]) \in ntrp $fg$ by (blast intro: gtrp-loc)
also have \(((v, \ u'), \ ch), \ map \ ENV \ (map \ le-rem-s \ w), \ [v, u'], \ che+\{\#s'\#\}+c') \in \ trcl \ (ntrp \ fg)\) proof –

from \(IHAPP(3,4)\) have \(mon-ww \ fg \ (map \ le-rem-s \ w) \subseteq \ Ml \cup \ Me\) by (auto simp add: mon-ww-of-le-rem)

with \(RU\text{-}spawn\_hyps(1,2,7)\) have \(\{\\text{mon-n} \ fg \ v \cup \ \text{mon-n} \ fg \ u'\} \cap \ \text{mon-ww} \ fg \ (map \ le-rem-s \ w) = \{\}\) by (auto simp add: mon-n-def edges-part)

with \(ntrp\text{-}\text{bin}\_\text{of}\_\text{OP} \ \text{gtrp}2\text{gtr}[OF \ \text{IHAPP}(1)], \ of \ [v,u'] \ \text{che}\) \(\text{PFMT}(2)\) CHNOMON show \(\text{thesis}\) by (auto simp add: union-ac mon-c-unconc)

qed

finally have \(((\{u\}, \ \{\#\}), \ \text{LOC} \ (LCall \ p \ \# \ wsl) \ # \ \text{map} \ ENV \ (map \ le-rem-s \ w), \ [v, u'], \ che+\{\#s'\#\}+c') \in \ trcl \ (\text{ntrp} \ fg)\). 

moreover from \(IHAPP(2)\) have \(\text{atU} \ U \ (\{\#[v,u]\#\} + \ (che+\{\#s'\#\} + c'))\) by auto

moreover have \(\text{mon-loc} \ fg \ (\text{LOC} \ (LCall \ p \ \# \ wsl) \ # \ \text{map} \ ENV \ (map \ le-rem-s \ w)) = \text{mon} \ fg \ p \cup \ M \) using \(\text{SLPATH}(2)\) by (auto simp del: map-map)

moreover have \(\text{mon-env} \ fg \ (\text{LOC} \ (LCall \ p \ \# \ wsl) \ # \ \text{map} \ ENV \ (map \ le-rem-s \ w)) = \text{Ml} \cup \ Me\) using \(\text{IHAPP}(3,4)\) by (auto simp add: mon-ww-of-le-rem simp del: map-map)

moreover have \(\text{oaah} \ (\text{map} \ (\text{onl} \ fg)) \ (\text{LOC} \ (LCall \ p \ \# \ wsl) \ # \ \text{map} \ ENV \ (\text{map} \ le-rem-s \ w))) = \text{ah}\_\text{update} \ h \ (\text{mon} \ fg \ p, \ M) \) (\(\text{Ml} \cup \ Me\)) proof –

have \(\text{oaah} \ (\text{map} \ (\text{onl} \ fg)) \ (\text{LOC} \ (LCall \ p \ \# \ wsl) \ # \ \text{map} \ ENV \ (\text{map} \ le-rem-s \ w))) = \text{ah}\_\text{update} \ \text{(oaah} \ (\text{map} \ (\text{onl} \ fg)) \ (\text{map} \ le-rem-s \ w))) \) (\(\text{mon} \ fg \ p, \ \text{mon-w} \ fg \ wsl\)) (\(\text{mon-pl} \ (\text{map} \ (\text{onl} \ fg)) \ (\text{map} \ le-rem-s \ w))\) by (simp add: ah-update-cons o-assoc)

also have \(\ldots = \text{ah}\_\text{update} \ h \ (\text{mon} \ fg \ p, \ M) \) (\(\text{Ml} \cup \ Me\)) proof –

from \(IHAPP(5)\) have \(\text{oaah} \ (\text{map} \ (\text{onl} \ fg)) \ (\text{map} \ le-rem-s \ w)) = h\) by (simp add: an-anl)

moreover from \(\text{SLPATH}(2)\) have \(\text{mon} \ fg \ p, \ \text{mon-w} \ fg \ wsl\) = \(\text{mon} \ fg \ p, \ M\) by simp

moreover from \(\text{IHAPP}(3,4)\) have \(\text{mon-pl} \ (\text{map} \ (\text{onl} \ fg)) \ (\text{map} \ le-rem-s \ w)) = \text{Ml} \cup \ Me\) by (auto simp add: mon-pl-of-anl an-anl)

ultimately show \(\text{thesis}\) by simp

qed

finally show \(\text{thesis}\).

qed

ultimately show \(\text{case}\) 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 \((u, \ Ml, \ Me) \in RU\text{-}cs \ fg \ U \ V\) means, that there is a path from \(\{\#u\}\) to some configuration that is simultaneously at \(U\) and at \(V\). That path uses monitors from \(Ml\) in the first thread and monitors from \(Me\) in the
other threads.

