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

August 28, 2014

Abstract
In this work we formally verify the soundness and precision of a
static program analysis that detects conflicts (e.g. data races) in pro-
grams 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, synchro-
nization by monitors, and thread creation. The analysis is based on
the observation that all conflicts already occur in a class of partic-
ularly 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 [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
\[
\text{mon-pl} \, w = \text{foldl} \, (\text{op} \, \cup) \, {} \, (\text{map} \, (\lambda e. \, \text{fst} \, e \cup \text{snd} \, e) \, w)
\]

lemma \( \text{mon-pl-empty} \, [] = {} \)
by (unfold mon-pl-def, auto)

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

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

** lemma mon-pl-ileq: w ⪯ w′ =⇒ mon-pl w ⊆ mon-pl w′
by (induct rule: less-eq-list-induct) auto

** lemma mon-pl-set:
mon-pl w = ∪ { fst e ∪ snd e | e ∈ set w }
by (unfold mon-pl-def) (safe, auto simp add: Bex-def foldl-set)

** fun

cil :: 'a list ⇒ ('a ⇒ ('m set × 'm set)) ⇒ 'a list set
(- ⊗ - [64,64,64] 64)
where
 — Interleaving with the empty word results in the empty word
 [] ⊗α w = {w}
| w ⊗α [] = {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 ⊗α e2#w2 = (  
  if fst (α e1) ∩ mon-pl (map α (e2#w2)) = {} then    
  e1·(w1 ⊗α e2#w2)  
  else {}  
) ∪ (  
  if fst (α e2) ∩ mon-pl (map α (e1#w1)) = {} then    
  e2·(e1#w1 ⊗α w2)  
  else {}  
)

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

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

** lemma cil-last-case-split[cases set, case-names left right]:
[ | w∈e1#w1 ⊗α e2#w2;  
  !!w', [w=e1#w' ; w'∈(w1 ⊗α e2#w2);  
  fst (α e1) ∩ mon-pl (map α (e2#w2)) = {} ] =⇒ P;  
  !!w', [w=e2#w' ; w'∈(e1#w1 ⊗α w2);  
  fst (α e2) ∩ mon-pl (map α (e1#w1)) = {} ] =⇒ P  
] =⇒ P
by (auto elim: list-set-cons-cases split: split-if-asm)

** lemma cil-cases[cases set, case-names both-empty left-empty right-empty app-left
app-right]:
\[ w \in wa \otimes \alpha wb; \]
\[ [w = ]; [wa = ]; [wb = ] ] \implies P; \]
\[ [wa = ]; [w = wb] \implies P; \]
\[ [w = wa; wb = ] ] \implies P; \]
\[ \not \exists ea \ wa' \ w'. [w = ea \# w'; wa = ea \# wa'; w' \in wa' \otimes \alpha wb; \]
\[ \text{fst} (\alpha \ ea) \cap \text{mon-pl} (\text{map} \ \alpha \ wb) = \{\} ] \implies P; \]
\[ \not \exists eb \ wb' \ w'. [w = eb \# w'; wb = eb \# wb'; w' \in wa \otimes \alpha wb'; \]
\[ \text{fst} (\alpha \ eb) \cap \text{mon-pl} (\text{map} \ \alpha \ wa) = \{\} ] \implies P \]
\[ \implies P \]

**proof** \((\text{induct} wa \ \alpha \ wb \ \text{rule:cil.induct})\)**

**case 1** **thus** ?case by simp next

**case 2** **thus** ?case by simp next

from 3.prems(1) **show** ?thesis **proof** (cases rule: cil-last-case-split)

**case (left w')** from 3.prems(5)[OF left(1) - left(2,3)] **show** ?thesis by simp

**next**

**case (right w')** from 3.prems(6)[OF right(1) - right(2,3)] **show** ?thesis by simp

**qed**

**qed**

**lemma cil-induct'[case-names both-empty left-empty right-empty append]:**

\[ \\land \alpha. \ P \ \alpha \ [] ]; \]
\[ \\land \alpha \ ad \ ae. \ P \ \alpha \ [] (ad \ # \ ae); \]
\[ \\land \alpha \ z \ aa. \ P \ \alpha \ (z \ # \ aa) ]; \]
\[ \\land \alpha \ e1 \ w1 \ e2 \ w2. \ [\]
\[ \text{fst} (\alpha \ e1) \cap \text{mon-pl} (\text{map} \ \alpha \ (e2 \ # \ w2)) = \{\} ] \implies P \ \alpha \ w1 (e2 \ # \ w2); \]
\[ \text{fst} (\alpha \ e2) \cap \text{mon-pl} (\text{map} \ \alpha \ (e1 \ # \ w1)) = \{\} ] \implies P \ \alpha (e1 \ # \ w1) w2 \]
\[ \implies P \ \alpha (e1 \ # \ w1) (e2 \ # \ w2) \]
\[ \implies P \ \alpha wa wb \]

**apply** (induct wa \ \alpha \ wb \ \text{rule: cil.induct})

**apply** (case-tac w)

**apply** auto

**done**

**lemma cil-induct-fixa:**

\[ P \ \alpha \ [] ]; \]
\[ \\land \alpha \ ad \ ae. \ P \ \alpha \ [] (ad \ # \ ae); \]
\[ \\land \alpha \ z \ aa. \ P \ \alpha \ (z \ # \ aa) ]; \]
\[ \\land \alpha \ e1 \ w1 \ e2 \ w2. \ [\]
\[ \text{fst} (\alpha \ e2) \cap \text{mon-pl} (\text{map} \ \alpha \ (e1 \ # \ w1)) = \{\} ] \implies P \ \alpha (e1 \ # \ w1) w2; \]
\[ \text{fst} (\alpha \ e1) \cap \text{mon-pl} (\text{map} \ \alpha \ (e2 \ # \ w2)) = \{\} ] \implies P \ \alpha w1 (e2 \ # \ w2); \]
\[ \implies P \ \alpha (e1 \ # \ w1) (e2 \ # \ w2) \]
\[ \implies P \ \alpha v \ w \]

**apply** (induct v \ \alpha \ w \ \text{rule: cil.induct})

**apply** (case-tac w)

**apply** auto

**done**

7
lemma cil-induct-fixα′[case-names both-empty left-empty right-empty append]:
\[ P \alpha [\ []]; \]
\[ \wedge \alpha. P \alpha [] (\alpha \# \alpha); \]
\[ \wedge z \alpha. P \alpha (z \# \alpha)]; \]
\[ \wedge e1 w1 e2 w2. \]
\[ \text{fst} (\alpha e1) \cap \text{mon-pl} (\text{map} \alpha (e2 \# w2)) = \{\} \Rightarrow P \alpha w1 (e2 \# w2); \]
\[ \text{fst} (\alpha e2) \cap \text{mon-pl} (\text{map} \alpha (e1 \# w1)) = \{\} \Rightarrow P \alpha (e1 \# w1) w2 \]
\[ \Rightarrow P \alpha (e1 \# w1) (e2 \# w2) \]
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]: \[ ([] \in w a \otimes_\alpha wb) = (wa=[] \land wb=[]) \]
by (induct wa α wb rule: cil.induct) auto

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

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

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

lemma cil-cons1: \[ [w \in wa \otimes_\alpha wb; \text{fst} (\alpha e) \cap \text{mon-pl} (\text{map} \alpha wb) = \{\}] \]
\[ \Rightarrow e \# w \in e \# wa \otimes_\alpha wb \]
by (cases wb) auto
lemma cil-cons2: \[ \{ w \in wa \otimes_\alpha wb \mid \text{fst} (\alpha e) \cap \text{mon-pl} (\text{map} \alpha wa) = \{ \} \} \]
by (cases wa) auto

2.2 Properties of consistent interleaving

— Consistent interleaving is a restriction of interleaving
lemma cil-subset-il: \[ w \otimes_\alpha w' \subseteq w \otimes w' \]
apply (induct w \alpha w' rule: cil.induct)
apply simp-all
apply safe
apply auto
done

lemma cil-subset-il': \[ w \in w_1 \otimes w_2 \Longrightarrow w \in w_1 \otimes w_2 \]
using cil-subset-il by (auto)

— Consistent interleaving preserves the set of letters of both operands
lemma cil-set: \[ w \in w_1 \otimes w_2 \Longrightarrow \text{set} w = \text{set} w_1 \cup \text{set} w_2 \]
by (induct rule: cil-set-induct-fix \alpha)
avto

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

— Consistent interleaving contains all letters of each operand in the original order
lemma cil-ileq: \[ w \in w_1 \otimes w_2 \Longrightarrow w_1 \preceq w \land w_2 \preceq w \]
by (intro conjI cil-subset-il' ileq-interleave)

— Consistent interleaving is commutative and associative
lemma cil-commute: \[ w \otimes_\alpha w' = w' \otimes_\alpha w \]
by (induct rule: cil.induct) auto

lemma cil-assoc1: \exists w\_l w\_1 w\_2 w\_3. \[ w \in w_1 \otimes w_2 \otimes w_3 \]
proof (induct w rule: length-compl-induct)
case Nil thus \&case by auto
next
case (Cons e w) from Cons.prems(1) show \&case proof (cases rule: cil-cons-cases)
case (left w\_l') with Cons.prems(2) have e\# w\_l' \in w_1 \otimes w_2 by simp
thus \&thesis proof (cases rule: cil-cons-cases[case-names left' right'])
case (left' w\_l')
from Cons.hyps[OF - left(2) left'(2)] obtain wr where IHAPP: \[ w \in w_1' \]
\otimes_\alpha wr \in w_2 \otimes_\alpha w_3 by blast
have e\# w\_e \in w_1' \otimes_\alpha wr proof (rule cil-cons[OF IHAPP(1)])
from left left' cil-mon-pl[OF IHAPP(2)] show \text{fst} (\alpha e) \cap \text{mon-pl} (\text{map} \alpha
\( w_r \) = \{ \} \text{ by auto}

\textbf{qed}

\begin{itemize}
\item \textbf{thus } \textbf{thesis using } \textbf{IHAPP(2) left' by blast}
\item \textbf{next}
\item \textbf{case (right' w2')} from \textbf{Cons.hyps(OF - left(2) right'(2)) obtain wr where}
\item \textbf{IHAPP}: \( w \in w_1 \otimes_\alpha w_r w_r \in w_2' \otimes_\alpha w_3 \text{ by blast}
\item from \textbf{IHAPP(2) left have e#w_r \in e#w_2' \otimes_\alpha w_3 \text{ by (auto intro: cil-cons1)}
\item moreover from \textbf{right' IHAPP(1) have e#w \in w_1 \otimes_\alpha e#wr \text{ by (auto intro: cil-cons2)}
\item ultimately show \textbf{thesis using } \textbf{right' by blast}
\item \textbf{qed}
\item \textbf{qed}
\end{itemize}

\textbf{lemma } \textbf{cil-assoc2:}
\begin{itemize}
\item \textbf{assumes A: } \( w \in w_1 \otimes_\alpha w_r w_r \in w_2 \otimes_\alpha w_3 \)
\item \textbf{shows } \exists \ w_l. \ w \in w_1 \otimes_\alpha w_3 \wedge w_l \in w_1 \otimes_\alpha w_2
\item \textbf{proof –}
\item from \textbf{A have A': } \( w \in w_r \otimes_\alpha w_1 \text{ by (simp add: cil-commute)}
\item from \textbf{B have B': } \( w_r \in w_2 \otimes_\alpha w_2 \text{ by (simp add: cil-commute)}
\item from \textbf{cil-assoc1(OF A' B')} \textbf{obtain wl where } \( w \in w_3 \otimes_\alpha w_l \wedge w_l \in w_2 \otimes_\alpha w_1 \)
\item \textbf{by blast}
\item \textbf{thus } \textbf{thesis} \textbf{ by (auto simp add: cil-commute)}
\item \textbf{qed}
\end{itemize}

— Parts of the abstraction can be moved to the operands

\textbf{lemma } \textbf{cil-map: } w \in w_1 \otimes_{(\alpha f)} w_2 \mapsto \text{ map } f w \in \text{ map } f w_1 \otimes_\alpha \text{ map } f w_2

\textbf{proof (induct rule: cil-set-induct-fiza)}
\begin{itemize}
\item \textbf{case empty thus } \textbf{?case by auto}
\item \textbf{next}
\item \textbf{case (left e w' w1' w2)}
\item have \( f e \# \text{ map } f w' \in f e \# \text{ map } f w_1' \otimes_\alpha \text{ map } f w_2 \text{ proof (rule cil-cons1)}
\item from \textbf{left(2) have } \textbf{fst } ((\alpha f) e) \cap \text{ mon-pl } (\text{ map } \alpha (\text{ map } f w_2)) = \{ \} \text{ by (simp only: map-map[symmetric])}
\item thus \textbf{fst } (\alpha (f e)) \cap \text{ mon-pl } (\text{ map } \alpha (\text{ map } f w_2)) = \{ \} \text{ by (simp only: o-apply)}
\item \textbf{qed (rule left(3))}
\item \textbf{thus } \textbf{?case by simp}
\item \textbf{next}
\item \textbf{case (right e w' w2' w1)}
have \( f e \# \text{map} f w1 \otimes \alpha \) \( f e \# \text{map} f w2 \) proof (rule cil-cons2)
from right(2) have \( \text{fst} ((\alpha \circ f) e) \cap \text{mon-pl} (\text{map} \alpha (\text{map} f w1)) = \{\} \) by (simp only: map-map[symmetric])
thus \( \text{fst} (\alpha (f e)) \cap \text{mon-pl} (\text{map} \alpha (\text{map} f w1)) = \{\} \) by (simp only: o-apply)
qed (rule right(3))
thus ?case by simp
qed

end

3 Acquisition Histories

theory AcquisitionHistory
imports ConsInterleave
begin

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

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

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

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

### 3.1 Definitions

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

**Definition**

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

**Lemma**

\[ ah-cases[\text{cases set}]: \left[ h \in ah; h m = \{ \} \Rightarrow P ; m \in h m \Rightarrow P \right] \Rightarrow P \]

**Proof**

\[ \text{by (unfold ah-def) blast} \]

### 3.2 Interleavability

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

**Definition**

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

**Where**

\[ h1 \[\star\] h2 == \neg(\exists m1 m2. m1 \in h2 m2 \land m2 \in h1 m1) \]

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

**Lemma**

\[ ah-il-lockset-disjoint: \left[ h1 \in ah; h2 \in ah; h1 \[\star\] h2 \right] \Rightarrow h1 m = \{\} \lor h2 m = \{\} \]

**Proof**

\[ \text{by (unfold ah-il-def) (auto elim: ah-cases)} \]

Of course, acquisition history interleavability is commutative.

**Lemma**

\[ ah-il-commute: h1 \[\star\] h2 \Rightarrow h2 \[\star\] h1 \]

**Proof**

\[ \text{by (unfold ah-il-def) auto} \]

### 3.3 Used monitors

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

**Definition**

\[ mon-ah : ('m \Rightarrow 'm set) \Rightarrow 'm set \]

**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.

**Lemma** `ah-leq-il`:
\[
\begin{align*}
& \{ h1 \ast h2 ; h1' \leq h1 ; h2' \leq h2 \} \implies h1' \ast h2' \\
& \text{by (unfold ah-il-def le-fun-def [where 'b='a set]) blast+}
\end{align*}
\]

**Lemma** `ah-leq-il-left`:
\[
\begin{align*}
&\{ h1 \ast h2 ; h1' \leq h1 \} \implies h1' \ast h2 \\
&\text{by (unfold ah-il-def le-fun-def [where 'b='a set]) blast+}
\end{align*}
\]

3.5 Acquisition histories of executions

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

**Primrec** `\alpha ah :: ('m set \times 'm set) list \Rightarrow 'm \Rightarrow 'm set where``
\[
\begin{align*}
& \alpha ah \emptyset m = \{ \} \\
& \alpha ah \ (e\#w) m = (if m\in\text{fst } e \text{ then } \text{fst } e \cup \text{snd } e \cup \text{mon-pl } w \text{ else } \alpha ah w m)
\end{align*}
\]

— \(\alpha ah\) generates valid acquisition histories

**Lemma** `\alpha ah-ah`:
\[
\begin{align*}
& \alpha ah \in \text{ah} \\
& \text{apply (induct w)} \\
& \text{apply (unfold ah-def)} \\
& \text{apply simp} \\
& \text{done}
\end{align*}
\]

**Lemma** `\alpha ah-hd`:
\[
\begin{align*}
& [m\in\text{fst } e ; x\in\text{fst } e \cup \text{snd } e \cup \text{mon-pl } w] \implies x\in\alpha ah \ (e\#w) m \\
& \text{by auto}
\end{align*}
\]

**Lemma** `\alpha ah-tl`:
\[
\begin{align*}
& [m\notin\text{fst } e ; x\in\alpha ah w m] \implies x\in\alpha ah \ (e\#w) m \\
& \text{by auto}
\end{align*}
\]

**Lemma** `\alpha ah-cases`:
\[
\begin{align*}
& x\in\alpha ah w m; \\
& ![e w'. [w=e\#w'; m\in\text{fst } e ; x\in\text{fst } e \cup \text{snd } e \cup \text{mon-pl } w'] \implies P; \\
& ![e w'. [w=e\#w'; m\notin\text{fst } e ; x\in\alpha ah w' m] \implies P \\
& ] \implies P \\
& \text{by (cases w) (simp-all split: split-if-asm)}
\end{align*}
\]

**Lemma** `\alpha ah-cons-cases`:
\[
\begin{align*}
& x\in\alpha ah \ (e\#w') m; \\
& [m\in\text{fst } e ; x\in\text{fst } e \cup \text{snd } e \cup \text{mon-pl } w'] \implies P; \\
& [m\notin\text{fst } e ; x\in\alpha ah w' m] \implies P \\
& \text{by (cases w) (simp-all split: split-if-asm)}
\end{align*}
\]

13
\[ \rightarrow P \]
by \((\text{simp-all split: split-if-asm})\)

**lemma** mon-ah-subset: \(\text{mon-ah} (\alpha \text{ah} w) \subseteq \text{mon-pl} w\)
by \((\text{induct } w)\) \((\text{auto simp add: mon-ah-def})\)

— Subwords generate smaller acquisition histories

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

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

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

\textbf{proof} (rule ccontr) — We do a proof by contradiction

— Assume there is a conflicting pair in the acquisition histories

\textbf{assume} \( \neg \text{ah} (\text{map} \ \alpha (e \# w'')) \) \( \land \) \( \text{ah} (\text{map} \ \alpha w2) \)

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

— It comes either from the first step or not

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

\textbf{by} \ (\text{auto split: split-if-asm})

\textbf{moreover} \{

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

\textbf{assume} ?CASE1 \textit{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

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

— But this is a contradiction to being a conflicting pair

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

\textbf{moreover} \{

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

\textbf{assume} ?CASE2 \textit{hence} \( C: m1 \in \text{ah} (\text{map} \ \alpha w1') \ m2 \) ..