**inductive-set**

\[ RUV-cs :: (\{n, p, ba, m, more\}) \implies \{n \implies \{q \times m \times m \times m\} \}\]  

**for** \(fg U V\)  

**where**

\[ RUV-call:\]

\[
\begin{array}{l}
(u, Call_p, u') \in \text{edges } fg; \quad \text{proc-of } fg v = p; \quad (v, M, P) \in S-cs fg 0; \\
(v, M, M_e, h) \in RUV-cs fg U V; \quad \text{mon-n} fg u \cap Me = \{\} \\
\Rightarrow (u, mon fg p \cup M \cup M_e) \in RUV-cs fg U V \\
\end{array}
\]

\[ RUV-spawn:\]

\[
\begin{array}{l}
(u, Call_p, u') \in \text{edges } fg; \quad \text{proc-of } fg v = p; \quad (v, M, P) \in S-cs fg 1; \quad q \in \# P; \\
\quad (entry fg q, M_l, M_e) \in RUV-cs fg U V; \\
\quad (\text{mon-n} fg u \cup \text{mon fg } p) \cap (M_l \cup M_e) = \{\} \\
\Rightarrow (u, mon fg p \cup M \cup M_e) \in RUV-cs fg U V \\
\end{array}
\]

\[ RUV-split-le:\]

\[
\begin{array}{l}
(u, Call_p, u') \in \text{edges } fg; \quad \text{proc-of } fg v = p; \quad (v, M, P) \in S-cs fg 1; \quad q \in \# P; \\
\quad (v, M_l, M_e, h) \in RUV-cs fg U V; \quad (entry fg q, M_l', M_e', h') \in RUV-cs fg U V; \\
\quad (\text{mon-n} fg u \cup \text{mon fg } p) \cap (\text{Me} \cup M_l \cup M_e) = \{\} \cup \{h \cup h'\} \\
\Rightarrow (u, mon fg p \cup M \cup M_l \cup M_e \cup M_l' \cup M_e') \in RUV-cs fg U V \\
\end{array}
\]

\[ RUV-split-el:\]

\[
\begin{array}{l}
(u, Call_p, u') \in \text{edges } fg; \quad \text{proc-of } fg v = p; \quad (v, M, P) \in S-cs fg 1; \quad q \in \# P; \\
\quad (v, M_l, M_e, h) \in RUV-cs fg U V; \quad (entry fg q, M_l', M_e', h') \in RUV-cs fg U V; \\
\quad (\text{mon-n} fg u \cup \text{mon fg } p) \cap (\text{Me} \cup M_l \cup M_e) = \{\} \cup \{h \cup h'\} \\
\Rightarrow (u, mon fg p \cup M \cup M_l \cup M_e \cup M_l' \cup M_e') \in RUV-cs fg U V \\
\end{array}
\]

\[ RUV-split-ee:\]

\[
\begin{array}{l}
(u, Call_p, u') \in \text{edges } fg; \quad \text{proc-of } fg v = p; \quad (v, M, P) \in S-cs fg 2; \\
\quad \{\# q\} \cup \{\# q'\} \subseteq \# P; \\
\quad (entry fg q, M_l, M_e, h) \in RUV-cs fg U V; \quad (entry fg q', M_l', M_e', h') \in RUV-cs fg U V; \\
\quad (\text{mon-n} fg u \cup \text{mon fg } p) \cap (\text{Me} \cup M_l \cup M_e) = \{\} \cup \{h \cup h'\} \\
\Rightarrow (u, mon fg p \cup M \cup M_e \cup M_l \cup M_l' \cup M_e') \in RUV-cs fg U V \\
\end{array}
\]

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

**RUV-call** Both, \(U\) and \(V\) are reached from the initial thread.

**RUV-spawn** Both, \(U\) and \(V\) are reached from a single spawned thread.

**RUV_split_le** \(U\) is reached from the initial thread, \(V\) is reached from a spawned thread.
RUV\_split\_el  \( V \) is reached from the initial thread, \( U \) is reached from a spawned thread.

RUV\_split\_ee  Both, \( U \) and \( V \) are reached from different spawned threads.

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

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

9.3.2 Soundness and precision

\textbf{Lemma (in flowgraph) \( RUV\)-sound:} \( !!u \ s' c' \).

\[ (([u],\{\#\}),v,(s',c)) \in trcl (ntrp fg); \text{ atUV} \ U \ V \ (\{\#s'\#\}+c') \] \[ \implies \exists Ml Me. \]

\((u,Ml,Me) \in RUV\_cs \ fg \ U \ V \ \wedge\
\ Ml \subseteq \text{mon-loc} \ fg \ w \wedge\
\ Me \subseteq \text{mon-env} \ fg \ w\]

— The soundness proof is done by induction over the length of the reaching path

\textbf{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

\textbf{Case Nil hence False by simp thus ?case ..

next}

\textbf{Case (Cons ee ww) then simp \( ?\) case ..

next}

\textbf{obtain} \( sh \ ch \ where \ SPLIT: (([u],\{\#\}),ee,(sh,ch)) \in ntrp fg ((sh,ch),ww,(s',c')) \in trcl (ntrp fg) \)

\textbf{by (fast dest: trcl-uncons)}

\textbf{from ntrp-split[where \( ?c.0=\{\#\}, \text{simplified}, \text{OF SPLIT(1)} \) ntrp-valid-preserve-s[OF SPLIT(1)], \text{simplified}] obtain} \( w1 \ w2 \ c1' \ c2' \ where \)

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

\( \text{by blast} \)

\textbf{obtain} \( p \ u' \ v \ w \ where \)

\( \text{FS-FMT: } ee = \text{LOC} (LCall p \ \# w) \ (u, \text{ Call p, } u') \in \text{edges} \ fg \ sh = \{v, u\}

\text{proc-of} fg \ v = p \text{ mon-c fg ch = } \{\}

\text{and} CHFMT: \{s, s \in ch \implies \exists p u v. s = [entry fg p] \wedge (u, \text{ Spawn p, } v) \in \text{edges} \ fg \wedge \text{initialproc fg p}

\text{and} S-ENTRY-PAT: \{P, (\lambda p. [entry fg p]) \# P \subseteq ch \implies (v, \text{ mon-w fg w, P}) \in S\text{-cs} fg \ (size } P) \}

\text{by (rule S-sound-ntrp[OF SPLIT(1)]) blast}

\textbf{from ntrp-mon-env-w-no-ctx[OF SPLIT(2)] FS-FMT(3,4) edges-part[OF FS-FMT(2)] have} \ MON\_PU: \text{ mon-env fg w w} \cap \text{ (mon fg p } \cup \text{ mon-n fg u) = } \{\}

\text{by (auto simp add: mon-n-def)}

\textbf{from cil-ileq[OF LESPLIT(1)] mon-loc-ileq[of w1 ww fg] mon-env-ileq[of w1 ww fg] have} \ MON1\_LEQ: \text{ mon-loc fg w1 } \subseteq \text{ mon-loc fg w w} \text{ mon-env fg w1 } \subseteq \text{ mon-env fg w w by auto}

\textbf{from cil-ileq[OF LESPLIT(1)] mon-env-ileq[of map ENV w2 ww fg] have} \ MON2\_LEQ: \text{ mon-ww fg w2 } \subseteq \text{ mon-env fg w w by simp}
from LESPLIT(3) FS-FMT(3) ntrp-stack-decomp[of v [] [u'] {} w1 s' c1', simplified] obtain v'rr where DECOMP-LOC: s'=v'rr@[u'] (([v],{}),w1,(v'rr,c1'))∈trcl (ntrp fg) by (simp, blast)

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

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

case left with DECOMP-LOC(1) have ATUV: atUV U V (({# s'## }+c1')

by simp

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

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

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

moreover have Me ⊆ mon-env fg (ce#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-s U s1' at-U-s V s2' by (unfold atUV-def) auto

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

obtain q ceh w21 w22 ce21' ce22' where

REVSPLIT: ch={#|entry fg q|##}+ceh add-mset s2' ce2' = ce21'+ce22', w2∈w21⊗αn fgw22 mon fg q ∩ (mon-c fg ceh ∪ mon-ww fg w22)={} mon-c fg ceh ∩ (mon fg q ∪ mon-ww fg w21) = {}

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

proof —

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

obtain seh ceh w21 w22 ce21' ce22' where

*: ch={#|seh|##}+ceh {#s2'##}+ce2' = ce21'+ce22', w2∈w21⊗αn fgw22 mon-s fg seh ∩ (mon-c fg ceh ∪ mon-ww fg w22)={} mon-c fg ceh ∩ (mon-s fg seh ∪ mon-ww fg w21) = {}

({#seh|##},w21{#s1'##}+ce21')∈trcl (ntr fg) (ceh,w22,ce22')∈trcl (ntr fg)

by (auto simp add: valid-unc onc)

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

with * have ch={#|entry fg q|##}+ceh add-mset s2' ce2' = ce21'+ce22', w2∈w21⊗αn fgw22 mon fg q ∩ (mon-c fg ceh ∪ mon-ww fg w22)={} mon-c fg ceh ∩ (mon fg q ∪ mon-ww fg w21) = {}

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

thus thesis using that by (blast)
\textbf{qed}

— For applying the induction hypothesis, it will be handy to have the reaching path in \texttt{loc/env} format:

\begin{verbatim}
from ntrsp,gr2gtrp\{where c={#}, simplified, OF REVSPILT(6)\} obtain sq’
csp-q w21 where
R-CONV: add-mset s’ ce21’ = add-mset sq’ csp-q w21 = map le-rem-s w21
((\{entry fg q\}, {#}), w21, sq’, csp-q) \in trcl (ntrp fg) by auto
from csp-q \{OF REVSPILT(3)\} 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 \{thesis\} proof \{cases rule: mset-unplusm-dist-cases\{case-names left’ right’\}
  case left’ — Both nodes are reached from the same thread
  have ATUV: atUV U V \{(\#sq’#)\}+csp-q using right C2’FMT R-STACKS(2,3)
  left’\{(1\}
    by (metis R-CONV(1) add-mset-add-single atUV-union atU-add-mset
    union-commute)
  from Cons.hyps[of - R-CONV(3) ATUV] cil-length[of REVSPILT(3)]
cil-length[of LESPLIT(1)] R-CONV(2) obtain Ml Me where IHAPP: (entry fg q
  Ml, Me) \in RUV-cs fg U V Ml \subseteq mon-loc fg w21 Me \subseteq mon-env fg w21 by auto
  from REVSPILT(1) S-ENTRY-PAT[of \{(\#q#)\}, simplified] have S-ENTRY:
    \{v, mon-w fg w, \{(\#q#)\} \in S-cs fg 1 by simp
      have MON-COND: (mon-n fg u \cup mon fg p) \cap (Ml \cup Me) = \{\} proof —
        from R-CONV(2) have mon-ww fg w21 = mon-loc fg w21 \cup mon-env fg
        w21 by (simp add: mon-ww-of-le-rem)
          with IHAPP(2,3) MON2N-LEQ(1) MON-PU MON2-LEQ show \{thesis\}
        by blast
      qed
    from RUV-spawn[of FS-FMT(2) FS-FMT(4) S-ENTRY - IHAPP(1)
      MON-COND] have \(\{u, mon-fg p \cup mon-w fg w, Ml \cup Me\} \in RUV-cs fg U V
    by simp
      moreover have mon fg p \cup mon-w fg w \subseteq mon-loc fg (ce\#ww) using
      FS-FMT(1) by auto
      moreover have Ml \cup Me \subseteq mon-env fg (ce\#ww) using IHAPP(2,3)
      R-CONV(2) MON2N-LEQ(1) MON-PU MON2-LEQ by (auto simp add: mon-ww-of-le-rem)
      ultimately show \{thesis\} by blast
    next
    case right’ — The nodes are reached from different threads
      from R-STACKS(2,3) have ATUV: atU U (add-mset sq’ csp-q) atU V ce22’
        by (-) (subst R-CONV(1)symmetric], simp, subst right’(1), simp)
        — We have to reverse-split the second path again, to extract the second
        interesting thread
          obtain q’ w22’ ce22’e where REVSPILT’: \{entry fg q\} \in\# cch w22’\leq w22
          ce22’ \subseteq\# ce22’ atU V ce22’e’ \{(\#\{entry fg q’\}\#), w22’,ce22’e’\} \in trcl (ntrp fg)
        proof —
          from ntr-reverse-split-atU[of - ATUV(2) REVSPILT(7)] ntrp-valid-preserve-s[of
          SPLIT(1), simplified] REVSPILT(1) obtain sq’” w22’ ce22’e’ where
\end{verbatim}
\[\text{\textbf{From the soundness of the RU-constraint system, we get the corresponding entries}}\]