— 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

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

\textbf{ultimately show} False ..

\textbf{qed}

\textbf{next}

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

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

\textbf{proof} (rule ccontr)

\textbf{assume} \( \neg \text{ah} (\text{map} \ \alpha w1) \) \( \lor \) \( \text{ah} (\text{map} \ \alpha (e\#w2')) \)

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

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

\textbf{by} \ (\text{auto split: split-if-asm})

\textbf{moreover} \{

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

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

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

\textbf{moreover} \{

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

qed

lemma ah-interleavable2:
  assumes A: αah (map α w1) [+] αah (map α w2)
  shows w1 ⊗₁ w2 ≠ {}
  — This lemma is shown by induction on the sum of the word lengths
proof –
  — To apply this induction in Isabelle, we have to rewrite the lemma a bit
  { fix n
    have !!w1 w2. [ [ αah (map α w1) [+] αah (map α w2); n=length w1 + length w2] \implies w1 ⊗₁ w2 ≠ {} ]
      proof (induct n rule: nat-less-induct [case-names I])
        — We first rule out the cases that one of the words is empty
        case (I n w1 w2) show ?thesis proof (cases w1)
          — If the first word is empty, the lemma is trivial
          case Nil with I.prems show ?thesis by simp
        next
          — The interesting case is if both words are not empty
          case (Cons e1 w1') note CONS1=this show ?thesis proof (cases w2)
            — If the second word is empty, the lemma is also trivial
            case Nil with I.prems show ?thesis by simp
          next
            — The interesting case is if both words are not empty
            case (Cons e2 w2') note CONS2=this
            — In this case, we check whether the first step of one of the words can safely be executed without blocking any steps of the other word
            show ?thesis proof (cases fst (α e1) \inter mon-pl (map α w2) = {})
              case True — The first step of the first word can safely be executed
              — From the induction hypothesis, we get that there is a consistent interleaving of the rest of the first word and the second word
              have w1' ⊗₁ w2 ≠ {} proof –
                from I.prems(1) CONS1 ah-leq-il-left[OF - αah-ileq[OF le-list-map, OF less-eq-list-drop[OF order-refl]]] have αah (map α w1') [+] αah (map α w2) by fast
                moreover from CONS1 I.prems(2) have length w1' + length w2 < n by simp
                ultimately show ?thesis using I.hyps by blast
              qed
              — And because the first step of the first word can safely be executed, we can prepend it to that consistent interleaving
              with cil-cons1[OF - True] CONS1 show ?thesis by blast
            next
              case False note C1=this
              show ?thesis proof (cases fst (α e2) \inter mon-pl (map α w1) = {})
have \( w_1 \otimes_{\alpha} w_2' \neq \{ \} \) proof

from \( I \text{.prems}(1) \) CONS2 ah-leq-il-right\([\text{OF - } \alpha \text{ ah-ileq}\{\text{OF le-list-map, OF less-eq-list-drop}\} \] have \( \alpha \text{ah} (\text{map } \alpha \ w_1) \] \[ \ast \] \( \alpha \text{ah} (\text{map } \alpha \ w_2') \)

by fast

moreover from CONS2 \( I \text{.prems}(2) \) have length \( w_1 + \)length \( w_2' < \) \( n \) by simp

ultimately show \( \text{thesis} \) using \( I \text{.hyps} \) 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 \( m_1 \ m_2 \) where \( m_1 \in \text{fst } (\alpha \ e_1) \ m_1 \in \text{mon-pl} \ (\text{map } \alpha \ w_2) \ m_2 \in \text{fst } (\alpha \ e_2) \ m_2 \in \text{mon-pl} \ (\text{map } \alpha \ w_1) \) by blast

with CONS1 CONS2 have \( m_2 \in \alpha \text{ah} (\text{map } \alpha \ w_1) \ m_1 \ m_1 \in \alpha \text{ah} (\text{map } \alpha \ w_2) \) m_2 by auto

— But by assumption, there are no conflicting pairs, thus we get a contradiction

with \( I \text{.prems}(1) \) have \( \text{False} \) by \( (\text{unfold ah-il-def}) \) blast

thus \( \text{thesis} \) ..

qed

qed

qed

qed

qed

} with \( A \) show \( \text{thesis} \) by blast

qed

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

theorem ah-interleaveable:

\((\alpha \text{ah} (\text{map } \alpha \ w_1) \] \[ \ast \] \( \alpha \text{ah} (\text{map } \alpha \ w_2)) \leftrightarrow (w_1 \otimes_{\alpha} w_2' \neq \{ \})\)

using ah-interleaveable1 ah-interleaveable2 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 :: \( ('m \Rightarrow 'm \ \text{set}) \Rightarrow ('m \ \text{set} \ast 'm \ \text{set}) \Rightarrow 'm \ \text{set} \Rightarrow ('m \Rightarrow 'm \ \text{set}) \)

where

ah-update \( h \ F \ M \ m \) \( == \) if \( m \in \text{fst } F \) then \( \text{fst } F \cup \text{snd } F \cup M \) else \( h \ m \)

Intuitively, \( \text{ah-update } h \ (E, U) \ M \ m \) 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
about the monitors that are used on a path before the first monitor that will not be left has been entered.

lemma ah-update-cons: α ah (e#w) = ah-update (α ah w) e (mon-pl w)
   by (auto intro!: ext simp add: ah-update-def)

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

lemma ah-update-mono: \[ h \leq h'; F=F'; M\subseteq M' \] \[ \Rightarrow \] ah-update h F M ≤ 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 ≤ 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 (c,a) LTS = (c × a × c) set

Transitive reflexive closure

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

4.2 Basic properties of transitive reflexive closure

lemma trcl-empty-cons: (c,[]c')∈trcl t \[ \Rightarrow \] (c=c')
   by (auto elim: trcl.cases)

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

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

lemma trcl-uncons-cases: 
[(c, a \# w, c') \in \text{trcl } t;
  \forall ch. [(c, e, ch) \in S; (ch, w, c') \in \text{trcl } S] \implies P
]\implies P
by (blast dest: trcl-uncons)

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

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

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

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

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

**Lemma trcl-rev-cons**: \[ (c, w, ch) ∈ trcl T; (ch, e, c') ∈ T \] \[⇒ (c, w[e], c') ∈ trcl T \]
by (auto dest: trcl-concat iff add: trcl-single)

**Lemma trcl-rev-uncons**: \[ (c, w[e], c') ∈ trcl T \]
\[⇒ ∃ ch. (c, w, ch) ∈ trcl T ∧ (ch, e, c') ∈ T \]
by (force dest: trcl-unconcat)

**Lemma trcl-rev-induct**: \[(\text{induct set}, \text{consumes 1}, \text{case-names empty snoc})\]: \[! \]
\[c'. \]
\[\]
\[
(c, w, ch) ∈ trcl T; (ch, e, c') ∈ T \]
\[⇒ (c, w[e], c') ∈ trcl T \]
by (induct w rule: rev-induct) (auto dest: trcl-rev-uncons)

**Lemma trcl-cons2**: \[\]
\[\]
\[
(c, e, ch) ∈ T; (ch, f, c') ∈ T \]
\[⇒ (c, [e, f], c') ∈ trcl T \]
by auto

4.2.2 Transitivity reasoning setup

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

**Declare cons[trans]**

**Declare trcl-concat[trans]**

**Declare trcl-rev-cons[trans]**

4.2.3 Monotonicity

**Lemma trcl-mono**: \[! A \subseteq B \]
\[⇒ trcl A \subseteq trcl B \]
apply (clarsimp)
apply (erule trcl.induct)
apply auto
done

**Lemma trcl-inter-mono**: \[x ∈ trcl (S ∩ R) \]
\[⇒ x ∈ trcl (S ∩ R) \]
\[⇒ x ∈ trcl R \]
proof
assume \[x ∈ trcl (S ∩ R)\]
with trcl-mono[of S ∩ R S] show \[x ∈ trcl S\] by auto
next
assume \[x ∈ trcl (S ∩ R)\]
with trcl-mono[of S ∩ R R] show \[x ∈ trcl R\] by auto
qed
4.2.4 Special lemmas for reasoning about states that are pairs

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

4.2.5 Invariants

lemma trcl-prop-trans[cases set, consumes 1, case-names empty steps]: 
\[ \forall (c,w,c') \in \text{trcl } S ; \\
( c = c' ; w = [] ) \Rightarrow P ; \\
( c \in \text{Domain } S ; c' \in \text{Range } ( \text{Range } S ) ) \Rightarrow P \]
\[ \Rightarrow P \]
apply (erule-tac trcl-rev-cases)
apply auto
apply (erule trcl.cases)
apply auto
done

end

5 Thread Tracking

theory ThreadTracking
imports Main ~~/src/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 == \{ (\#s\#)+c,e,\#s'\#)+c' | s c e s' c' . ((s,c),e,(s',c')\in gtrs \}

lemma gtrl-s: ((s,c),e,(s',c')\in gtrs \Rightarrow (\#s\#)+c,e,\#s'\#)+c'\in gtr gtrs
by (unfold gtr-def, auto)

lemma gtrl: ((s,c),w,(s',c')\in trcl gtrs
\Rightarrow (\#s\#)+c,w,\#s'\#)+c'\in trcl (gtr gtrs)
by (induct rule: trcl-pair-induct) (auto dest: gtrl-s)
lemma \textit{gtrE: [ ]}
\( (c,e,c') \in \text{gtr} \ T; \)
\( \forall !s \ ce' s' ce'! \ [ c=\{\#s\#\}+ce; c'=\{\#s'\#\}+ce'; ((s,ce),e,(s',ce')) \in T ] \implies P \)
\[ \implies P \]
\( \text{by (unfold gtr-def) blast} \)

lemma \textit{gtr-empty-conf1[simp]: [ ]}
\( (\{\#\},w,e') \in \text{gtr} \ S \)
\( (c,w,\{\#\}) \in \text{gtr} \ S \)
\( \text{by (auto elim: gtrE)} \)

lemma \textit{gtr-empty-conf2[simp]: [ ]}
\( (\{\#\},c'w,e') \in \text{trcl} (\text{gtr} \ S) \) \iff \( (w=\[] \land e'=\{\#\}) \)
\( \text{by (induct w) (auto dest: trcl-uncons)} \)

lemma \textit{gtr-find-thread: [ ]}
\( (c,e,c') \in \text{gtr} \ gtrs; \)
\( \forall !s \ ce' s' ce'! \ [ c=\{\#s\#\}+ce; c'=\{\#s'\#\}+ce'; ((s,ce),e,(s',ce')) \in gtrs] \implies P \)
\[ \implies P \]
\( \text{by (unfold gtr-def) auto} \)

lemma \textit{gtr-step-cases[cases set, case-names loc other]: [ ]}
\( (\{\#s\#\}+ce,e,c') \in \text{gtr} \ gtrs; \)
\( \forall s' ce'! \ [ c'=\{\#s'\#\}+ce'; ((s,ce),e,(s',ce')) \in gtrs ] \implies P; \)
\( \forall cc ss ss' ce'! \ [ ce=\{\#ss\#\}+cc; c'=\{\#ss'\#\}+ce'; \)
\( (ss,\{\#s\#\}+cc),e,(ss',ce')) \in gtrs ] \implies P \)
\[ \implies P \]
\( \text{by (auto elim!: gtr-find-thread mset-single-cases)} \)

lemma \textit{gtr-rev-cases[cases set, case-names loc other]: [ ]}
\( (c,e,c') \in \text{gtr} \ gtrs; \)
\( \forall !s \ ce' s' ce'! \ [ c=\{\#s\#\}+ce; (s,ce),e,(s',ce') \in gtrs ] \implies P; \)
\( \forall cc ss ss' ce'! \ [ c=\{\#ss\#\}+cc; ce'=\{\#ss'\#\}+cc; \)
\( (ss,ce),e,(ss',\{\#s'\#\}+cc)) \in gtrs ] \implies P \)
\[ \implies P \]
\( \text{by (auto elim!: gtr-find-thread mset-single-cases)} \)

\section*{5.2 Invariants}

lemma \textit{gtr-preserve-s: [ ]}
\( (c,e,c') \in \text{gtr} \ T; \)
\( P \ c; \)
\( \forall !s \ ce' s' ce'! \ [ P (\{\#s\#\}+c); ((s,ce),e,(s',ce')) \in T ] \implies P (\{\#s'\#\}+c') \)
\[ \implies P c' \]
\( \text{by (unfold gtr-def) blast} \)

lemma \textit{gtr-preserve: [ ]}
\( (c,w,e) \in \text{trcl} (\text{gtr} \ T); \)
\( P \ c; \)
5.3 Context preservation assumption

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

locale env-no-step =
fixes gtrs :: (('s s multiset),'l) LTS
assumes env-no-step-s [cases set, case-names csp]:
[(((s,c),e,(s',c'))∈gtrs; !] csp. c' = csp+c =⇒ P ] ] =⇒ 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]: [ ]
((s,c),w,(s',c'))∈trcl gtrs;
!! csp. c' = csp+c =⇒ P ] ] =⇒ P

proof —

have ((s,c),w,(s',c'))∈trcl gtrs =⇒ ∃ csp. c' = csp+c proof (induct rule: trcl-pair-induct)

case empty thus ?case by (auto intro: exI[of - {#}])

next

case (cons s c e sh ch w s' c') note IHP=this
from env-no-step-s[OF IHP(1)] obtain csph where ch = csph+c by auto
moreover from IHP(3) obtain csp' where c' = csp'+ch by auto
ultimately have c' = csp'+csph+c by (simp add: union-assoc)
thus ?case by blast

qed

moreover assume ((s,c),w,(s',c'))∈trcl gtrs !! csp. c' = csp+c =⇒ P
ultimately show ?thesis by blast

qed

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

loc The thread made the step
spawn The thread was spawned by the step
env The thread was not involved in the step
lemma (in env-no-step) rev-cases-p[cases set, case-names loc spawn env]:

assumes STEP: (c,e,\{#s'\#\}+ce')\in gtrs gtrs and
LOC: !!s ce. \[ c=\{#s\#\}+ce; ((s,ce),e,(s',ce'))\in gtrs \] \implies P and
SPAWN: !!ss ss' ce csp.
\[
\begin{align*}
& ((c,ss') + ce; ce' = \{#ss'\#\} + csp + ce; \\
& ((ss,ce),(ss',\{#s'\#\} + csp + ce)) \in gtrs \\
& \implies P \text{ and}
\end{align*}
\]

ENV: !!ss ss' ce csp.
\[
\begin{align*}
& ((c,ss') +\{#s'\#\} + ce; ce' = \{#ss'\#\} + csp + ce; \\
& ((ss,\{#s'\#\} + ce),e,(ss',csp + \{#s'\#\} + ce)) \in gtrs \\
& \implies P
\end{align*}
\]

shows P

proof (rule gtr-rev-cases[OF STEP])

case goal1 thus thesis using LOC by auto

next

case goal2 note CASE=this

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

from env-no-step-s[OF CASE'(3)] obtain csp where EQ: \{#s'\#\} + cc = csp + cc by blast

thus thesis proof (cases rule: mset-unplsm-dist-cases)

case left note CC=this

with CASE' have cc' = \{#ss'\#\} + (csp - \{#s'\#\}) + cc by (auto simp add: union-assoc)

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

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=\{#ss\#\} +\{#s'\#\} + (ce - \{#s'\#\}) by (simp add: union-assoc)

moreover from CC(2) CASE'(2) have cc' = \{#ss'\#\} + csp + (ce - \{#s'\#\}) by (simp add: union-assoc)

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

by (simp add: union-ac)

ultimately show thesis using CASE'(3) CASE(3) CC(1) ENV by auto

qed

qed

5.4 Explicit local context

In the multiset semantics, a single thread has no identity. This may become a problem when reasoning about a fixed thread during an execution. For example, in our constraint-system-based approach the operational characterization of the least solution of the constraint system requires to state properties of the steps of the initial thread in some execution. With the multiset semantics, we are unable to identify those steps among all steps. There are many solutions to this problem, for example, using thread ids
either as part of the thread’s configuration or as part of the whole configuration by using lists of stacks or maps from ids to stacks as configuration datatype.