\[\text{\textbf{from RU-sound[OF R-CONV(9) ATUV(1)] obtain MI Me h where RU:}}\]

\[(entry\ fg\ q,\ Mi,\ Me,\ h)\in\text{RU-CS fg U MI} \subseteq \text{mon-loc fg w21 Me} \subseteq \text{mon-env fg w21} \ h \leq \alphaah\ (map\ (\alpha\ n\ fg)\ w21)\ by\ blast\]

\[\text{\textbf{from RU-sound[OF R-CONV'(3), of V] REVSPILIT'(4) R-CONV'(1) obtain MI' Me' h' where RV:}}\]

\[(entry\ fg\ q',\ MI',\ Me',\ h')\in\text{RU-CS fg V MI'} \subseteq \text{mon-loc fg w21'}\]

\[(\alpha\ n\ fg)\ w21' \leq \alphaah\ (map\ (\alpha\ n\ fg)\ w21')\ by\ auto\]

\[\text{\textbf{from S-ENTRY-PAT[of \{\#q\}\ +\{\#q'\}, simplified] REVSPILIT'(1) REVSPILIT'(1) have S-ENTRY:}}\]

\[\\text{S-ENTRY:}}\ (v,\ mon-w\ fg\ w,\ \{\#q\}\ +\{\#q'\})\in\text{S-CS fg (2 :: nat)}\]

\[\\text{by\ simp\ add:\ numerals}}\]

\[\text{\textbf{have \(u,\ mon-w\ fg\ w,\ Mi'\cup Me'\cup Me'\cap Me\) \in RUV-CS fg U V}}\]

\[\text{\textbf{proof}}\ (\text{rule RUV-split-ce[OF FS-FMT(2,4) S-ENTRY - RU(1) RV(1)]})\]

\[\text{\textbf{from MON-PU MON2-LEQ MON2N-LEQ R-CONV(2) REVSPILIT'(2) OF REVSPILIT(2), of fg] RU(2,3) RV(2,3) show}}\]

\[\text{\textbf{mon-ww-ileq}}\ (\text{mon-n fg u} \cup\ \text{mon-n fg p})\cap (\text{MI} \cup Me \cup Mi' \cup Me') = \{\} \text{ by simp add: mon-ww-ileq) blast next}}\]

\[\text{\textbf{from ah-interleavable[OF REVSPILIT(3)] have \alphaah\ (map\ (\alpha\ n\ fg)\ w21)\ [\star]}\]

\[\\text{\alphaah\ (map\ (\alpha\ n\ fg)\ w22).}}\]

\[\\text{thus \(h\ [\star] h')}}\]

\[\\text{proof\ (erule_tac ah-leq-id)}\]

\[\\text{note RU(4)}\]

\[\\text{also have map\ (\alpha\ n\ fg)\ w21 \leq map\ (\alpha\ n\ fg)\ w21\ using\ R-CONV(2) by}}\]

\[\\text{simp\ add: an\ n\ fg}]

\[\\text{hence \alphaah\ (map\ (\alpha\ n\ fg)\ w21) \leq \alphaah\ (map\ (\alpha\ n\ fg)\ w21)\ by\ (rule \alphaah-ileq)}\]

\[\\text{finally show} \ h \leq \alphaah\ (map\ (\alpha\ n\ fg)\ w21)\ .}\]

\[\\text{next}}\]

\[\\text{note RV(4)}\]

\[\\text{also have map\ (\alpha\ n\ fg)\ w22' \leq map\ (\alpha\ n\ fg)\ w22\ using\ R-CONV'(2) REVSPILIT'(2) by\ (simp\ add: an\ n\ [symmetric] le-list-map map-map[symmetric]}}\]

\[\\text{del: map-map)}\]

\[\\text{hence \alphaah\ (map\ (\alpha\ n\ fg)\ w22') \leq \alphaah\ (map\ (\alpha\ n\ fg)\ w22)\ by\ (rule \alphaah-ileq)}\]

\[\\text{finally show} \ h' \leq \alphaah\ (map\ (\alpha\ n\ fg)\ w22)\ .}\]

\[\text{qed}\]

\[\text{qed\ (simp)}\]

\[\text{moreover have mon\ fg\ p \cup\ mon-n\ fg\ w} \subseteq \text{mon-loc\ fg\ (ee\ #\ wv)\ using}}\]

\[\text{FS-FMT(1) by auto}}\]

\[\text{moreover have MI \cup Me \cup Mi' \cup Me' \subseteq mon-env\ fg\ (ee\ #\ wv)\ using RV(2,3)}\]
\(RU(2,3)\) mon-ww-ileq(OF REV_SPLIT'(2), of fg) MON2N-LEQ R-CONV(2) R-CONV'(2)

MON2-LEQ by (simp add: mon-ww-of-le-rem) blast

ultimately show thesis by blast

qed

next

- 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

\[\text{ML} \ \text{Me} \ \text{h} \ \text{where} \ \text{RU}: \ (v, \text{ML}, \text{Me}, \text{h}) \in \text{RU-cs fg U ML} \subseteq \text{mon-loc fg w1 Me} \subseteq \text{mon-env fg w1 h} \leq \text{alpha (map (\alpha fg) w1)} \] by auto

obtain \(\text{ML'} \ \text{Me'} \ \text{h'} \ \text{q'} \ \text{where} \ \text{RV}: \) (entry fg q') \in # ch (entry fg q', ML', Me', h') in RU-cs fg V ML' \subseteq mon-ww fg w2 Me' \subseteq mon-ww fg w2 h' \leq alpha (map (\alpha fg) w2)

proof

- We have to extract the interesting thread from the spawned threads in order to get an entry in RU fg V

obtain q' w2' c2i' where REV_SPLIT: (entry fg q') \in # ch w2' \sqsubseteq w2 c2i' \subseteq # c2i' at U V c2i' ((#entry fg q'#), w2', c2i') \in trcl (ntr fg)

using ntr-reverse-split-atU[OF - br(2) LESPLIT(4)] ntrp-valid-preserve-s[OF SPLIT(1), simplified] CHFMT by (simp add: valid-unconc) blast

from ntrs QR2qtrp[where c=\{\}, simplified, OF REV_SPLIT(5)] obtain s2i' c2ie' w2' where R-CONV: c2i'=add-mset s2i' c2ie' w2'=map le-rem-s w2' (((entry fg q'), \{\})), w2', s2i', c2i') \in trcl (ntrp fg).

from RU-sound[OF REV_SPLIT(3), of V] REV_SPLIT(4) R-CONV(1) obtain

\(\text{ML'} \ \text{Me'} \ \text{h'} \ \text{where} \ \text{RV}: \) (entry fg q', ML', Me', h') \in RU-cs fg V ML' \subseteq mon-loc fg w2' Me' \subseteq mon-env fg w2' h' \leq \text{alpha (map (\alpha fg) w2') by auto}

moreover have mon-loc fg w2' \subseteq mon-ww fg w2 mon-env fg w2' \subseteq mon-ww fg w2 using mon-ww-ileq[OF REV_SPLIT(2), of fg] R-CONV(2) by (auto simp add: mon-ww-of-le-rem)

moreover have alpha (map (\alpha fg) w2') \leq alpha (map (\alpha fg) w2) using REV_SPLIT(2) R-CONV(2) by (auto simp add: \alpha-\alpha[\text{symmetric}] le-list-map map-map[\text{symmetric}] simp del: map-map intro: alpha-ileq del: predicate21)

ultimately show thesis using that REV_SPLIT(1) by (blast intro: order-trans)

qed

from S-ENTRY-PAT[of \{\#q\#\}, simplified] RV(1) have S-ENTRY: (v, mon-w fg w, \{\#q\#\}) \in S-cs fg \ V by simp

have \(\{u, \text{mon fg p} \cup \text{mon-w fg w } \cup \text{ML, Me \cup ML' } \cup \text{Me'}\} \in \text{RV-cs fg U V}

proof (rule REV_SPLIT(OF FS-FMT(2,4) S-ENTRY - RU(1) RV(2)))

from MON-PU MON1-LEQ MON2-LEQ RU(3) RV(3,4) show (mon-n fg u \cup mon fg p) \cap (\text{Me } \cup \text{ML' } \cup \text{Me'}) = \{\} by blast

next

from ah-interleavable[OF LESPLIT(1)] have alpha (map (\alpha fg) w1) [\*] alpha (map (\alpha fg) w2) by simp

thus h \[\*\] h' using RV(4) RV(5) by (auto elim: ah-ileq-il)

qed (simp)

moreover have mon fg p \cup mon-w fg w \cup ML \subseteq mon-loc fg (ee # ww) using

FS-FMT(1) MON1-LEQ RU(2) by (simp) blast

moreover have Me \cup ML' \cup Me' \subseteq mon-env fg (ee # ww) using MON1-LEQ MON2-LEQ RU(3) RV(3,4) by (simp) blast

107
ultimately show \( ?\thesis \) by \texttt{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 \( \text{RU-sound|OF DECOMP-LOC(2), of V|} \) \( rl(1) \) \( \text{DECOMP-LOC(1)} \) obtain \( Ml \) \( Me \) \( h \) where \( RV: (v, Ml, Me, h) \in \text{RU-cs fg V Ml} \subseteq \text{mon-loc fg w1 Me} \subseteq \text{mon-env fg w1 h} \leq \text{aah (map (\alpha nl fg) w1)} \) by auto

obtain \( Ml' \) \( Me' \) \( h' \) \( q' \) where \( RU: [\entry fg q'] \in \# ch (\entry fg q', Ml', Me', h') \in \text{RU-cs fg U Ml'} \subseteq \text{mon-ww fg w2 Me'} \subseteq \text{mon-ww fg w2 h'} \leq \text{aah (map (\alpha nl fg) w2)} \)

proof –

— We have to extract the interesting thread from the spawned threads in order to get an entry in \( RU \) \( fg \) \( V \)

obtain \( q' \) \( w2' \) \( c2l' \) where \( \text{REVSPILT: [entry fg q'] \in \# ch w2' \leq w2 c2l' \subseteq \# c2l' at U U c2l' } \) \( \{\#\}\{\entry fg q'\}\{\#\} \) \( w2',c2l' \in \text{trcl (ntr fg)} \)

using \( \text{ntr-reverse-split-atU|OF - rl(2) LESPLIT(4)} \) \( \text{ntrp-valid-preserve-s|OF SPLIT(1), simplified} \) \( \text{CHFMT by (simp add: valid-unconc) blast} \)