In the following we present a special solution that is strong enough to suit our purposes but not meant as a general solution. Instead of identifying every single thread uniquely, we only distinguish one thread as the local thread. The other threads are environment threads. We then attach to every step the information whether it was on the local or on some environment thread.

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

### 5.4.1 Lifted step datatype

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

**definition**

\[ \text{loc } w \equiv \text{filter } (\lambda e. \text{case } e \text{ of } \text{LOC } a \Rightarrow \text{True} \mid \text{ENV } a \Rightarrow \text{False}) \ w \]

**definition**

\[ \text{env } w \equiv \text{filter } (\lambda e. \text{case } e \text{ of } \text{LOC } a \Rightarrow \text{False} \mid \text{ENV } a \Rightarrow \text{True}) \ w \]

**definition**

\[ \text{le-rem-s } e \equiv \text{case } e \text{ of } \text{LOC } a \Rightarrow a \mid \text{ENV } a \Rightarrow a \]

Standard simplification lemmas

**lemma** loc-env-simps [simp]:

\[ \text{loc } [] = [] \]

\[ \text{env } [] = [] \]

by (unfold loc-def env-def) auto

**lemma** loc-single [simp]: \( \text{loc } [a] = (\text{case } a \text{ of } \text{LOC } e \Rightarrow [a] \mid \text{ENV } e \Rightarrow []) \)

by (unfold loc-def) (auto split: el-step.split)

**lemma** loc-uncons [simp]:

\( \text{loc } (a \# b) = (\text{case } a \text{ of } \text{LOC } e \Rightarrow [a] \mid \text{ENV } e \Rightarrow [])@loc b \)

by (unfold loc-def) (auto split: el-step.split)

**lemma** loc-unconc [simp]: \( \text{loc } (a @ b) = \text{loc } a \@ \text{loc } b \)

by (unfold loc-def, simp)

**lemma** env-single [simp]: \( \text{env } [a] = (\text{case } a \text{ of } \text{LOC } e \Rightarrow [] \mid \text{ENV } e \Rightarrow [a]) \)

by (unfold env-def) (auto split: el-step.split)

**lemma** env-uncons [simp]:

\( \text{env } (a \# b) = (\text{case } a \text{ of } \text{LOC } e \Rightarrow [] \mid \text{ENV } e \Rightarrow [a]) @ \text{env } b \)

by (unfold env-def) (auto split: el-step.split)

**lemma** env-unconc [simp]: \( \text{env } (a @ b) = \text{env } a @ \text{env } b \)

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

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

**5.4.2 Definition of the loc/env-semantics**

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

**inductive-set**
gtrp :: (′s el-conf, ′l el-step) 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,#sl#)+c),e,(s',#sl'#+c')∈S
           ⇒ ((sl,#s#)+c),ENV e,(sl,#s'#+c')∈gtrp S

**5.4.3 Relation between multiset- and loc/env-semantics**

**lemma gtrp2gtr-s [simp]:**
((s,c),e,(s',c'))∈gtrp T ⇒ (#s#)+c,le-rem-s e,#s'#+c'∈gtr T
**proof** (cases rule: gtrp.cases, auto intro: gtrI-s)
  fix c c' e ss ss'
  assume ((ss,#s#)+c),e,(ss',#s'#+c')∈T
  hence ((#ss#)+(#ss#)+c),e,#ss'#+((#ss#)+c')∈gtr T by (auto intro: gtrI-s)
  thus (#s#)+((#ss#)+c),e,#s'#+((#ss#)+c')∈gtr T by (auto simp add: union-ac)
  qed
lemma \texttt{gtrp2gtr}:
\[(s,c),w,(s',c')\in \text{trcl (gtr T)} \rightarrow ((s#)+c,s'c)\in \text{trcl (gtr T)}\]
by (induct rule: trcl-pair-induct) (auto dest: gtrp2gtr)

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

qed

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

qed

5.4.4 Invariants

lemma gtrp-preserve-s:
assumes A: ((s,c),e,(s',c'))∈gtrp T
and INIT: P (\{#s#\}+c)
and PRES: !s c s' c' e. [P ((s,c),e,(s',c')): (s',c')\in T]
shows P (\{#s'\#\}+c')
proof −
from gtr-preserve-s[OF gtrp2gtr-s[OF A], where P=P, OF INIT] PRES show P ((\{#s'\#\} + c') by blast
qed

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

end

6 Flowgraphs

theory Flowgraph
imports Main Misc
begin

We use a flowgraph-based program model that extends the one we used previously [6]. A program is represented as an edge annotated graph and a set of procedures. The nodes of the graph are partitioned by the procedures, i.e. every node belongs to exactly one procedure. There are no edges between nodes of different procedures. Every procedure has a distinguished entry and return node and a set of monitors it synchronizes on. Additionally, the program has a distinguished main procedure. The edges are annotated with statements. A statement is either a base statement, a procedure call or a thread creation (spawn). Procedure calls and thread creations refer to the called procedure or to the initial procedure of the spawned thread, respectively.
We require that the main procedure and any initial procedure of a spawned thread does not to synchronize on any monitors. This avoids that spawning of a procedure together with entering a monitor is available in our model as an atomic step, which would be an unrealistic assumption for practical problems. Technically, our model would become strictly more powerful without this assumption.

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

6.1 Definitions

datatype \( ('p, 'ba) \) edgeAnnot = Base 'ba | Call 'p | Spawn 'p

type-synonym \( ('n, 'p, 'ba) \) edge = ('n \times ('p, 'ba) edgeAnnot \times 'n)

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

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

lemma main-is-initial[simp]: \( \text{initialproc } \text{fg } (\text{main } \text{fg}) \)
  by (unfold initialproc-def, simp)

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

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

6.2 Basic properties

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

lemma (in flowgraph) main-no-mon[simp]: \( \text{mon } \text{fg } (\text{main } \text{fg}) = \{\} \)
  using \( \text{initial-no-mon } \) by (unfold initialproc-def, blast)

lemma (in flowgraph) entry-return-same-proc[simp]:
entry \( \text{fg} \ p = \text{return} \ \text{fg} \ p' \implies p = p' \)
apply (subgoal-tac proc-of \( \text{fg} \) (entry \( \text{fg} \) \( p \)) = proc-of \( \text{fg} \) (return \( \text{fg} \) \( p' \))))
apply (simp (no-asm-use))
by simp

lemma (in flowgraph) entry-entry-same-proc[simp]:
entry \( \text{fg} \) \( p \) = entry \( \text{fg} \) \( p' \) \( \implies \) \( p = p' \)
apply (subgoal-tac proc-of \( \text{fg} \) (entry \( \text{fg} \) \( p \)) = proc-of \( \text{fg} \) (entry \( \text{fg} \) \( p' \))))
apply (simp (no-asm-use))
by simp

lemma (in flowgraph) return-return-same-proc[simp]:
return \( \text{fg} \) \( p \) = return \( \text{fg} \) \( p' \) \( \implies \) \( p = p' \)
apply (subgoal-tac proc-of \( \text{fg} \) (return \( \text{fg} \) \( p \)) = proc-of \( \text{fg} \) (return \( \text{fg} \) \( p' \))))
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 \( \text{fg} \) \( p \) ==
(\( \forall \ u \ l \ . \ \neg(u,l,\text{return} \ \text{fg} \ p)\in\text{edges} \ \text{fg} \) \land entry \( \text{fg} \) \( p \) \( \neq \) return \( \text{fg} \) \( p \))

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 \( \text{fg} \) \( p \) \( \implies \) entry \( \text{fg} \) \( p \) \( \neq \) return \( \text{fg} \) \( p \)
— The only outgoing edges of initial procedures’ entry nodes are call edges to
procedures with isolated return node
assumes initial-call-no-ret: \([\text{initialproc} \ \text{fg} \ \text{p} \ (\text{entry} \ \text{fg} \ \text{p},\text{l,v})\in\text{edges} \ \text{fg}]\)
\( \implies \exists \text{p'}. \ \text{l=Call} \ \text{p}' \ \land \ \text{isolated-ret} \ \text{fg} \ \text{p'} \)
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.

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

```plaintext
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