from \( \text{ntrp.| U | \text{REVSPILT(5)|simplified)} \) \( \text{OF REVSPILT(5)} \) obtain \( s2l' \) \( c2l' c2l'i \) \( \text{ww2' where R-CONV: c2l'i = add-set s2l'i c2l'i = map le-rem-s \text{ww2'} ((\{\entry fg q'\}, \{\#\})) \text{ww2', s2l'i e c2l'i} \in \text{trcl (ntrp fg)} .} \)

from \( \text{RU-sound|OF R-CONV(3), of U| REVSPILT(4) R-CONV(1) \text{ obtain Ml' Me' h'} where RU: (entry fg q', Ml', Me', h') \in \text{RU-cs fg U Ml'} \subseteq \text{mon-loc fg w2' Me'} \subseteq \text{mon-env fg w2' h'} \leq \text{aah (map (\alpha nl fg) w2')} \text{ by auto} \)

moreover have \( \text{mon-loc fg w2' \subseteq mon-ww fg w2 mon-env fg w2' \subseteq mon-ww fg w2 using mon-ww-ileq(OF REVSPILT(2), of fg) R-CONV(2) by (auto simp add: mon-ww-of-le-rem) \}

moreover have \( \text{aah (map (\alpha nl fg) w2') \leq aah (map (\alpha nl fg) w2) using REVSPILT(2) R-CONV(2) by (auto simp add: \alpha-\alpha\|\text{symmetric} le-list-map map-map\|\text{symmetric} simp del: map-map intro: aah-ileq del: predicate2I) \}

ultimately show \( \text{thesis using that REVSPILT(1) by (blast intro: order-trans)} \)

qed

from \( \text{S-ENTRY-PAT|of \{\#\#\} | simplified} \) \( \text{RU(1)} \) \text{have S-ENTRY: (v, mon-w fg w, \{\#\}\{\#\}} \in \text{S-cs fg I by simp} \)

have \( (u, \text{mon fg p \cup mon-w fg w \cup Ml, Me \cup Ml' \cup Me'}) \in \text{RUV-cs fg U V} \)

proof (rule \( \text{RUV-split-c|OF FS-FMT(2,4) S-ENTRY - RV(1) RU(2)} \))

from \( \text{MON-PU MON1-LEQ MON2-LEQ RV(3) RU(3,4) show (mon-n fg u \cup mon fg p) \cap (Me \cup Ml' \cup Me') = \{\}} \text{ by blast} \)

next

from \( \text{ah-interleavable|OF LESPLIT(1)} \) \text{have aah (map (\alpha nl fg) w1) [\star aah (map (\alpha nl fg) w2)] by simp} \)

thus \( h [\star] h' \) using \( \text{RV(4) RU(5)} \) by (auto elim: ah-leq-il)

qed (simp)

moreover have \( \text{mon fg p \cup mon-w fg w \cup Ml \subseteq mon-loc fg \text{ ee \# \#}} \text{ using FS-FMT(1) MON1-LEQ RV(2) by (simp) blast} \)

moreover have \( \text{Me \cup Ml' \cup Me' \subseteq mon-env fg \text{ ee \# \#}} \text{ using MON1-LEQ MON2-LEQ RV(3) RU(3,4) by (simp) blast} \)

ultimately show \( \text{thesis by blast} \)

qed
lemma (in flowgraph) RUV-precise: \((u, M, Me) \in \text{RUV-cs} f g U V\)

\[
\rightarrow \exists w s' c'. \\
((\{u\}, \{\#\}), w, (s', c')) \in \text{trcl (ntrp} f g) \land \\
\text{atUV} U V (\{\#s'\#\} + c') \land \\
\text{mon-loc} f g w = M \land \\
\text{mon-env} f g w = Me \\
\]

proof (induct rule: RUV-cs.induct)

\begin{enumerate}
    \item \textbf{case (RUVCall} u p u' v M P Mi Me) then obtain \(ww s' c'\) where \(IH:\) \(((\{v\}, \{\#\}), ww, s', c') \in \text{trcl (ntrp} f g) \text{atUV} U V (\{\#s'\#\} + c') \text{mon-loc} f g ww = M \land \text{mon-env} f g \text{ww = Me} \text{by blast}

from \text{S-precise-ntrp [OF RUV-call(3,2,1), simplified]} obtain \(w c h\) where \(FS:\) \(((\{u\}, \{\#\}), \text{LOC (LCall p} \# \text{w}), [v, u'], \text{ch}) \in \text{ntrp} f g \text{P} = \{\#\} \text{M} = \text{mon-w} f g w \text{mon-n} f g v = \text{mon} f g p \text{mon-c} f g \text{ch} = \{\} \text{by blast}
\end{enumerate}

\begin{enumerate}
    \item \textbf{note} \(FS(1)\)
    \item also have \(((\{u\}, \{\#\}), \text{LOC (LCall p} \# \text{w}), \text{ww, s' @ [u'], c' + ch}) \in \text{trcl (ntrp} f g) \text{by simp}
    \item finally have \(((\{u\}, \{\#\}), \text{LOC (LCall p} \# \text{w}), \text{ww, s' @ [u'], c' + ch}) \in \text{trcl (ntrp} f g) \text{by auto}
    \item moreover from \text{IH(2)} have \text{atUV} U V (\{\#s' @ [u']\#\} + (c' + \text{ch})) \text{by auto}
    \item moreover have \text{mon-loc} f g (\text{LOC (LCall p} \# \text{w}), \text{ww}) \text{= mon} f g p \cup M \cup \text{Mi using \text{IH(4)} by auto}
\end{enumerate}

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

next

\begin{enumerate}
    \item \textbf{case (RUVC-Spawn} u p u' v M q Mi Me) then obtain \(ww s' c'\) where \(IH:\) \(((\{entry f g q\}, \{\#\}), \text{ww, s', c'}) \in \text{trcl (ntrp} f g) \text{atUV} U V (\{\#s'\#\} + c') \text{mon-loc} f g \text{ww = Mi mon-env} f g \text{ww = Me by blast}

from \text{S-precise-ntrp [OF RUV-Spawn(3,2,1), simplified]} \text{mset-size1elem [OF - RUV-Spawn(4)]}
\end{enumerate}

\text{obtain} \(w c h\) where

\begin{enumerate}
    \item \text{FS:\} \(((\{u\}, \{\#\}), \text{LOC (LCall p} \# \text{w}), [v, u'], \{\#(entry f g q)\#\} + \text{che}) \in \text{ntrp} f g \text{P} = \{\#q\#\} \text{M = mon-w} f g w \text{mon-n} f g v = \text{mon} f g p \text{mon-c} f g \text{((\#(entry f g q)\#) + \text{che}) = \{\} by (auto elim: mset-le-addE)\}
    \item moreover have \(((v, u'), \text{che} + \{\#(entry f g q)\#\}), \text{map ENV (map le-rem-s} ww\}, [v, u', \text{che + ((\#s'\#)} + c')) \in \text{trcl (ntrp} f g) \text{by (auto simp add: mon-c-unique mon-w-ww-of-le-rem)\}
    \item using \text{ntr2ntrp [OF \text{gtrp2gtr [OF IH(1)}], \text{of intrp} [v, u'] \text{che} \text{IH(3,4) RUV-Spawn(7)}\)
    \item FS(4,5) \text{mon-n-same-proc [OF edges-part [OF RUV-Spawn(1)]]\ by (auto simp add: mon-c-unique mon-w-ww-of-le-rem)\}
    \item ultimately have \(((\{u\}, \{\#\}), \text{LOC (LCall p} \# \text{w}) \text{map ENV (map le-rem-s} ww), [v, u', \text{che + ((\#s'\#) } + c')] \in \text{trcl (ntrp} f g)\text{ by (auto simp add: union-ac)\}
    \item moreover have \text{atUV} U V (\{\#[v, u']\#\} + (\text{che + ((\#s'\#)} + c')) \text{using \text{IH(2) by auto)}
\end{enumerate}

\text{by auto}

\begin{enumerate}
    \item moreover have \text{mon-loc} f g (\text{LOC (LCall p} \# \text{w}) \text{map ENV (map le-rem-s} ww)) \text{= mon} f g p \cup M \text{using \text{FS(3) by (simp del: map-map)}\}
    \item moreover have \text{mon-env} f g (\text{LOC (LCall p} \# \text{w}) \text{map ENV (map le-rem-s} ww))\}
\end{enumerate}
ultimately show \(?\)case by blast

next

\textbf{case (RU\textsubscript{V}-split-le \(u \ u' \ v \ M \ P \ q \ Ml \ M e \ h \ Ml' \ Me' \ h') — This is the symmetric case to RU\textsubscript{V}-split-le, it is proved completely analogously, just need to swap \(U\) and \(V\).}

— Get paths from precision results
from S-precise-ntrp[of RUV-split-el(3,2,1), simplified] mset-size1elem[of RUV-split-el(4)]

obtain w cheating where

FS: (((u, {}), LOC (LCall p # w), [v, w'), [#entry fg q]#) + cheat) ∈ ntrp
fg P=#{#q} # M = mon-w fg w mon-n fg v = mon fg p mon-e fg ((#entry fg q) #) + cheat) = {} by (auto elim: mset-le-addE)

from RU-precise[of RUV-split-el(5)] obtain w w1 s1' c1' where P1: (((v, {}),
ww1, s1', c1') ∈ trcl (ntrp fg) at U V (#{s1'#} + c1') mon-loc fg ww1 = Ml mon-env
fg ww1 = Me aah (map (cnl fg) w w1) = h by blast

from RU-precise[of RUV-split-el(6)] obtain w w2 s2' c2' where P2: (((entry fg q), {}),
ww2, s2', c2') ∈ trcl (ntrp fg) at U V (#{s2' #} + c2') mon-loc fg ww2 = Ml mon-env fg ww2 = Me aah (map (cnl fg) w w2) = h' by blast

— Get combined path from the acquisition history interleaveability, need to remap
loc/env-steps in second path

from P2(5) have aah (map (cnl fg) (map ENV (map le-rem-s w w1))) = h' by
(simp add: an-cnI o-assoc)

with P1(5) RUV-split-el(8) obtain w w where IL: w w ∈ w w1 ⊗ anl fg (map ENV
(map le-rem-s w w2)) using ah-interleaveable2 by (force)

— Use the ntrp-unsplit-theorem to combine the executions

from ntrp-unsplit[where ca=(#), OF IL P1(1) gtrp2grt[OF P2(1), simplified]
have (((v, {}), [#entry fg q]#), ww, s1', c1' + (#{s2#} + c2') ∈ trcl (ntrp fg)
using FS(4,5) RUV-split-el(7)
by (auto simp add: mon-c-uncon mon-ww-of-le-rem)

from ntrp-unsplit-context[OF ntrp-stack-comp[OF this, of [u'], of cheat] have (([v]
@ [u'], [#entry fg q]#) + cheat), ww, s1' @ [u'], c1' + (#{s2#} + c2') + cheat) ∈
trcl (ntrp fg)

using mon-n-nsame-prec[OF edges-part[of RUV-split-el(1)]] mon-loc-cil[of IL, of fg]
mon-ww[of IL, of fg] FS(4,5) RUV-split-el(7) by (auto simp add:
mon-c-uncon P1(3,4) P2(3,4) mon-ww-of-le-rem simp del: map-map)