```plaintext
theory Semantics
imports Main Flowgraph ~~/src/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.

```plaintext
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** $(\mathit{p}, \mathit{ba}) \text{ label} = \text{LBase } \mathit{ba} \mid \text{LCall } \mathit{p} \mid \text{LRet} \mid \text{LSpawn } \mathit{p}$

### 7.2 Monitors

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

**definition**

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

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

**definition**

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

$$\text{mon-}s \ fg \ s \ == \ \bigcup \ \{ \ \text{mon-}n \ fg \ n \mid n \ . \ . n \in \text{set } s \ \}$$

**definition**

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

$$\text{mon-}c \ fg \ c \ == \ \bigcup \ \{ \ \text{mon-}s \ fg \ s \mid s \ . \ s : \# \ c \ \}$$

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

**definition** $\text{mon-e :: } (\mathit{b}, \mathit{c}, \mathit{d}, \mathit{a}, \mathit{e}) \text{ flowgraph-rec-scheme } \Rightarrow (\mathit{c}, \mathit{f}) \text{ label } \Rightarrow \mathit{a}$

**set where**

$$\text{mon-e } fg \ e \ = \ (\text{case } e \ of \ \text{LCall } \mathit{p} \ \Rightarrow \ \text{mon } fg \ \mathit{p} \ \mid \ . \ \Rightarrow \ {\}})$$

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

$$\text{mon-e } fg \ (\text{LBase } \mathit{a}) \ = \ {\}$$
$$\text{mon-e } fg \ (\text{LCall } \mathit{p}) \ = \ \text{mon } fg \ \mathit{p}$$
$$\text{mon-e } fg \ (\text{LRet}) \ = \ {\}$$
$$\text{mon-e } fg \ (\text{LSpawn } \mathit{p}) \ = \ {\}$$
**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**

$$\text{mon-w } fg \ w \ == \ \bigcup \ \{ \ \text{mon-e } fg \ e \mid e. \ e \in \text{set } w \}$$

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

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

**lemma** $\text{mon-c-alt: } \text{mon-c } fg \ c \ == \ \bigcup \ (\text{mon-}s \ fg \ \mathit{set-of } c)$

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

**lemma** $\text{mon-w-alt: } \text{mon-w } fg \ w \ == \ \bigcup \ (\text{mon-e } fg \ \mathit{set } w)$

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

**lemma** $\text{mon-sl: } [\ n \in \text{set } s \ ; \ m \in \text{mon-}n \ fg \ n \ ] \ \Rightarrow \ m \in \text{mon-s } fg \ s$

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

**lemma** $\text{mon-sD: } m \in \text{mon-s } fg \ s \ \Rightarrow \ \exists \ n \in \text{set } s. \ m \in \text{mon-}n \ fg \ n$
lemma mon-n-same-proc:
\[\text{proc-of } fg \ n = \text{proc-of } fg \ n' \implies \text{mon-n } fg \ n = \text{mon-n } fg \ n'\]
by (unfold mon-n-def, simp)

lemma mon-s-same-proc:
\[\text{proc-of } fg ^ \set s = \text{proc-of } fg ^ \set s' \implies \text{mon-s } fg \ s = \text{mon-s } fg \ s'\]
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-s-empty[simp]:
\[\text{mon-c } fg \{\#\} = \{\}\]
by (unfold mon-c-def) auto

lemma mon-s-unconc: mon-s fg (a@b) = mon-s fg a ∪ mon-s fg b
by (unfold mon-s-def) auto

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

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

lemma mon-cl: [s:#c; m∈mon-s fg s] ⇒ m∈mon-c fg c
by (unfold mon-c-def, auto)

lemma mon-cD: [m∈mon-c fg c] ⇒ ∃ s. s:#c ∧ m∈mon-s fg s
by (unfold mon-c-def, auto)

lemma mon-s-mono: set s ⊆ set s' ⇒ mon-s fg s ⊆ mon-s fg s'
by (unfold mon-s-def) auto

lemma mon-c-mono: c≤c' ⇒ mon-c fg c ⊆ mon-c fg c'
by (unfold mon-c-def) (auto intro: mset-le-trans-clem)

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

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

lemma mon-w-unconc: mon-w fg (wa@wb) = mon-w fg wa ∪ mon-w fg wb
by (unfold mon-w-def) auto

lemma mon-w-uncons[simp]: mon-w fg (e#w) = mon-e fg e ∪ mon-w fg w
by (rule mon-w-unconc[where wa=[e], simplified])
lemma mon-w-ileq: \( w \leq w' \Longrightarrow mon-w \ fg \ w \subseteq mon-w \ fg \ w' \)
by (induct rule: less-eq-list-induct) auto

7.3 Valid configurations

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

definition valid fg c == \( \forall s s'. \{\#s\}\{\#s'\} \leq c \longrightarrow mon-s \ fg \ s \cap mon-s \ fg \ s' = \{\} \)

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

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

lemma valid-split1:
valid fg (c+c') \Longrightarrow valid fg c \land valid fg c' \land mon-c \ fg \ c \cap 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\}\{\#s'\} \leq c+c')
done

lemma valid-split2:
[valid fg c; valid fg c'; mon-c \ fg \ c \cap mon-c \ fg \ c' = \{\}] \Longrightarrow 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-unconc:
valid fg (c+c') \longleftrightarrow (valid fg c \land valid fg c' \land mon-c \ fg \ c \cap mon-c \ fg \ c' = \{\})
by (blast dest: valid-split1 valid-split2)

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

34
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 == ∃ su sv. {#su#} + {#sv#} ≤ c ∧ atU-s U su ∧ atU-s V sv

lemma atUV-empty[simp]: ¬atUV U V {#}
  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-xchange-stack: atU U ({u#r#} + c) ⇒ atU U ({u#r'r#} + c)
  by (simp)

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

definition atUV U V c == ∃ su sv. {#su#} + {#sv#} ≤ c ∧ atU-s U su ∧ atU-s V sv

lemma atUV-empty[simp]: ¬atUV U V {#}
lemma atUV-single[simp]: \( \neg \text{atUV } U \cap V \) 

by (unfold atUV-def) auto

lemma atUV-union[simp]:
\[
\text{atUV } U \cap V (c_1 + c_2) \leftrightarrow (atUV U V c_1) \lor (atUV U V c_2) \lor (atU U c_1 \land atU V c_2) \lor (atU V c_1 \land atU U c_2)
\]
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-union-cases[case-names left right lr rl, consumes 1]:
\[
\begin{align*}
&[ \quad \text{atUV } U \cap V (c_1 + c_2); \\
&\quad \text{atUV } U \cap V c_1 \Rightarrow P; \\
&\quad \text{atUV } U \cap V c_2 \Rightarrow P; \\
&\quad [\text{atU } U c_1; \text{atU } V c_2] \Rightarrow P; \\
&\quad [\text{atU } V c_1; \text{atU } U c_2] \Rightarrow P \\
\end{align*}
\]
by auto

7.5 Operational semantics

7.5.1 Semantic reference point

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

inductive-set refpoint :: (′n, ′p, ′ba, ′m, ′more) flowgraph-rec-scheme ⇒ (′n conf × (′p, ′ba) label × ′n conf) set

for fg

where
\[
\begin{align*}
&\text{— A base edge transforms the top node of one stack and leaves the other stacks} \\
&\text{untouched.} \\
&\text{refpoint-base: } [ (u, \text{Base } a, \nu) \in \text{edges } fg; \ \text{valid } fg \ (\{\#u\#r\#\} + c) ] \ \Rightarrow (\{\#u\#r\#\} + c, \text{LBase } a, \{\#v\#r\#\} + c) \in \text{refpoint } fg \\
&\text{— A call edge transforms the top node of a stack and then pushes the entry node} \\
&\text{of the called procedure onto that stack. It can only be executed if all monitors the} \\
&\text{called procedure synchronizes on are available. Reentrant monitors are modeled} \\
&\text{here by checking availability of monitors just against the other stacks, not against} \\
&\text{the stack of the thread that executes the call. The other stacks are left untouched.} \\
&\text{refpoint-call: } [ (u, \text{Call } p, \nu) \in \text{edges } fg; \ \text{valid } fg \ (\{\#u\#r\#\} + c); \\
\end{align*}
\]
mon fg p ∩ mon-c fg c = \{\} \]

implies ((\#u\#\#)+c), LCall p, (#entry fg p\#v\#r\#)+c) ∈ refpoint fg |

— A return step pops a return node from a stack. There is no corresponding
flowgraph edge for a return step. The other stacks are left untouched.

refpoint-ret: \[ valid fg ((\#return fg p\#r\#)+c) \]

implies ((\#return fg p\#r\#)+c), LRet((\#r\#)+c) ∈ refpoint fg |

— A spawn edge transforms the top node of a stack and adds a new stack to
the environment, with the entry node of the spawned procedure at the top and no
stored return addresses. The other stacks are also left untouched.

refpoint-spawn: \[ (u, Spawn p, v) ∈ edges fg; valid fg ((\#u\#r\#)+c) \]

implies ((\#u\#r\#)+c), LSpawn p, (#v\#r\#)+c) ∈ refpoint fg |

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

**inductive-set**

\[ trss : (\langle n, p, b, a, m, \text{more} \rangle \rangle flowgraph-rec-scheme \Rightarrow \]

\[ (\langle n \text{ list } \star \langle n \text{ conf} \rangle \rangle \rangle \langle p, b, a \rangle \rangle \langle n \text{ list } \star \langle n \text{ conf} \rangle \rangle \rangle \]

set

for fg

where

\[ trss-base: [(u, Base a, v) ∈ edges fg] \Rightarrow \]

\[ ((u\#r,c), LBase a, (v\#r,c)) ∈ trss fg \]

| trss-call: [(u, Call p, v) ∈ edges fg; mon fg p ∩ mon-c fg c = \{\} \]

implies ((u\#r,c), LCall p, (#entry fg p\#v\#r\#)+c) ∈ trss fg |

| trss-ret: (((\#return fg p\#r\#)+c), LRet,(r,c)) ∈ trss fg |

| trss-spawn: \[ (u, Spawn p, v) ∈ edges fg \]

implies ((u\#r,c), LSpawn p, (#v\#r\#)+c) ∈ trss fg |

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

**abbreviation** tr where tr fg == gtr (trss fg)

— We also generate the loc/env-semantics

**abbreviation** trp where trp fg == gtrp (trss fg)

### 7.6 Basic properties

#### 7.6.1 Validity

**lemma** (in flowgraph) trss-valid-preserve-s:

\[ [valid fg ((\#s\#)+c); ((s,c), e, (s’,c’)) ∈ trss fg] \Rightarrow valid fg ((\#s’\#)+c’) \]

**apply** (erule trss.cases)

**apply** (simp-all add: valid-unconc mon-c-unconc)

37
by (blast dest: mon-nsame-proc edges-part)+

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

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

lemma [in flowgraph] trp-valid-preserve-s:
\[ [(s,c),w,(s',c') \in trcl (tr fg); valid fg \{\#s\#\} + c] \implies valid fg \{\#s'\#\} + c' \]
by (rule gtrp-preserve-s [where \( P = valid fg \)]) (auto dest: trss-valid-preserve-s)

lemma [in flowgraph] trp-valid-preserve:
\[ [(s,c),w,(s',c') \in trcl (tr pg); valid fg \{\#s\#\} + c] \implies valid fg \{\#s'\#\} + c' \]
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,e,c') \in refpoint fg) \iff ((c,e,c') \in tr fg)
  apply rule
  apply (erule refpoint_cases)
  apply (auto intro: gtrI-s trss.intros simp add: union-assoc)
  apply (erule gtrE)
  apply (erule trss_cases)
  apply (auto intro: refpoint.intros simp add: union-assoc[symmetric])
  done

lemma [in flowgraph] refpoint-eq:
valid fg c \iff ((c,w,c') \in trcl (refpoint fg)) \iff ((c,w,c') \in trcl (tr fg))

proof
  have ((c,w,c') \in trcl (refpoint fg)) \iff valid fg c \iff ((c,w,c') \in trcl (tr fg))
  by (induct rule: trcl.induct) (auto simp add: refpoint-eq-s trss-valid-preserve-s)

  moreover have ((c,w,c') \in trcl (tr fg)) \iff valid fg c \iff ((c,w,c') \in trcl (refpoint fg))
  by (induct rule: trcl.induct) (auto simp add: refpoint-eq-s trss-valid-preserve-s)

  ultimately show valid fg c \iff ((c,w,c') \in trcl (refpoint fg)) = ((c,w,c') \in trcl (tr fg))
qed

7.6.3 Case distinctions

lemma trss-c-cases-s [cases set, case_names no-spawn spawn]: []
\[ ((s,c),e,(s',c')) \in trss fg; \]
\[
\begin{align*}
\text{lemma \ trss-c-fmt-s:} & \quad [((s,c),e,(s',c'))\in \text{trss } fg] \\
& \quad \Rightarrow \exists \text{ csp. } c' = \text{csp} + c \land \\
& \quad (\text{csp}\{\#\} \lor (\exists \text{ p. } e=\text{LSpawn p} \land \text{csp}\{\#|\text{ entry fg p}|\#\})) \\
& \quad \text{by (force elim!: trss-c-cases-s)}
\end{align*}
\]

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

\[
\begin{align*}
\text{lemma trss-c-cases\{cases set, case-names c-case\}:} & \quad [\]
\text{!!s c.} \\
& \quad [\text{(s,c),w,(s',c'))}\in \text{trcl} \text{ (trss } fg);} \\
& \quad \text{!!csp. } c' = \text{csp} + c; \text{ !!s:}\text{#csp} \Rightarrow \exists \text{ p u v. } s=\text{[entry fg p]} \land \\
& \quad (u,\text{Spawn p,v})\in \text{edges fg } \land \\
& \quad \text{initialproc } fg \text{ p}] \\
& \quad \Rightarrow P \\
& \quad [\]
& \quad \text{proof (induct w)} \\
& \quad \text{case Nil note A=\text{this}} \\
& \quad \text{hence } s' = s c' = c \text{ by simp-all} \\
& \quad \text{hence } c' = \{\#\} + c \text{ by simp} \\
& \quad \text{from A(2)(OF this) show } P \text{ by simp}
\end{align*}
\]

\[
\begin{align*}
\text{next} \quad \text{case (Cons e w) note IHP=\text{this}} \\
& \quad \text{then obtain sh ch where SPLIT1: } [(s,c),e,(sh,ch))\in \text{trss } fg \text{ and SPLIT2:} \\
& \quad [(sh,ch),w,(s',c'))\in \text{trcl} \text{ (trss } fg) \text{ by (fast dest: trcl-uncons)} \\
& \quad \text{from SPLIT2 show } ?\text{case proof (rule IHP(1))} \\
& \quad \text{fix csp} \\
& \quad \text{assume } C'\text{FMT: } c' = \text{csp} + c \text{ and CSPFMT: } \forall s. \text{s :\# csp} \Rightarrow \exists \text{ p u v. } s=\text{[entry fg p]} \land \\
& \quad (u,\text{Spawn p,v})\in \text{edges fg } \land \text{initialproc } fg \text{ p} \\
& \quad \text{from SPLIT1 show } ?\text{thesis proof (rule trss-c-cases-s)} \\
& \quad \text{assume ch}=c \text{ with } C'\text{FMT CSPFMT IHP(3) show } ?\text{case by blast}
\end{align*}
\]

\[
\begin{align*}
\text{next} \quad \text{fix p} \\
& \quad \text{assume EFMT: } e=\text{LSpawn p and CHFMT: } ch=\{\#[\text{entry fg p}]\#\} + c \\
& \quad \text{with } C'\text{FMT have } c' = \{\#[\text{entry fg p}]\#\} + c \text{ by (simp add: union-ac)} \\
& \quad \text{moreover from EFMT SPLIT1 have } \exists \text{ u v. } (u,\text{Spawn p, v})\in \text{edges fg by (blast elim!:)}
\end{align*}
\]
lemma (in flowgraph) c-of-initial-no-mon:
assumes A: !!s. s: # csp =⇒ ∃ p. s=[entry fg p] ∧ initialproc fg p
shows mon-c fg csp = {} by (unfold mon-c-def) (auto dest: A initial-no-mon)

lemma (in flowgraph) trss-c-no-mon-s:
assumes A: ((s,c),e,(s',c'))∈trss fg
shows mon-c fg c' = mon-c fg c using A
proof (erule-tac trss-c-cases-s)
assume c'=c thus ?thesis by simp
next
  fix p assume EFMT: e=L Spawn 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-anconc)
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. 0 < count csp s =⇒ (∃ p. s=[entry fg p] ∧ (∃ u v. (u, Spawn p,v)∈edges fg) ∧ initialproc fg p)
  ultimately obtain p u v where s=[entry fg p] and (u,Spawn p,v)∈edges fg by blast
    hence mon-s fg s = {} by (simp)
    with M have False by simp
    thus x∈mon-c fg c ..
qed
lemma (in flowgraph) trss-spawn-no-mon-step[simp]:
\[ ((s,c), LSpawn p, (s',c')) \in \text{trss} \text{ fg} \implies \text{mon} \text{ fg} \ p = \{ \} \]
by (auto elim: trss.cases)

lemma trss-no-empty-s[simp]: \[(([],c),e,(s',c')) \in \text{trss} \text{ fg} = \text{False} \]
by (auto elim!: trss.cases)

lemma trss-no-empty[simp]:
assumes \( A \): \( (([],c),w,(s',c')) \in \text{trcl} \ (\text{trss} \text{ fg}) \)
shows \( w=[] \land s'=[] \land c=c' \)
proof -
  note \( A \)
  moreover {
    fix \( s \)
    have \( ((s,c),w,(s',c')) \in \text{trcl} \ (\text{trss} \text{ fg}) \implies s=[] \implies w=[] \land s'=[] \land c=c' \)
      by (induct rule: trcl-pair-induct) auto
  } ultimately show \(?thesis\) by blast
qed

lemma trs-step-cases[cases set, case-names NO-SPAWN SPAWN]:
assumes \( A \): \((c,e,c')\in\text{tr} \text{ fg} \)
assumes \( A-NO-SPAWN\): \(!!s ce s'\text{ esp. \[ ((s,ce),e,(s',ce)) \in \text{trss} \text{ fg}; \]
  c=\#s\#+ce; c'=\#s'\#+ce \] \implies P \)
assumes \( A-SPAWN\): \(!!s ce s' p. \[ ((s,ce),LSpawn p,(s',\#[entry fg p]\#+ce)) \in \text{trss} \text{ fg}; \]
  c=\#s\#+ce;
  c'=\#s'\#+\#[entry fg p]\#+ce;
  e=LSpawn p \] \implies P \)
shows \( P \)
proof -
  from \( A \) show \(?thesis\) proof (erule-tac gtr-find-thread)
    fix \( s ce s' ce' \)
    assume \( FMT\): \( c = \#s\# + ce \land c' = \#s'\# + ce' \)
    assume \( B\): \( ((s, ce), e, s', ce') \in \text{trss} \text{ fg} \) thus \(?thesis\) proof (cases rule: trss-c-cases-s)
        case no-spawn thus \(?thesis\) using \( FMT \) \( B \) by \( (\) rule \( A-NO-SPAWN\), \( auto\) \)
        next
        case \( (\text{spawn} p) \) thus \(?thesis\) using \( FMT \) \( B \) by \( (\) rule \( A-SPAWN\), \( auto\) \ simp add: union-assoc)\n    qed
  qed
qed

41
7.7 Advanced properties

7.7.1 Stack composition / decomposition

**Lemma** \(\text{trss-stack-comp-s}:

\[
((s,c), e, (s',c')) \in \text{trss } fg \implies ((s@r, c), e, (s'@r, c')) \in \text{trss } fg
\]

by (auto elim!: \text{trss.cases intro: } \text{trss.intros})

**Lemma** \(\text{trss-stack-comp}:

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

**Proof** (induct rule: \text{trcl-pair-induct})

- **Case** empty thus ?case by auto

  **Case** (cons s c e sh ch w s' c') note IHP=this from \text{trss-stack-comp-s}[OF IHP(1)] have ((s @ r, c), e, sh @ r, ch) \in \text{trss } fg . also note IHP(3) finally show ?case .

**Qed**

**Lemma** \(\text{trss-stack-decomp-s}:

\[
[(s@r, c), e, (s',c')] \in \text{trss } fg; \ s \neq []
\]

\[
\implies \exists sp'. \ s'=sp'@r \land ((s,c), e, (sp',c')) \in \text{trss } fg
\]

by (cases s, simp) (auto intro: \text{trss.intros elim!: } \text{trss.cases})

**Lemma** \(\text{trss-find-return}:

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

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

\[
(r, ch), wb, (r', c') \in \text{trcl } (\text{trss } fg) \implies P
\]

— If \(s = []\), the proposition follows trivially

apply (cases s=[])

apply fastforce

**Proof**

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

have IM: \(!s \ c. \ [(s@r, c), w, (r',c')] \in \text{trcl } (\text{trss } fg); \ s \neq []\) \implies \exists wa \ wb \ ch.

\[
w=wa@wb \land ((s,c), wa,([],ch)) \in \text{trcl } (\text{trss } fg) \land ((r, ch), wb, (r', c')) \in \text{trcl } (\text{trss } fg)
\]

**Proof** (induct \(w\))

**Case** Nil thus ?case by (auto)

**Next**

**Case** (Cons e w) note IHP=this then obtain sh ch where SPLIT1: ((s@r, c), e, (sh, ch)) \in \text{trss } fg and SPLIT2:

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

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

  with SPLIT1 obtain \(p\) where EDGE: \(s@r=\text{return } fg \ p \neq \ sh \ c=ch\) by (auto elim!: \text{trss.cases})

  with IHP(3) obtain ss where SHFMT: \(s=\text{return } fg \ p \neq \ ss \ sh=ss@r\) by \(\text{cases } s, \ \text{auto}\)

  { assume CC: \(ss \neq []\)

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

    moreover {

      assume CC: \(ss = []\)

      }
by simp
ultimately have ?case by blast
\}
ultimately have ?case \ve (\exists ss. ss\neq[] \et sh=ss@r) by blast
\}
moreover \{
assume e\neq LRet
with SPLIT1 IHP(3) have (\exists ss. ss\neq[] \et sh=ss@r) by (force elim!: trss.cases
simp add: append.eq-Cons-conv)
\}
moreover \{
assume (\exists ss. ss\neq[] \et sh=ss@r)
then obtain ss where CASE: ss\neq[] sh=ss@r by blast
with SPLIT2 have ((ss@r, ch), w, r, c') \in trcl (trss fg) by simp
from IHP(1)|OF this CASE(1) obtain wa wb ch' where IHP: w=wa@wb ((ss,ch),wa,([],ch')) \in trcl (trss fg) ((r,ch'),wb,(r,c')) \in trcl (trss fg) by blast
moreover from CASE SPLIT1 have ((s @ r, c), e, ss@r, ch) \in trss fg by simp
from trss-stack-decomp-s|OF this IHP(3) have ((s, c), e, ss, ch) \in trss fg by auto
with IHP have ((s, c), e\#wa, ([]',ch')) \in trcl (trss fg) by (rule-tac trcl.cons)
moreover from IHP have e\#wa=(e\#wa)@wb by auto
ultimately have ?case by blast
\}
ultimately show ?case by blast
qed

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

lemma trss-return-cases[cases set]: !!u r c. [ (u\#r,c),w,(r',c') \in trcl (trss fg);
!! s' u'. [ r'=s' @ u'\#r; (([u],c),w,(s'@[u'],c')) \in trcl (trss fg) ] \implies P;
!! wa wb ch. [ w=wa@wb; (([u],c),wa,([],ch)) \in trcl (trss fg); ((r,ch),wb,(r',c')) \in trcl (trss fg) ] \implies P ] \implies P
proof (induct w rule: length-compl-induct)
case Nil thus ?case by auto
next
case (Cons e w) note IHP=this
then obtain sh ch where SPLIT1: ((u\#r,c),e,(sh,ch)) \in trss fg and SPLIT2: ((sh,ch),w,(r',c')) \in trcl (trss fg) by (fast dest: trcl-ancons)
{ 
fix ba q
assume CASE: e=LBase ba \et e=LSpawn q
with SPLIT1 obtain v where E: sh=v\#r (((u],c),e,([v],ch)) \in trss fg by (auto elim!: trss.cases intro: trss.intros)
with SPLIT2 have \((v\#r, ch), w, (r', c'))\in\text{trcl (trss fg)} by simp 

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

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

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

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

next 

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

with \(E(2)\) have \(((u, c), e\# wa, ([], ct))\in\text{trcl (trss fg)}\) 

\(e\# w = (e\# wa)@wb\) by simp-all

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

qed 

\}

moreover 

\}

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

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

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

}\)  

moreover 

\}

fix \(q\)

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

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

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

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

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

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

\(((\text{entry fg} q \# [u'], ch), w, (st @ [ul]) @ [u'], c')\in\text{trcl (trss fg)}\) by auto 

with \(\text{SHFMT}(2)\) have 

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

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

next 

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

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

\(((\text{entry fg} q \# [u'], ch), wa, [u'], ct)\in\text{trcl (trss fg)}\) by simp 

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

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

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

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

CCC(1) have 

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

from IHP(3)[OF CCC(1) this] show \(?thesis\).

next 

case \((2 \text{ wba wbb c'})\) note CC=this from trcl-concat[OF PREPATH CCC(2)] 

CC(1) CCC(1) have \(e\# w = (e\# w@wba)@wbb (([u], c), e \# wa @ wba, [], c')\in\text{trcl (trss fg)}\) by auto 

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

qed 

qed 

}\) ultimately show \(?case by (cases e, auto)

qed
lemma (in flowgraph) trss-find-call:

\[ \forall v \ r' \ e'. \ [ (([sp], c), w, (v\#r', e')) \in \text{trcl} (\text{trss} fg); \ r' \neq [] \] \implies \exists rh \ ch \ p \ wa \ wb.

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

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

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

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

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

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

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

next

case (snoc w e) note IHP\(=\)this

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

\[
\{ \\
\text{assume } \exists u. \ rh = u \# r' \\
\text{then obtain } u \text{ where RHFFMT[simp]: } rh = u \# r' \text{ by blast} \\
\text{with SPLIT2 have } \text{proc-of} \ fg \ u = \text{proc-of} \ fg \ v \text{ by (auto elim: trss.cases intro: edges-part)} \\
\text{moreover from IHP(1) of w u r' ch, OF - SPLIT1[simplified] IHP(3) obtain}\rt ct p wa wb where \\
\text{IHAPP: } w = wa \@ \text{LCall} p \# wb \text{ proc-of} \ fg \ u = p \ ((([sp], c), wa, (rt, ct)) \in \text{trcl} (\text{trss} fg) ((rt, ct), \text{LCall} p, \text{entry} fg p \# r', ct) \in \text{trss} fg \\
(((\text{entry} fg p, ct), wb, ([u], ch)) \in \text{trcl} (\text{trss} fg) \text{ by (blast)} \\
\text{moreover} \\
\text{have } (((\text{entry} fg p, ct), wb\@[e], ([v], c')) \in \text{trcl} (\text{trss} fg) \text{ proof = } \\
\text{note IHAPP(3)} \\
\text{also from SPLIT2 have } (((u], ch), e, ([v], c')) \in \text{trss} fg \text{ by (auto elim!: trss.cases intro!: trss intros)} \\
\text{finally show } \text{thesis} . \\
\text{qed} \\
\text{moreover from IHAPP have } w\@[e] = wa \@ \text{LCall} p \# (wb\@[e]) \text{ by auto} \\
\text{ultimately have } \forall \text{case by auto} \\
\}
\]

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

proof (rule trss.cases[OF SPLIT2, simp-all]) — Cases for base- and spawn edge are discharged automatically

— Case: call-edge

case (goal1 ca p r u vv) with SPLIT1 SPLIT2 show \( \forall \)case by fastforce

next

— Case: return edge

case (goal2 q r ca) note CC\(=\)this

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

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

IHAPP: \( w = wa \@ \text{LCall} q \# wb \ ((([sp], c), wa, rt, ct) \in \text{trcl} (\text{trss} fg) ((rt, ct), \text{LCall} q, \text{entry} fg q \# v \# r', ct) \in \text{trss} fg \)
\[
((\text{entry } f \, q, \, c t), \, w b, \, [\text{return } f \, g \, q], \, c h) \in \text{trl} \, (\text{trss } f \, g) \text{ by force}
\]

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

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

from IHAPP(1) [OF LENWA IHAPP(2) [simplified] IHAP(3)] obtain \( \text{rhh } c h h \, p \)

\( \text{waa } w a b \) where

IHAPP': \( \text{waa}=w a a @ L \text{Call } p \# w a b \) \( \text{proc-of } f \, g \, u = p \) (((\([w a],[c],w a a,(\text{rhh},c h h)\)\)\in trcl (trss fg) \((\text{rhh},c h h)\),L \text{Call } p, \, (\text{entry } f \, g \, p \# r',c h h)\)\in trss fg

\( (((\text{entry } f \, g \, p),c h h),w a b,([u],c t))\in trcl \, (\text{trss } f \, g) \)

by blast

from IHAPP IHAPP' PROC-OF-U have \( \forall ([e]=[w a a @ L \text{Call } p \# (w a b @ L \text{Call } q) \# w b @ [e]), ([v],c')\)\in trcl (trss fg)

proof -

note IHAPP'(5)

also from IHAPP have \( (((u),c t),L \text{Call } q, \, \text{entry } f \, g \, q \# [v], \, c t) \in \text{trss } f \, g \) by (auto elim: trss.cases intro: trss.intro)

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

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

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

qed

moreover note IHAPP' CC

ultimately show \( \text{?case} \) by auto

qed

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

qed

— This lemma is better suited for application in soundness proofs of constraint systems than \text{flowgraph.trss-find-call}

\textbf{Lemma} (in \text{flowgraph}) \text{trss-find-call'}:

\textbf{Assumes:} \( A \): \( ((([s p],c,w,(\text{return } f \, g \, p \# [u'],c'))\in \text{trcl} \, (\text{trss } f \, g) \)

\textbf{and} \( \text{EX:} \): \( !u h \, c h \, w a \, w b \, .\)

\( w=w a a @ (L \text{Call } p) \# w b;\)

\( \text{(([s p],c,w a a,(u h),c h))}\in \text{trcl} \, (\text{trss } f \, g);\)

\( \text{((u h),c h),L \text{Call } p,((\text{entry } f \, g \, p) \# [u'],c h))\in \text{trss } f \, g;\)

\( w h, \text{Call } p, u' \in \text{edges } f g;\)

\( \text{((entry } f \, g \, p),c h),w b,((\text{return } f \, g \, p),c')\in \text{trcl} \, (\text{trss } f \, g)\)

\( \implies P \)

shows \( P \)

proof -

from \text{trss-find-call}[OF \, A] \text{ obtain } r h \, c h \, w a \, w b \text{ where FC:}

\( w = w a a @ L \text{Call } p \# w b\)

\( \text{(([s p],c), w a a, r h, c h) } \in \text{trcl} \, (\text{trss } f \, g)\)

\( \text{((r h, c h), L \text{Call } p, ([\text{entry } f \, g \, p, u'],c h) } \in \text{trss } f \, g\)

\( \text{((entry } f \, g \, p),c h),w b,([\text{return } f \, g \, p],c')\in \text{trcl} \, (\text{trss } f \, g) \)

by auto

moreover from FC(3) obtain \( u h \text{ where } ADD: \text{ rh=}[u h] \, (u h, \text{Call } p, u') \in \text{edges}\)

46
\[ \text{fg by (auto elim: trss.cases)} \]
\[
\text{ultimately show } \exists \text{thesis using EX by autoqed}
\]

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

**lemma** trss-2empty-to-2return:
\[ (((s,c),w,([]),c')) \in \text{trcl (trss fg)} ; s\notin [] \implies \exists w'. w=w@([\text{LRet}]) \land (((s,c),w',([\text{return fg p}],c')) \in \text{trcl (trss fg)}) \]
**proof** —
\[ \text{assume A: ((s,c),w,([]),c')) \in \text{trcl (trss fg)} s\notin [] \]
\[ \text{hence w\notin [] by auto} \]
\[ \text{then obtain w' e where WD: w=w@([e]) by (blast dest: list-rev-decomp)} \]
\[ \text{with A(1) obtain sh ch where SPLIT: ((s,c),w',((sh,ch)) \in \text{trcl (trss fg)} ((sh,ch),c,([]),c')) \in \text{trss fg by (fast dest: trcl-rev-uncs)} \]
\[ \text{from SPLIT(2) obtain p where e=LRet sh=[return fg p] ch=c' by (cases rule: trss.cases, auto)} \]
\[ \text{with SPLIT(1) WD show } ? \text{thesis by blast} \]
**qed**

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

### 7.7.2 Adding threads

**lemma** trss-env-increasing-s:
\[ (((s,c),e,(s',c')) \in \text{trss fg} \implies c \leq c' \]
\[ \text{by (auto elim!: trss.cases)} \]
**lemma** trss-env-increasing:
\[ (((s,c),w,(s',c')) \in \text{trcl (trss fg)} \implies c \leq c' \]

47
by (induct rule: trcl-pair-induct) (auto dest: trss-env-increasing-s order-trans)

7.7.3 Conversion between environment and monitor restrictions

**Lemma trss-mon-e-no-ctx:**
\[(s,c),e, (s',c') \in trss fg \implies mon-e fg e \cap mon-c fg c = \{\}\]
by (erule trss.cases) auto

**Lemma (in flowgraph) trss-mon-w-no-ctx:**
\[(s,c),w, (s',c') \in trcl (trss fg) \implies mon-w fg w \cap mon-c fg c = \{\}\]
by (induct rule: trcl-pair-induct) (auto dest: trss-mon-e-no-ctx simp add: trss-c-no-mon-s)

**Lemma (in flowgraph) trss-modify-context-s:**
\[\forall c, n. (s, c, e, (s', c')) \in trss fg; mon-e fg e \cap mon-c fg c n = \{\}\]
\[\implies \exists csp. c' = csp + c \land mon-c fg csp = \{\} \land ((s,c),e, (s',csp+cn)) \in trss fg\]
by (erule trss.cases) (auto intro!: trss.intros)

**Lemma (in flowgraph) trss-modify-context-s [rule-format]:**
\[\forall c, n. (s, c, e, (s', c')) \in trcl (trss fg)\]
\[\implies \forall c, n. mon-w fg w \cap mon-c fg c n = \{\}\]
\[\implies \exists csp. c' = csp + c \land mon-c fg csp = \{\} \land ((s,c),e, (s',csp+cn)) \in trss fg\]
\[\text{proof} \ (induct rule: trcl-pair-induct)\]
\[\text{case empty thus ?case by simp}\]
\[\text{next}\]
\[\text{case (cons s c e sh w s' c')} \text{ note IHP = this show ?case}\]
\[\text{proof} \ (intro allI impI)\]
\[\text{fix cn}\]
\[\text{assume MON: mon-w fg (e \# w) \cap mon-c fg en = \{\}\]
\[\text{from trss-modify-context-s[OF IHP(1)] MON obtain csp where S1: ch = csp + c mon-c fg csp = \{\}\]
\[\text{with MON have mon-w fg w \cap mon-c fg (csp+cn) = \{\}\ by (auto simp add: mon-c-unconc)}\]
\[\text{with IHP(3)[rule-format] obtain csp where S2: c' = csp + ch mon-c fg csp = \{\}\]
\[\text{((sh,csp+cn),w,(s',csp+(csp+cn))) \in trcl (trss fg) by blast}\]
\[\text{from S1 S2 have c' = csp + c mon-c fg csp = \{\} \land (s, c, e, (s',csp+cn)) \in trcl (trss fg) by (auto simp add: union-assoc mon-c-unconc)}\]
\[\text{thus \exists csp. c' = csp + c \land mon-c fg csp = \{\} \land ((s, c, e, (s',csp+cn)) \in trcl (trss fg) by blast}\]
\[\text{qed}\]
\[\text{qed}\]

**Lemma trss-add-context-s:**
\[\forall (s,c),e, (s',c') \in trss fg; mon-e fg e \cap mon-c fg c e = \{\}\]
\[\implies ((s,c+ce),e,(s',c'+ce)) \in trss fg\]
by (auto elim!: trss.cases intro!: trss.intros simp add: union-assoc mon-c-unconc)

**Lemma trss-add-context:**
\[\forall (s,c),e, (s',c') \in trcl (trss fg); mon-w fg w \cap mon-c fg c e = \{\}\]
\[\implies ((s,c+ce),w,(s',c'+ce)) \in trcl (trss fg)\]
proof (induct rule: trcl-pair-induct)
  case empty thus \( \text{?case} \) by simp

next
  case (cons s c e sh ch w s’ e’)
  note IHP="this

  from IHP(4) have \( \text{MM} \): \( \text{mon-e fg e } \cap \text{ mon-c fg ce } = \{ \} \) by auto
  from trcl.cons[OF trss-add-context-s[OF IHP(1) MM(1)]] IHP(3)[OF MM(2)]
  show \text{?case} .

qed

lemma trss-drop-context-s:
  assumes A: \( ((s,c+ce),e, (s’,e’+ce)) \in \text{trss fg} \)
  shows \( \{ \} \) by blast

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

next
  case (Cons e w)
  note IHP="this

  then obtain sh ch where SPLIT: \( ((s,c+ce), e, (sh, ch)) \in \text{trss fg} \)
  (\text{trcl (trss fg)}) by (fast dest: trcl-uncons)
  from trss-c-fmt-s[OF SPLIT(1)] obtain csp where CHFMT: \( ch = (csp + c) + ce \)
  by (auto simp add: union-assoc)
  from CHFMT trss-drop-context-s SPLIT(1) have \( ((s,c), e, (sh, csp+c)) \in \text{trss fg} \)
  \( \text{mon-e fg e } \cap \text{ mon-c fg ce } = \{ \} \) by blast+
  moreover from CHFMT IHP(1) SPLIT(2) have \( ((sh, csp+c), w, (s’, e’)) \in \text{trcl (trss fg)} \)
  \( \text{mon-w fg w } \cap \text{ mon-c fg ce } = \{ \} \) by blast+
  ultimately show \text{?case} by auto

qed

lemma trss-xchange-context-s:
  assumes A: \( ((s,c), e, (s’, csp+c)) \in \text{trss fg} \)
  and M: \( \text{mon-c fg cn } \subseteq \text{ mon-c fg e} \)
  shows \( ((s, cn), e, (s’, csp+cn)) \in \text{trss fg} \)

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

qed

lemma trss-xchange-context-
  assumes A: \( ((s,c), w, (s’, csp+c)) \in \text{trcl (trss fg)} \)
  and M: \( \text{mon-c fg cn } \subseteq \text{ mon-c fg e} \)
  shows \( ((s, cn), w, (s’, csp+cn)) \in \text{trcl (trss fg)} \)

proof –
  from trss-drop-context[of - \( \{\#\} \), simplified, OF A] have DC: \( ((s, \{\#\}), w, s’, csp) \in \text{trcl (trss fg)} \)

49
csp ∈ trcl (trss fg) mon-w fg w ∩ mon-c fg c = {} by simp-all
with M have mon-w fg w ∩ mon-c fg cn = {} by auto
from trss-add-context[OF DC(1) this] show ?thesis by auto
qed

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

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

lemma tr-add-context-s:
  [ (c,e,c')∈tr fg; mon-e fg e ∩ mon-c fg ce = {} ] \implies (c+ce,e,c'+ce)∈tr fg
  by (erule gtrE) (auto simp add: mon-c-unconc union-assoc intro: gtrI-s dest: trss-add-context-s)

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

8 Normalized Paths

theory Normalization
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

\begin{verbatim}
lemma (in cf Flowgraph) iso-ret-no-ret: \forall u c. [ [isolated-ret fg p; ]
proc-of fg u = p;
\not\exists return fg p;
(([[u], c], w, (return fg p', c')) \in trcl (trss fg) ] \Rightarrow False
\end{verbatim}

\begin{verbatim}
proof (induct w rule: length-compl-induct)
case Nil thus \_ by auto
next
case (Cons e w) note IHP = this
then obtain sh ch where SPLIT1: 
(([[u], c], e, (sh, ch)) \in trss fg and SPLIT2: 
((sh, ch), w, (return fg p', c')) \in trcl (trss fg) by (fast dest: trcl-uncons)
show \_ by auto
\end{verbatim}

\begin{verbatim}
case LRet with SPLIT1 IHP(3,4) show False by (auto elim!: trss_cases)
next
case LBase with SPLIT1 IHP(2,3) obtain v where A: sh = [v] proc-of fg v = p v\not\exists return fg p by (force elim!: trss_cases simp add: edges-part isolated-ret-def)
with IHP SPLIT2 show False by auto
next
case (LSpawn q) with SPLIT1 IHP(2,3) obtain v where A: sh = [v] proc-of fg v = p v\not\exists return fg p by (force elim!: trss_cases simp add: edges-part isolated-ret-def)
with IHP SPLIT2 show False by auto
next
case (LCall q) with SPLIT1 IHP(2,3) obtain uh where A: sh = entry fg q\#[uh] proc-of fg uh = p uh\not\exists return fg p by (force elim!: trss_cases simp add: edges-part isolated-ret-def)
with SPLIT2 have B: 
((entry fg q\#[uh], ch), w, (return fg p', c')) \in trcl (trss fg) by simp
from trss-return-cases[OF B] obtain w1 w2 ct where C: 
w1 = w1 \@ w2 length w2 \leq length w 
((entry fg q, ch), w1, ([], ct)) \in trcl (trss fg) (([uh], ct), w2, (return fg p', c')) \in trcl (trss fg) by (auto)
\end{verbatim}

51
from IHP(1)[OF C(2) IHP(2) A(2,3) C(4)] show False .

qed

— The first step of an initial procedure is a call

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

\[ ((\text{entry } fg \ p),c,e,(s',c')) \in \text{trss } fg; \text{ initialproc } fg \ p \]  
\[ \Rightarrow \exists p'. e=LCall \ p' \land \text{isolated-ret } fg \ p' \]

by (auto elim: trss.cases dest: initial-call-no-ret initial-no-ret entry-return-same-proc)

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

**lemma (in eflowgraph) no-sl-from-initial:**

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

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

shows \( \text{False} \)

proof —

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

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

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

with SPLIT(2) have \( ((\text{entry } fg \ p' \#[u'],\ ch),w',(\{v,c')\} \in \text{trcl } (\text{trss } fg) \)

by simp

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

then obtain \( wa' \) where \( (((\text{entry } fg \ p'),\ ch),wa',((\text{return } fg \ p''),ct)) \in \text{trcl } (\text{trss } fg) \) by (blast dest: trss-2empty-to-2return)

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

qed

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

**lemma (in eflowgraph) no-retsl-from-initial:**

assumes \( A: w\neq[] \)

initialproc \( fg \ p \)

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

length \( r' \leq 1 \)

shows \( \text{False} \)

proof (cases \( r' \))

case Nil with \( A(3) \) have \( (((\text{entry } fg \ p),c),w,([[],c')) \in \text{trcl } (\text{trss } fg) \) by simp

from trss-2empty-to-2return[OF this, simplified] obtain \( w' \) where \( B: w=w'@[\text{LRet}] \)

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

by (blast)

show \( \text{thesis} \) proof (cases \( w' \))

case Nil with \( B \) have \( p=q \) entry \( fg \ p = \text{return } fg \ p \) by (auto dest: trcl-empty-cons entry-return-same-proc)

with \( A(2) \) initial-no-ret show False by blast

next

case Cons hence \( w'\neq[] \) by simp
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 to derive interleaving semantics on multisets and configurations with an explicit local thread (loc/env-semantics, cf. Section 5.4).

---

\(^2\)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.
8.3 Representation property for reachable configurations

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

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

**Lemma** ntrs-is-trss-s: \((s,c,w,(s',c')) \in ntrs \quad \implies \quad (s,c,w,(s',c')) \in trcl \quad (trss \quad f g)\)

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