with FS(1) have (((v, {}), LOC (LCall p # w) # ww, (s1' @ [u'], c1' +
(#{s2#} + c2') + cheat)) ∈ trcl (ntrp fg) by simp

moreover have atUV U V (#{s1' @ [u']#} + (c1' + (#{s2#} + c2') + cheat))
using P1(2) P2(2) by auto

moreover have mon-loc fg (LOC (LCall p # w) # ww) = mon fg p ∪ M ∪ Ml
using FS(3) P1(3) mon-loc-cil[of IL, of fg] by (auto simp del: map-map)

moreover have mon-ww fg (LOC (LCall p # w) # ww) = M ∪ Ml ∪ Ml
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-ce 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-ce(3,2,1), simplified] mset-size2elem[of RUV-split-ce(4)]

obtain w cheating where

FS: (((v, {}), LOC (LCall p # w), [v, w'], [#entry fg q]#) + [#entry fg q']#) + cheat) ∈ ntrp
fg P=#{#q} + (#{q'}#) M = mon-w fg w mon-n fg v = mon fg p mon-c fg ((#entry fg q) #) + cheat) = {]

by (auto elim: mset-le-addE)

from RU-precise[of RUV-split-ce(5)] obtain w w1 s1' c1' where P1: (((entry fg q), {}),
ww1, s1', c1') ∈ trcl (ntrp fg) at U V (#{s1' #} + c1') mon-loc fg ww1 =
The Isabelle prover must additionally trust, is the following:

At this point everything is available to prove the main result of this project:

```isar
begin
  ConstraintSystems
      imports theory MainResult
end
```

```isar
qed
```

— Get interleaved paths, project away loc/env information first

```isar
from P1(5) P2(5) have αah (map (αf) (map le-rem-s wv2)) = h' by (force simp del: map-map)
```

— Use the ntr-unsplit-theorem to combine the executions

```isar
from ntr-unsplit(OF IL gtrip2gtr(OF P1(1)) gtrip2gtr(OF P2(1)), simplified) have
PC: (\{#|entry fg q|#\} + \{#|entry fg q'|#\}, wv, \{#s1'|#\} + c1' + ((\{#s2'|#\} + c2')) \} \in trcl (ntr fg)
```

— Prepend first step

```isar
from ntr2ntrp(OF PC(1), of [v',u'] che) have ((\{v, u\}', che + (\{#|entry fg q|#\}
    + \{#|entry fg q'|#\}), map ENV wv, [v, u'], che + (\{#s1'|#\} + c1' + ((\{#s2'|#\} + c2')) \} \in trcl (ntrp fg)
```

```isar
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-af-le-rem P1(3,4) P2(3,4))
```

```isar
with FS(1) have (((u, #), LOC (LCall p # w) # map ENV wv, [v, u'], che + (\{#s1'|#\} + c1' + ((\{#s2'|#\} + c2')) \}) \in trcl (ntrp fg) by (auto simp add: union-ac)
```

```isar
moreover have atUV U V (\{#v, u'#\} + (che + (\{#s1'|#\} + c1' + ((\{#s2'|#\} + c2')))) using P1(2) P2(2) by auto
```

```isar
moreover have mon-loc fg (LOC (LCall p # w) # map ENV wv) = mon fg p \cup M using FS(3) by auto
```

```isar
moreover have mon-env fg (LOC (LCall p # w) # map ENV wv) = Ml \cup Me \cup Ml' \cup Me' using mon-ww-cil(OF IL, of fg) by (auto simp add: P1(3,4) P2(3,4) mon-ww-af-le-rem)
```

ultimately show ?case by blast

qed

end

10 Main Result

theory MainResult
imports ConstraintSystems
begin

At this point everything is available to prove the main result of this project:

*The constraint system RUV-cs precisely characterizes simultaneously reachable control nodes w.r.t. to our semantic reference point.*

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

- The flowgraph and the assumptions made on it in the flowgraph-
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:**

\[
(\exists w c'. \{ \# [entry fg (main fg)] \# \}, w, c') \in \text{trcl} (\text{trcl} fg) \land \text{atUV} U V c' \\
\iff (\exists Ml Me, (entry fg (main fg), Ml, Me) \in \text{RUW-cs} fg U V)
\]

— The proof uses the soundness and precision theorems wrt. to normalized paths (eflowgraph.RUV-sound, eflowgraph.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 (eflowgraph.refpoint-eq).

(is ?lhs \iff \?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

  t: c'=add-mset s' ce' ww = map le-rem-s ww (entry fg (main fg)), \#), ww, s', ce' \in trcl (ntrp fg) by blast

with C(2) have \#: atUV U V \{\# s'\}+ce' \by auto

from RUV-sound[OF C(3) 2] show ?rhs by blast

next

assume ?rhs

then obtain Ml Me where C: (entry fg (main fg), Ml, Me) \in \text{RUW-cs} fg U V by blast

  from \text{RUW-precise}[OF C] obtain ww s' c' where P: \{\# [entry fg (main fg)], \#\}, ww, s', c' \in trcl (ntrp fg) atUV U V \{\# s'\}+c' \by blast

  from gtrp2gtr[OF P(1)] have \{\# [entry fg (main fg)] \#\}, map le-rem-s ww, \{\# s'\}+c' \in trcl (ntrp fg) by (auto)

  from ntr-is-tr[OF this| P(2) have \exists w c'. \{\# [entry fg (main fg)] \#\}, w, c' \in trcl (tr fg) \land \text{atUV} U V c' by blast

  thus ?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


checking threads for LTL properties. In Proc. of LICS 2006, pages

[5] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads com-

[6] P. Lammich and M. Müller-Olm. Precise fixpoint-based analysis of pro-