```latex
\begin{align*}
\text{case empty:} & \quad \text{thus by simp} \\
\text{next} & \\
\text{case (cons s c e sh ch w s' c') note IHP=this} \\
& \quad \text{from trcl-concat[OF ntrs-is-trss-s[OF IHP(1)] IHP(3)] foldl-conc-empty-eq[of e w]} \\
& \quad \text{show ?case by simp} \\
\end{align*}
```

**Qed**

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

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

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

**Lemma (in cflowgraph) ntr-sl-move-left:** $!ce \ u \ r \ w \ r' \ ce'$.

- $((\#\text{entry} fg p \#), w u, \# u \# r \ # \ # + ce) \in \text{trcl} (ntr fg);$
- $((\#\text{entry} fg p \#), w (r' \ ce') \in \text{trcl} (\text{trss} fg);$  
- \begin{align*}
    \text{initial}proc & : fg p; \\
    \text{length} & : r' \leq 1; \ w \notin []
\end{align*}

$I \Rightarrow \exists wu'. (\#\text{entry} fg p \#), wu', \# r' \# + ce' \in \text{trcl} (ntr fg)$  

**Proof:** (induct $ww$ rule: rev-induct)

- **case Nil** note $CC$=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[$OF \ CC(5,3) - CC(4)$] have False by blast thus ??case ..

**next**

- **case** (snc $ce \ wu$) note $IHP$=this
  - In the induction step, we extract the last macrostep

**then obtain** $ch \ where$ $\text{SPLIT}$: $(\#\text{entry} fg p \#), wu, ch) \in \text{trcl} (ntr fg) (ch, ce, \# u \# r \ #) + ce) \in ntr fg \ by$ (fast dest: trcl-rev-uncons)
  - The last macrostep first executes a call and then a same-level path

**from** $\text{SPLIT}(2)$ **obtain** $q$ $wus \ uh \ rh \ ceh \ uh' \ wt \ cet \ where$

**STEPFMT**: $ce=LCall \ q\#wus \ ch=(\# \ uh\#rh \ #) \#+ceh$ $\{\# \ u\#r \ #\} + ce = \{\# \ wt\#uh'\#rh \ #\} + cet ((uh\#rh\#cet),LCall \ q,(\text{entry} fg q\#uh'\#rh, ceh)) \in \text{trss} fg (\text{trcl} (\text{trss} fg) (\text{trcl} \ (\text{trss} fg))$  

by (blast elim!: gtrE ntrs.csas[simplified])
  - 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** ??case **proof** (cases rule: mset-single-cases')
  - If the sl/ret path was executed on the same thread as the last macrostep
by path − proof macrostep that already have been there and the threads that have been spawned by the last thesis by q rh = trss fg) by (simp add: trcl-concat)
— We then distinguish whether we appended a same-level or a returning path show ?thesis proof (cases r’)
— If we appended a same-level path case (Cons v)— Same-level path with IHP(5) have CC: r’=[v] by auto
— The macrostep still ends with a same-level path with NEWPATH have (((entry fg q), ceh), wws@w, ([v'], ce'))∈trcl (trss fg) by simp
— and thus remains a valid macrostep from gtrI-s[OF ntrs-step[OF STEPFMT(4)], simplified, OF this]] have {(#uh # rh#) + ceh, LCall q # wws@w, {#v' # uh' # rh#} + ce'}∈ ntr fg .
— that we can append to the prefix of the normalized path to get our proposition with STEPFMT(2) SPLIT(1) CC C’(2) have {(#[entry fg p]#, wws@[LCall q#wws@w], (# r#rh #) + ce'})∈trcl (ntr fg) by (auto simp add: trcl-rev-cons)
thus ?thesis by blast
next
— If we appended a returning path case Nil note CC=this
— The macrostep now ends with a returning path, and thus gets a same-level path have NEWSL: {([uh], ceh), LCall q # wws @ w, [uh'], ce'}∈ trcl (trss fg)
proof
from STEPFMT(4) have (((uh], ceh), LCall q,(entry fg q#[uh'], ceh))∈trss fg by (auto elim!: trss_cases intro: trss_intros)
also from trss-stack-comp[OF NEWPATH] CC have (((entry fg q#[uh'], ceh), wws@w, ([uh'], ce'))∈trcl (trss fg) by auto
finally show ?thesis .
qed
— Hence we can apply the induction hypothesis and get the proposition from IHP(1)(OF - NEWSL] SPLIT STEPFMT(2) IHP(4) CC C’(2) show ?thesis by auto
qed
next
— If the sl/ret path was executed on a different thread than the last macrostep case (env cc) note CASE=this
— we first look at the context after the last macrostep. It consists of the threads that already have been there and the threads that have been spawned by the last macrostep from STEPFMT(5) obtain cspt where CETFMT: cet=cspt+ceh !!s. s:#cspt
⇒ ∃p. s=[entry fg p] ∧ initialproc fg p
by (unfold initialproc-def) (erule trss-cases, blast)
— The spawned threads do not hold any monitors yet hence CSPT-NO-MON: mon-c fg cspt = {} by (simp add: c-of-initial-no-mon)
— We now distinguish whether the sl/ret path is executed on a thread that was just spawned or on a thread that was already there
from CASE(1) CETFMT(1) have u≠r :# csp[t+c eh] by auto
thus ?thesis proof (cases rule: mset-un-cases[cases set])
— The sl/ret path cannot have been executed on a freshly spawned thread
due to the restrictions we made on the flowgraph
   case left — Thread was spawned with CETFMT obtain q where u=entry
   fg q r=[[ initialproc fg q by auto
with IHP(3,5,6) no-retsl-from-initial have False by blast
thus ?thesis ..
next
— Hence let’s assume the sl/ret path is executed on a thread that was already
there before the last macrostep
   case right note CC=This
— We can write the configuration before the last macrostep in a way that one
sees the thread that executed the sl/ret path
   hence CEHFMT: ceh=(# u≠r #)+(ceh-(# u≠r #)) by auto
   have CHFMT: ch = (# u≠r #) + ((# uh#rh #)+(ceh-(# u≠r #)))
   proof —
   from CEHFMT STEPFMT(2) have ch = (# uh#rh #) + ((# u≠r #)+(ceh-(# u≠r #))) by simp
   thus ?thesis by (auto simp add: union-ac)
qed
— There are not more monitors than after the last macrostep
   have MON-CE: mon-c fg ((# uh#rh #)+(ceh-(# u≠r #))) ⊆ mon-c fg
   proof —
   have mon-n fg uh ⊆ mon-n fg uh’ using STEPFMT(4) by (auto elim!: trss.cases dest: mon-n-same-proc edges-part)
   moreover have mon-c fg (ceh-(# u≠r #)) ⊆ mon-c fg cc proof —
   from CASE(3) CETFMT have cc=(csp[t+ceh]-(#u≠r #)) by simp
   with CC have cc = csp[t+(ceh-(#u≠r #))] by (auto simp add: diff-union-single-conv)
   with CSPT-NO-MON show ?thesis by (auto simp add: mon-c-unconc)
qed
ultimately show ?thesis using CASE(2) by (auto simp add: mon-c-unconc)
qed
— The same-level path preserves the threads in its environment and the threads
that it creates hold no monitors
   from IHP(3) obtain csp’ where CE’FMT: ce’=csp’+cc mon-c fg csp’ = {} by (–) (crule 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-CE] IHP(3) CE’FMT have NSL:
((#[u], {#uh # rh#} + (ceh - {#u # r#})), w, r’, csp’ + ({#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#} + (csp’ + ({#uh # rh#} + (ceh - {#u # r#})))) ∈ trcl (ntr fg) by blast
— We now show that the last macrostep can also be executed from the new
configuration, after the sl/ret path has been executed (on another thread)
   have ({#r’ @ r#} + (csp’ + ({#uh # rh#} + (ceh - {#u # r#}))), ee,
Finally we can prove: Any reachable configuration can also be reached by a normalized path. With eflowgraph.ntr-sl-move-left we can easily show this lemma. With eflowgraph.ntr-sl-move-left we can easily show this by induction on the reaching path. For the empty path, the proposition follows trivially. Else we consider the last step. If it is a call, we can execute it as a macrostep and get the proposition. Otherwise the last step is a same-level (Base, Spawn) or returning (Ret) path of length 1, and we can append it to the normalized path using eflowgraph.ntr-sl-move-left.
lemma (in eflowgraph) normalize:\[
(cstart, w, c) \in \text{trcl}(tr fg);\]
\[cstart = \{ \# [entry fg p] \};\]
\[[ \text{initialproc} fg p \]]
\[
\rightarrow \exists w'. (\{ \# [entry fg p] \}, w', c') \in \text{trcl} (ntr fg)
\]
— The lemma is shown by induction on the reaching path.

proof (induct rule: \text{trcl-rev-induct})
— The empty case is trivial, as the empty path is also a valid normalized path.

\text{case empty thus} \ ?\text{case by} (auto intro: efl[of - [] ])

next
\text{case} (\text{snoc} cstart w c e c') \ note IHP=\text{this}
— In the inductive case, we can assume that we have an already normalized path and need to append a last step.

\text{then obtain} w' \text{where} IHP' = (\{ \# [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 \text{gtr-find-thread}[OF IHP'(2)] \ obtain s ce s' ce' \text{where} TSTEP: c = \{ s, s' \}
+ ce c' = \{ s', s, 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=\text{LCall} q
— As it is the last step, the procedure call will not return and thus is a valid macrostep.

have \ (c, \text{LCall} q \ # [], c') \in \text{etr} fg \text{using} TSTEP \ CASE \ by \ (auto elim!: trss.cases intro!: ntr.s intros gtr.s trss.intros)
— That can be appended to the initial normalized path.

from \text{trcl-rev-cons}[OF IHP'(1) \ \text{this}] \ have \ ?\text{case by} \ blast
\}
moreover \{ — Last step was no procedure call

fix q a
assume CASE: \ e=\text{LBase} a \lor \ e=\text{LSpawn} q \lor \ e=\text{LRet}
— Then it is a same-level or returning path.

with \text{TSUP(3)} \ obtain \ u \ r \ r' \text{where} \ SLR: s=u\#r \ s'=r'@r \text{length} r'\leq 1
(\{(u), (r'), (ce')\} \in \text{trcl} (tr fg) \ ) \text{by} (force elim!: trss.cases intro!: ntr.s intros)
— That can be appended to the normalized path using the \{\{ \# [entry fg p] \# \}, \ w w, \{ \# ?u \ ?r \# \} + ?ce \} \in \text{trcl} (ntr fg): (\{(u), (r'), (ce')\} \in \text{trcl} (tr ss fg); \text{initialproc} fg \ ?p; \text{length} r' \leq 1 ; \ w \neq [] \} \rightarrow \ \exists w'. (\{ \# [entry fg p] \# \}, w w', \{ \# ?r' @ ?r \# \} + ?ce') \in \text{trcl} (ntr fg) \ - \text{lemma}

from \text{ntr-sl-move-left}[OF - \text{SLR}(4) \ IHP(5) \ SLR(3)] \ IHP'(1) \ TSTEP(1)
\text{SLR(1) obtain} \ \text{ww'} \text{where} (\{ \# [entry fg p] \# \}, \ w w', \{ \# ?r' @ ?r \# \} + ce') \in \text{trcl}
(ntr fg) \ \text{by auto}

with \text{SLR(2) \ TSTEP(2)} \ have \ ?\text{case by} \ auto
\}
ultimately \text{show} \ ?\text{case by} \ (\text{cases} \ e, \ \text{auto})

qed

As the main result of this section we get: \text{A configuration is reachable if and only if it is also reachable via a normalized path.}
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]:

\[
\begin{align*}
&(s, c, w, (s', c')) \in \text{ntrs fg}; \\
&\text{csp} \cdot [c' = \text{csp} + c; \text{!} s: \# \text{csp}] \Rightarrow \exists p u v. s = [\text{entry fg p}] \\
&\quad \land (u, \text{Spawn p, v}) \in \text{edges fg} \\
&\quad \land \text{initialproc fg p}
\end{align*}
\]

\[ \models P \]
by (auto dest!: ntrs-is-trss-s elim!: trss-c-cases)

**lemma** ntrs-c-cases[cases set]:

\[
\begin{align*}
&(s, c, w, (s', c')) \in \text{trcl (ntrs fg)}; \\
&\text{csp} \cdot [c' = \text{csp} + c; \text{!} s: \# \text{csp}] \Rightarrow \exists p u v. s = [\text{entry fg p}] \\
&\quad \land (u, \text{Spawn p, v}) \in \text{edges fg} \\
&\quad \land \text{initialproc fg p}
\end{align*}
\]

\[ \models P \]
by (auto dest!: ntrs-is-trss elim!: trss-c-cases)

8.4.1 Validity

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

**lemmas** (in flowgraph) ntrs-valid-preserve-s = trss-valid-preserve[OF ntrs-is-trss-s]

**lemmas** (in flowgraph) ntrs-valid-preserve-s = tr-valid-preserve[OF ntrs-is-tr-s]

**lemmas** (in flowgraph) ntrs-valid-preserve = tr-valid-preserve[OF ntrs-is-tr]

**lemma** (in flowgraph) ntrp-valid-preserve-s:

assumes A: ((s, c), (s', c')) \in ntrp fg

and V: valid fg (\{\#s\#\} + c)

shows valid fg (\{\#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), (s', c')) \in trcl (ntrp fg)

and V: valid fg (\{\#s\#\} + c)

shows valid fg (\{\#s'\#\} + c)

using ntr-valid-preserve[OF gtrp2gtr[OF A] V] by assumption
8.4.2 Monitors

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

**definition**

\[ \text{mon-ww } fg \; ww \equiv \text{foldl} \ (\text{op} \cup) \ \{\} \ (\text{map} \ (\text{mon-w } fg) \; ww) \]

**definition**

\[ \text{mon-loc } fg \; ww \equiv \text{mon-ww } fg \ (\text{map} \ \text{le-rem-s} \ (\text{loc} \; ww)) \]

**definition**

\[ \text{mon-env } fg \; ww \equiv \text{mon-ww } fg \ (\text{map} \ \text{le-rem-s} \ (\text{env} \; ww)) \]

**lemma** \text{mon-ww-empty}[simp]: \text{mon-ww } fg \; [] = \{\}

\text{by (unfold mon-ww-def, auto)}

**lemma** \text{mon-ww-uncons}[simp]:

\[ \text{mon-ww } fg \ (ee \# \; ww) = \text{mon-w } fg \; ee \cup \text{mon-ww } fg \; ww \]

\text{by (unfold mon-ww-def, auto simp add: foldl-un-empty-eq[of mon-w } fg \; ee\)]

**lemma** \text{mon-ww-unconc}:

\[ \text{mon-ww } fg \ (ww1 \@ \; ww2) = \text{mon-ww } fg \; ww1 \cup \text{mon-ww } fg \; ww2 \]

\text{by (induct \; ww1) auto}

**lemma** \text{mon-env-empty}[simp]: \text{mon-env } fg \; [] = \{\}

\text{by (unfold mon-env-def) auto}

**lemma** \text{mon-env-single}[simp]:

\[ \text{mon-env } fg \ [e] = (\text{case } e \text{ of } \text{LOC } a \Rightarrow \{\} \mid \text{ENV } a \Rightarrow \text{mon-w } fg \; a) \]

\text{by (unfold mon-env-def) (auto split: el-step.split)}

**lemma** \text{mon-env-uncons}[simp]:

\[ \text{mon-env } fg \ (e \# \; w) = (\text{case } e \text{ of } \text{LOC } a \Rightarrow \{\} \mid \text{ENV } a \Rightarrow \text{mon-w } fg \; a) \cup \text{mon-env } fg \; w \]

\text{by (unfold mon-env-def) (auto split: el-step.split)}

**lemma** \text{mon-env-unconc}:

\[ \text{mon-env } fg \ (w1 \@ \; w2) = \text{mon-env } fg \; w1 \cup \text{mon-env } fg \; w2 \]

\text{by (unfold mon-env-def) (auto simp add: mon-ww-unconc)}

**lemma** \text{mon-loc-empty}[simp]: \text{mon-loc } fg \; [] = \{\}

\text{by (unfold mon-loc-def) auto}

**lemma** \text{mon-loc-single}[simp]:

\[ \text{mon-loc } [e] = (\text{case } e \text{ of } \text{ENV } a \Rightarrow \{\} \mid \text{LOC } a \Rightarrow \text{mon-w } fg \; a) \]

\text{by (unfold mon-loc-def) (auto split: el-step.split)}

**lemma** \text{mon-loc-uncons}[simp]:

\[ \text{mon-loc } (e \# \; w) = (\text{case } e \text{ of } \text{ENV } a \Rightarrow \{\} \mid \text{LOC } a \Rightarrow \text{mon-w } fg \; a) \cup \text{mon-loc } fg \; w \]

\text{by (unfold mon-loc-def) (auto split: el-step.split)}

**lemma** \text{mon-loc-unconc}:

\[ \text{mon-loc } (w1 \@ \; w2) = \text{mon-loc } fg \; w1 \cup \text{mon-loc } fg \; w2 \]

\text{by (unfold mon-loc-def) (auto simp add: mon-ww-unconc)}

61
lemma mon-ww-of-foldl[simp]: mon-w fg (foldl (op @) [] ww) = mon-ww fg ww
apply (induct ww)
apply (unfold mon-ww-def)
apply simp
apply simp
apply (subst foldl-conc-empty-eq, subst foldl-un-empty-eq)
apply (simp add: mon-w-unconc)
done

lemma mon-ww-ileq: \( w \leq w' \implies \text{mon-ww fg } w \subseteq \text{mon-ww fg } w' \)
by (induct rule: less-eq-list-induct) auto

lemma mon-ww-cil:
\( \forall w1, w2. w \in w1 \otimes \alpha w2 \implies \text{mon-ww fg } w = \text{mon-ww fg } w1 \cup \text{mon-ww fg } w2 \)
by (induct rule: cil-set-induct-fix \( \alpha \)) auto

lemma mon-env-cil:
\( \forall w1, w2. w \in w1 \otimes \alpha w2 \implies \text{mon-env fg } w = \text{mon-env fg } w1 \cup \text{mon-env fg } w2 \)
by (induct rule: cil-set-induct-fix \( \alpha \)) auto

— As monitors are syntactically bound to procedures, and each macrostep starts with a non-returning call, the set of monitors allocated during the execution of a normalized path is monotonically increasing

lemma (in flowgraph) ntrs-mon-increasing-s: 
\( ((s,c), e, (s',c')) \in ntrs fg \implies \text{mon-s fg } s \subseteq \text{mon-s fg } s' \land \text{mon-c fg } c = \text{mon-c fg } c' \)
apply (erule ntrs.cases)
apply (auto simp add: trss-c-no-mon)
apply (subgoal-tac mon-n fg u = mon-n fg w')

—
apply (simp)
apply (auto elim!: trss_cases dest!: mon-u-same-proc edges-part)
done

lemma (in flowgraph) ntr-mon-increasing-s:
  \((e, ee, e') \in \text{ntr} f g \implies \text{mon-c} f g c \subseteq \text{mon-c} f g e'\)
by (erule gtrE) (auto dest: ntrs-mon-increasing-s simp add: mon-c-unconc)

lemma (in flowgraph) ntrp-mon-increasing-s:
  \((s, c, e, (s', c')) \in \text{ntrp} f g \implies \text{mon-s} f g s \subseteq \text{mon-s} f g s' \land \text{mon-c} f g c \subseteq \text{mon-c} f g e'\)
apply (erule gtrp_cases) apply (auto dest: ntrs-mon-increasing-s simp add: mon-c-unconc) apply (erule ntrs-c-cases-s)
apply auto

proof -
fix \(c, c', s', x, e\)
assume \(\{\#s\} + c' = e\) by (simp add: union-ac)
with union-left-cancel[of \(\{\#s\} + c\)] have \(c' = e\) by (simp add: union-ac)
moreover assume \(x \in \text{mon-c} f g c = {}\) by (simp add: union-ac)
ultimately have False by (auto simp add: mon-c-unconc)
thus \(x \in \text{mon-s} f g s'\).
qed

lemma (in flowgraph) ntrp-mon-increasing:
  \((s, c, e, (s', c')) \in \text{trcl} (\text{ntrp} f g) \implies \text{mon-s} f g s \subseteq \text{mon-s} f g s' \land \text{mon-c} f g c \subseteq \text{mon-c} f g e'\)
by (induct rule: trcl-rev-pair-induct) (auto dest!: ntrp-mon-increasing-s)

8.4.3 Modifying the context

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

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

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

lemma (in flowgraph) ntrp-mon-env-e-no-ctx:
  \((s, c, \text{ENV} e, (s', c')) \in \text{ntrp} f g \implies \text{mon-w} f g e \cap \text{mon-s} f g s = {}\)
by (auto elim!: gtrp_cases dest!: ntrs-mon-e-no-ctx simp add: mon-c-unconc)
lemma (in flowgraph) ntrp-mon-loc-e-no-ctx:
  \((s, c, \text{LOC} e, (s', c')) \in \text{ntrp} f g \implies \text{mon-w} f g e \cap \text{mon-c} f g c = {}\)
by (auto elim!: gtrp_cases dest!: ntrs-mon-e-no-ctx)
lemma (in flowgraph) ntrp-mon-env-w-no-ctx:

\[(s,c),w,(s',c')\in\text{trcl (ntrp fg)} \implies \text{mon-env fg w } \cap \text{ mon-c fg s } = \{\}\]


lemma (in flowgraph) ntrp-mon-loc-w-no-ctx:

\[(s,c),w,(s',c')\in\text{trcl (ntrp fg)} \implies \text{mon-loc fg w } \cap \text{ mon-c fg c } = \{\}\]


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

lemma (in flowgraph) ntrs-modify-context-s:

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

and: \(B: \text{mon-w fg ee } \cap \text{ mon-c fg cn } = \{\}\)

shows: \(\exists \text{ c' } \in \text{csp } + \text{ mon-c fg csp } = \{\}\ \land \ ((s,cn),(s',csp+cn))\in\text{ntrs fg}\)

proof –

from \(A\) obtain \(p\ \text{r } u\ \text{v w}\) where \(S: s=u\#r\ \text{ee=LCall p}\#w\ s'=v\#u\#r\ ((u\#r,c),\text{LCall p),(entry fg p}\#u\#r,c))\in\text{trss fg} (((\text{entry fg p}),c),w,(\text{v},c'))\in\text{trcl (trss fg)}\) by (blast elim!: ntrs.cases[simplified])

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

moreover from \(S\) \text{trss-modify-context[OF S(5)]} \(B\) obtain \(csp\) where \(c' = csp + c\)

mon-c fg csp = \{\}\ \land \ ((\text{entry fg p},cn),w,(\text{v},csp+cn))\in\text{trcl (trss fg)}\) by auto

ultimately show \(?\text{thesis using } S\) by (auto intro!: ntrs-step)

qed

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

\[\text{[(s,c),w,(s',c')}\in\text{trcl (ntrp fg)}] \implies \forall cn. \text{mon-ww fg w } \cap \text{ mon-c fg cn } = \{\}\]

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

proof (induct rule: trcl-pair-induct)

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

next
case (cons s c e sh ch w s' c')

note \(\text{IHP=this show } ?\text{case}\)

proof (intro allI simp)

fix cn

assume \(\text{MON: mon-ww fg (c } \# \text{ w) } \cap \text{ mon-c fg cn } = \{\}\)

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

with \(\text{MON have mon-ww fg w } \cap \text{ mon-c fg (csph+cn) } = \{\}\\) by (auto simp add: mon-c-unconc)

with \(\text{IHP(3)[rule-format] obtain csp where } S2: c' = csp + c\ \text{mon-c fg csp } = \{\}\)

\(((sh,\text{csph+cn}),w,(s',csp+(\text{csph+cn})))\in\text{trcl (ntrp fg)}\) by blast

from \(S1\) \(S2\) have \(c' = (csp + csph) + c\ \text{mon-c fg (csp+csph)} = \{\}\ ((s, cn), e\#w,(s',(csp+csph)+cn))\in\text{trcl (ntrp fg)}\) by (auto simp add: union-assoc mon-c-unconc)

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

64
\begin{align*}
&\text{qed}
\end{align*}

**Lemma** \textit{ntrs-xchange-context-s}:
\begin{align*}
&\text{assumes } A: ((s,c),ee,(s',csp+c)) \in ntrs fg \\
&\text{and } B: \text{mon-c fg cn} \subseteq \text{mon-c fg c} \\
&\text{shows } ((s,\text{cn}),ee,(s',csp+cn)) \in ntrs fg
\end{align*}

**Proof**

\begin{align*}
&\text{obtain } \ p \ r \ u \ u' \ v \ w \ \text{where } S: s = u \# r \ ee = LCall p \# s' = v \# u' \# r \ ((u \# r, c), LCall p, (\text{entry fg p} \# u' \# r, c)) \in \text{trss fg} \\
&\text{by } \text{F (lemma ntrs-replace-context-s) OF S} \\
&\text{proof}:
\end{align*}

\begin{itemize}
\item **Case** \text{goal1} \text{moreover}
\begin{align*}
&\text{from } \text{ntrs-cases[OF A, simplified]} \ \text{obtain } ce \ c' \ p \ r \ u \ u' \ v \ w \ \text{where } s = u \# r \ c = ce \ ee = LCall p \# w \ s' = v \# u' \# r \ ce' \ ((u \# r, ce), LCall p, \text{entry fg p} \# u' \# r, ce) \in \text{trss fg} \\
&\text{by auto}
\end{align*}
\item **Hence** \text{s = u \# r ee = LCall p \# w s' = v \# u' \# r ((u \# r, c) = ce', LCall p, (\text{entry fg p} \# u' \# r, c)) \in \text{trss fg} by auto}
\item **Ultimately** \text{show } \text{thesis by simp}
\end{itemize}

**Lemma** \textit{ntrs-replace-context-s}:
\begin{align*}
&\text{assumes } A: ((s,c+cr),ee,(s',c'+cr)) \in ntrs fg \\
&\text{and } B: \text{mon-c fg crn} \subseteq \text{mon-c fg cr} \\
&\text{shows } ((s,\text{c+cr}),ee,(s',c'+crn)) \in ntrs fg
\end{align*}

**Proof**

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

**Lemma** \textit{(in flowgraph) ntrs-xchange-context}:
\begin{align*}
&\text{Assume } s \ c \ c' \ \text{cn}. \ [ \\
&(\text{in}\ (s,d),\text{ww},(s',c')) \in \text{trcl (ntrs fg)}; \\
&\text{mon-c fg cn} \subseteq \text{mon-c fg c} \\
&\Rightarrow \exists \ csp. \\
&c' = csp + c \wedge ((s,\text{cn}),\text{ww},(s',csp+cn)) \in \text{trcl (ntrs fg)}
\end{align*}

**Proof** \text{(induct \text{ww})}

\begin{itemize}
\item **Case** \text{Nil} \text{note CASE=this}
\item **Thus** \text{?case by (auto intro!: exI[of - {#}]})
\item **Next**
\item **Case** \text{(Cons ee \text{ww})} \text{note \text{IHP=this} }
\item **Then obtain** \text{sh \ ch where SPLIT: ((s,c),ee,(s,sh,\text{ch})) \in ntrs fg ((sh,\text{ch}),\text{ww}(s',c')) \in \text{trcl}}
\end{itemize}
(ntrs fg) by (fast dest; trcl-uncons)
from ntrs-cases-s[OF SPLIT(1)] obtain csph where CHFMT: ch=csph+c
!!s. s:#csph \rightarrow \exists p u v. s=[entry fg p] \land (u, Spawn p, v) \in edges fg \land initialproc fg p by blast
with ntrs-xchange-context-s SPLIT(1) IHP(3) have ((s, cn), ee, (sh, csph+cn))\in ntrs fg by blast
also
from c-of-initial-no-mon CHFMT(2) have CSPH-NO-MON: mon-c fg csph = {} by auto
with IHP(3) CHFMT have 1: mon-c fg (csph+cn) \subseteq mon-c fg ch by (auto simp add: mon-c-unconc)
from IHP(1)[OF SPLIT(2) this] obtain csp where C’FMT: c’=csp+ch and SND: (sh,csph+cn),ww,(s’,csp+(csph+cn)))\in trcl (ntrs fg) by blast
note SND
finally have ((s, cn), ee \# ww, s’, (csp + csph) + cn) \in trcl (ntrs fg) by (simp add: union-assoc)
moreover from CHFMT(1) C’FMT have c’=(csp+csph)+c by (simp add: union-assoc)
ultimately show ?thesis by blast
qed

lemma (in flowgraph) ntrs-replace-context:
  assumes A: ((s,c+cr),ww,(s’,c’+cr))\in trcl (ntrs fg)
  and B: mon-c fg crn \subseteq mon-c fg cr
  shows ((s,c+crn),ww,(s’,c’+crn))\in trcl (ntrs fg)
proof –
from ntrs-cases[OF A] obtain csp where G: c’+cr = csp+(c+cr) . hence F: c’=csp+c by (auto simp add: union-assoc[symmetric])
from B have MON: mon-c fg (c+crn) \subseteq mon-c fg (c+cr) by (auto simp add: mon-c-unconc)
from ntrs-xchange-context[OF A MON] G have ((s,c+crn),ww,(s’,csp+(c+crn)))\in trcl (ntrs fg) by auto
with F show ?thesis by (simp add: union-assoc)
qed

lemma (in flowgraph) ntr-add-context-s:
  assumes A: (c,e,c’)\in ntr fg
  and B: mon-w fg e \cap mon-c fg cn = {}
  shows (c+cn,e,c’+cn)\in ntr fg
proof –
from gtrE[OF A] obtain s ce s’ ce’ where NTRS: c = {#s#} + ce c’ = {#s’#} + ce’ ((s, ce), e, s’, ce’) \in ntrs fg .
from ntrs-mon-e-no-ctx[OF NTRS(3)] B have M: mon-w fg e \cap (mon-c fg (ce+cn)) = {} by (auto simp add: mon-c-unconc)
from ntrs-modify-context-s[OF NTRS(3) M] have ((s,ce+cn),e,(s’,ce’+cn))\in ntrs fg by (auto simp add: union-assoc)
with NTRS show ?thesis by (auto simp add: union-assoc intro: gtrI-s)
qed
lemma (in flowgraph) ntr-add-context:
\[(c,w,c') \in \text{ntr cl (} ntr fg) \cap \text{mon-ww} \to w \cap \text{mon-c} \to fg \cap cn = \{\})\]
\[\Rightarrow (c+c\cap,w,c'+c\cap) \in \text{ntr cl (} ntr fg)\]
by (induct rule: trcl.induct) (simp, force dest: ntr-add-context-s)

lemma (in flowgraph) ntrp-add-context-s:
assumes A: \[((s,c),e,(s',c')) \in \text{ntrp fg} \cap \text{mon-ww} \cap e \cap \text{mon-c} \cap fg \cap cn = \{\}\]
and B: \[((s,c+cn),e,(s',c'+cn)) \in \text{ntr fg} \cap \text{trcl-pair-induct}\]
shows \([(s,c+cn),e,(s',c'+cn)) \in \text{ntrp fg} \cap \text{trcl-pair-induct}\]
(force simp add: mon-e-unconc union-ac)

lemma (in flowgraph) ntrp-add-context-s:
\[\[(s,c),e,(s',c') \in \text{ntrp fg} \cap \text{mon-ww} \cap e \cap \text{mon-c} \cap fg \cap cn = \{\}\]\n\[\Rightarrow ((s,c+cn),e,(s',c'+cn)) \in \text{ntrp fg} \cap \text{trcl-pair-induct}\]
apply (erule gtrp.cases)
apply (auto dest: ntrp-add-context-s intro!: gtrp.intros)
apply (simp only: union-assoc)
apply (rule gtrp-env)
apply (simp only: union-assoc[symmetric])
apply (rule ntrp-add-context-s)
apply assumption+
done

lemma (in flowgraph) ntrp-add-context:
\[((s,c),w,(s',c')) \in \text{trcl (} ntrp fg)\]
\[\Rightarrow ((s,c+cn),w,(s',c'+cn)) \in \text{trcl (} ntrp fg)\]
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: \[((s,c),ee,(s',c')) \in \text{ntrs fg}\]
shows \[((s@r,c),ee,(s'@r,c')) \in \text{ntrs fg}\]
using A
by (auto dest: trss-stack-comp trss-stack-comp-s elim!: ntrs.cases intro!: ntrs-step[simplified])

lemma ntrs-stack-comp: \[((s,c),ww,(s',c')) \in \text{trcl (} ntrs fg)\]
\[\Rightarrow ((s@r,c),ww,(s'@r,c')) \in \text{trcl (} ntrs fg)\]
by (induct rule: trcl-pair-induct) (auto intro!: trcl.cons[OF ntrs-stack-comp-s])

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

lemma (in flowgraph) ntrsp-stack-comp:
( ((s,c),ww,(s’,c’))∈trcl (ntrsp fg); mon-s fg r \cap mon-env fg ww = {} )

by (induct rule: trcl-pair-induct) (auto intro!: trcl.cons[OF ntrsp-stack-comp-s])

lemma ntrsp-stack-top-decomp-s:
  assumes A: ((u#r,c),ee,(s’,c’))∈ntrs fg
and EX: !!v u’ v p. [ s’=v#u’#r; (([u],c),ee,([v,u’],c’))∈ntrs fg; (u,Call p,u’)∈edges fg ]

by auto

using A

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

lemma ntrsp-stack-decomp-s:
  assumes A: ((u#s@r,c),ee,(s’,c’))∈ntrs fg
and EX: !!v u’ v p. [ s’=v#u’#r; ((u#s@r,c),ee,(s’@r,c’))∈ntrs fg
by auto

using A

proof (cases rule: ntrs.cases)
  case ntrs-step then obtain u’ v p w where CASE: ec=LCall p#w s’=v#u’#r
((u#r,c),LCall p,(entry fg p#u’#r,c))∈trss fg (([entry fg p],c),w,([v],c’))∈trcl (trss fg)
by (simp)
from trss-stack-decomp-s[where s=[u], simplified, OF CASE(3)] have SDC: 
(([u],c),LCall p, (entry fg p, u’, c)) ∈ trss fg by auto
with CASE(1,4) have (([u],c),ee,([v,u’],c’))∈ntrs fg by (auto elim!: ntrs.ntrs-step)
moreover from SDC have (u,Call p,u’)∈edges fg by (auto elim!: trss.cases)
ultimately show ?thesis using CASE(2) by (blast intro!: EX)
qed
\[s' = v\# u'\# s@r; \]
\[((u\#s,c), ee, (v\#u'\# s, c'))\in ntrs fg; \]
\((u, Call p, u')\in edges fg\]
\]
\[\implies P\]
shows \(P\)
\apply (rule ntrs-stack-top-decomp-s[OF \(A\)])
\apply (rule EX)
\apply (auto dest: ntrs-stack-comp-s)
done


\textbf{lemma} ntrs-stack-decomp: \(!u \ s \ r \ c. \big[
\]
\((u\#s@r,c), ee, (s',c')\)\in trcl (ntrs fg); \]
\(!v \ rr. \ [s'=v\#rr@r; ((u\#s,c), ee, (v\#rr, c'))\in trcl (ntrs fg)] \implies P\]
\]
\[\implies P\]
proof (induct \(ww\))
\textbf{case} Nil thus \(\textbf{case by fastforce}\)
\textbf{next}
\textbf{case} (\(Cons \ e \ w\)) from Cons.prems show \textbf{case proof} (cases rule: trcl-pair-unconsE)
\textbf{case} (split \(sh \ ch\))
from ntrs-stack-decomp-s[OF split(1)] obtain \(vh \ uh \ p\) where \(F\): \(sh = vh\#uh\#s@r\)
\((u\#s, c), e, vh\#uh\#s, ch) \in ntrs fg (u, Call p, uh) \in edges fg by blast
from \(P(1)\) split(2) Cons.prems of \(vh\#uh\#s\) \(r\) \(ch\) obtain \(v' \ rr\) where \(S\):
\((v\#rr@r, ((vh\#uh\#s, ch), w, (v'\#rr, c'))\in trcl (ntrs fg)\) by auto
\from trcl.cons[OF \(F(2)\)] \(S(2)\) Cons.prems show \textbf{thesis by blast}
qed

\textbf{lemma} ntrp-stack-decomp-s:
\textbf{assumes} \(A\): \((u\#s@r,c), ee, (s',c')\)\in ntrp fg
\textbf{and} \(EX\): \(!v \ rr. \ [s'=v\#rr@r; ((u\#s,c), ee, (v\#rr, c'))\in ntrp fg ] \implies P\]
\textbf{shows} \(P\)
\textbf{using} \(A\)
\textbf{proof} (cases rule: gtrp.cases)
\textbf{case} gtrp-env thus \textbf{thesis using EX} by (force elim!: ntrp-stack-decomp-s intro!: gtrp.intro)
\textbf{next}
\textbf{case} gtrp-env then obtain \(e ss \ ss' ce ce'\) where \(S:\ vce=ENV\ e\ s'=u\#s@r\)
\(c=\{\#ss\}+ce\ c'=\{\#ss'\}+ce'(\{ss,ce+\{\#u\#s@r\}\},e,\{ss',ce'+\{\#u\#s@r\}\})\in ntrs fg\) by (auto simp add: union-ac)
\from ntrs-replace-context-s[OF \(S(5)\), where \(crn=\{\#u\#s\}\)] have \((ss, \{\#u \# s\} + ce), ss', \{\#u \# s\} + ce'\) \in ntrs fg by (auto simp add: mon-s-unconc union-ac)
\with \(S\) show \(P\) by (rule-tac EX) (auto intro: gtrp.gtrp-env)
qed

\textbf{lemma} ntrp-stack-decomp: \(!u \ s \ r \ c. \big[
\]
\((u\#s@r,c), ee, (s',c')\)\in trcl (ntrp fg); \]
\(!v \ rr. \ [s'=v\#rr@r; ((u\#s,c), ee, (v\#rr, c'))\in trcl (ntrp fg)] \implies P\]
\]
\[\implies P\]
proof (induct \( \text{ww} \))
    case Nil thus \( \text{case} \) by fastforce
next
    case (Cons \( e \) \( w \)) from Cons.prems show \( \text{case} \) proof (cases rule: trcl-pair-unconsE)
        case (split \( sh \) \( ch \))
            from ntrp-stack-decomp-s[OF split(1)] obtain \( vh \) \( rrh \) where \( F : sh = vh \# rrh \) \( @ \) \( r \) by blast
            from \( F(1) \) split(2) Cons.hyps[of vh rrh r ch] obtain \( v' \) \( rr \) where \( S : s' = v' \# rr \) \( @ \) \( r \) \( ((vh\# rrh, ch), (v'\# rr, c')) \) \in \( \text{trcl} \) \( (\text{ntrp f g}) \) by auto
            from trcl.cons[OF \( F(2) \) \( S(2) \)] \( S(1) \) Cons.prems(2) show \( \text{thesis} \) by blast
        qed
    qed

8.5 Relation to monitor consistent interleaving

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

8.5.1 Abstraction function for normalized paths

We first need to define an abstraction function that maps a macrostep on a pair of entered and passed monitors, as required by the \( \bigotimes_{\alpha} \)-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 \( \alpha_n \) \( fg e \) \( == \)
    if \( e = [] \) then (\{\},\{\}) else (mon-e \( fg e \) \( (\text{hd} \) \( e \)), mon-w \( fg w \) \( (\text{tl} \) \( e)\))

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

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

definition \( \alpha_n \) \( l e \) \( e \) \( == \)
    \( \alpha_n \) \( fg (\text{le-rem-s} \) \( e \))

lemma \( \alpha_n\)-def': \( \alpha_n \) \( fg == \alpha_n \) \( fg \circ \text{le-rem-s} \)
    by (rule eq-reflection[OF ext]) (auto simp add: \( \alpha_n\)-def)

— These are some ad-hoc simplifications, with the aim at converting \( \alpha_n \) back to \( \alpha_n \)

lemma \( \alpha_n\)-simps[simp]:
    \( \alpha_n \) \( fg (\text{ENV} \) \( x) = \alpha_n \) \( fg x \)
    \( \alpha_n \) \( fg (\text{LOC} \) \( x) = \alpha_n \) \( fg x \)
    by (unfold \( \alpha_n\)-def, auto)

lemma \( \alpha_n\)-simps1[simp]:
    \( (\alpha_n \) \( fg) \circ \text{ENV} = \alpha_n \) \( fg \)

70
\((\alpha nl \circ \text{LOC}) = \alpha fn\)
by (unfold \(\alpha nl\)-def’ comp-def) (simp-all)

**Lemma \(\alpha nl\)-cons**: \((\alpha fn \circ \text{le-rem-s}) = \alpha nl\)

unfolding \(\alpha nl\)-def [symmetric]

**Lemma \(\alpha nl\)-fst-snd**: \(\text{fst} (\alpha fn w) \cup \text{snd} (\alpha fn w) = \text{mon-w fn w}\)
by (induct w) auto

**Lemma mon-pl-of-\(\alpha nl\)**:

\[
\text{mon-pl} (\text{map} (\alpha nl fn) w) = \text{mon-loc fn w} \cup \text{mon-env fn w}
\]
by (induct w) (auto split: el-step, split)

We now derive specialized introduction lemmas for \(\otimes_{\alpha fn}\)

**Lemma cil-\(\alpha nl\)-cons-helper**: \(\text{mon-pl} (\text{map} (\alpha nl fn) wb) = \text{mon-ww fn wb}\)

apply (unfold mon-pl-def)
apply (induct wb)
apply simp-all
apply (unfold mon-ww-def)
apply (subst foldl-an-empty-eq)
apply (case-tac a)
apply simp-all
done

**Lemma cil-\(\alpha nl\)-cons-helper**:
\(\text{mon-pl} (\text{map} (\alpha nl fn) wb) = \text{mon-ww fn wb}\)
by (simp add: \(\alpha nl\)-cil-\(\alpha nl\)-cons-helper [symmetric])

**Lemma cil-\(\alpha nl\)-cons1**:
\[\forall w \in \mathcal{W} \otimes_{\alpha fn} \otimes_{\alpha fn} \text{wb}; \text{fst} (\alpha fn e) \cap \text{mon-ww fn wb} = \{\}\]
\[\Rightarrow e \# w \in e \# \mathcal{W} \otimes_{\alpha fn} \text{wb}\]
apply (rule cil-cons1)
apply assumption
apply (subst cil-\(\alpha nl\)-cons-helper)
apply assumption
done

**Lemma cil-\(\alpha nl\)-cons2**:
\[\forall w \in \mathcal{W} \otimes_{\alpha fn} \otimes_{\alpha fn} \text{wb}; \text{fst} (\alpha fn e) \cap \text{mon-ww fn wa} = \{\}\]
\[\Rightarrow e \# w \in \mathcal{W} \otimes_{\alpha fn} e \# \text{wb}\]
apply (rule cil-cons2)
apply assumption
apply (subst cil-\(\alpha nl\)-cons-helper)
apply assumption
done

### 8.5.2 Monitors

**Lemma (in flowgraph) ntrs-mon-s**:

assumes \(A : ((s,c),e,(s’,c’)) \in \text{ntrs fg}\)
shows \(\text{mon-s fg s’} = \text{mon-s fg s} \cup \text{fst} (\alpha fn e)\)

proof –
from \(A\) obtain \(u r p u’ w v\) where \(\text{DET}: s = u \# r e = LCall p \# w (u \# r, c). LCall p, (\text{entry fn p } u \# r, c) \in \text{trss fg} ((\text{entry fn p}[c], w, ([v], c’)) \in \text{trcl} (\text{trss fg}) s’ = v \# u’ \# r\)
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 an-def)

qed

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

lemma (in flowgraph) ntrp-mon-s:
assumes A: ((s,c),(s',c'))\in ntrp fg
shows mon-c fg (\{#,\}+c') = mon-c fg (\{#,\}+c) \cup fst (anl fg e)
using ntrp-mon-s[OF gtrp2gtr-s[OF A]] by (unfold anl-def)

8.5.3 Interleaving theorem

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

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

lemma (in flowgraph) ntr-split:
\forall ca. cb. [(ca+cb, w, c')\in trcl (ntr fg); valid fg (ca+cb)] \implies
\exists ca' cb' wa wb.
\quad c'=ca'+cb' \land
\quad w\in (wa\oplus an fg wb) \land
\quad mon-c fg ca \cap (mon-c fg cb \cup mon-ww fg wb) = \{\} \land
\quad mon-c fg cb \cap (mon-c fg ca \cup mon-ww fg wa) = \{\} \land
\quad (ca,wa,ca')\in trcl (ntr fg) \land (cb,wb,cb')\in trcl (ntr fg)

proof (induct w) — The proof is done by induction on the path
— If the path is empty, the lemma is trivial
— \textbf{case Nil} thus ?case by — (rule exI[of - ca], rule exI[of - cb], intro exI[of - []], auto simp add: valid-unconc)

next
— We split a non-empty paths after the first (macro) step
then obtain \( ch \) where \( SPLIT: (ca+ch,e,ch) \in ntr fg \) \((ch,w,c') \in trcl \) \((ntr fg)\) by
(fast dest; trcl-uncons)
— Pick the stack that made the first step

from \( \text{gtrE}[\text{OF} \ SPLIT(1)] \) obtain \( s \) \( ce \) \( sh \) \( ceh \) where \( \text{NTRS}: ca+cb=\{\#s\}+ce \)
\((ch=\{\#sh\}+ceh \((s,ce),e,(sh,ceh)\)) \in ntr \fg \).
— And separate the threads that where spawned during the first step from the ones that where already there

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

— Needed later: The first macrostep uses no monitors already owned by threads
from \( \text{ntrs-mon-c-no-ctx}[\text{OF} \ \text{NTRS}(3)] \) have \( \text{MONED}: \text{mon-w} \ fg \ e \cap \text{mon-c} \ fg \)
\( ce=\{\} \) by (auto simp add: \( \text{mon-c-unconc} \))
— Needed later: The intermediate configuration is valid
from \( \text{ntr-valid-preserve-s}[\text{OF} \ SPLIT(1)] \) \( \text{IHP}(3) \) have \( \text{CHVALID}: \text{valid} \ fg \ ch \).

— 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 \( \text{NTRS}(1)[\text{symmetric}] \) show \( ?\text{case} \) proof (cases rule: \( \text{mset-unplusm-dist-cases} \))

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

\begin{itemize}
  \item \textbf{case left note} \( \text{CASE=this} \)
  \begin{itemize}
    \item We can write the intermediate configuration so that it is suited for
    the induction hypothesis
    \item with \( \text{CEHFMT} \) \( \text{NTRS} \) have \( \text{CHFMT}: ch=\{(\#sh\}+csp+(ca-\{\#s\})\}+cb \)
    by (simp add: union-ac)
    \item and by the induction hypothesis, we split the path from the intermediate
    configuration
    \item with \( \text{IHP}(1) \) \( \text{SPLIT}(2) \) \( \text{CHVALID} \) obtain \( ca' \) \( cb' \) \( wa \) \( wb \) where \( \text{IHAPP}: \)
    \[(ca+cb+cb') \in\]
    \[(w \in w \cap \cap ntr \fg \ wb) \in\]
    \[\text{mon-c} \ fg \ \{(\#sh\}+csp+(ca-\{\#s\})\} \cap \ (\text{mon-c} \ cb \cup \text{mon-ww} \ fg \ wb)\in\]
    \[(mon-c \ fg \ cb \cup \text{mon-ww} \ fg \ wa)\in\]
    \[(\{(\#sh\}+csp+(ca-\{\#s\}),wa,ca') \in trcl \)]
    \[\text{trcl} \ (ntr \ fg) \]
    \[(cb,wb,cb') \in\]
    \[\text{trcl} \ (ntr \ fg) \]
    \[\text{by blast}\]
    \item moreover
    \end{itemize}
  \end{itemize}
— It remains to show that we can execute the first step with the right part of
the configuration removed
have \( \text{FIRSTSTEP}: (ca,e,\{\#sh\}+csp+(ca-\{\#s\})) \in ntr \ fg \)
proof
— from \( \text{CASE}(2) \) have \( \text{mon-c} \ fg \ (ca-\{\#s\}) \subseteq \text{mon-c} \ fg \ ce \) by (auto simp
add: \( \text{mon-c-unconc} \))
with \( \text{ntrs-xchange-context-s} \) \( \text{NTRS}(3) \) \( \text{CEHFMT} \) \( \text{CASE}(2) \) have \( ((s,ca-\{\#s\}),e,(sh,csp+(ca-\{\#s\}))) \)
\( fg \) by blast
from \( \text{gtrI-s}[\text{OF this}] \) \( \text{CASE}(1) \) show \( ?\text{thesis} \) by (auto simp add: union-assoc)
qed

with IHAPP(5) have (ca,e#wa,ca')∈trcl (ntr fg) by simp

moreover
— and that we can prepend the first step to the interleaving
have e#w ∈ e#wu ⊗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-cons1[OF IHAPP(2) this] show ?thesis.
qed

moreover
— and that the monitors of the initial context does not interfere
have mon-c fg ca ∩ (mon-c fg cb ∪ mon-ww fg wb) = {} mon-c fg cb ∩ (mon-c fg ca ∪ mon-ww fg (e#wa)) = {}.

proof —
  from ntr-mon-increasing-s[OF FIRSTSTEP] IHAPP(3) show mon-c fg ca ∩ (mon-c fg cb ∪ mon-ww fg wb) = {} by auto
  from MONED CASE have mon-c fg cb ∩ mon-w fg e = {} by (auto simp add: mon-c-unconc)
  with ntr-mon-increasing-s[OF FIRSTSTEP] IHAPP(4) show mon-c fg cb ∩ (mon-c fg ca ∪ mon-ww fg (e#wa)) = {} by auto

qed

ultimately show ?thesis by blast

next
— The other case, that is if the first step was made on a thread in the right part of the configuration, is shown completely analogously

  case right note CASE=this
  with CEHFMT NTRS have CHFMT: ch=ca+(#sh#)+csp+(cb-(#s#))
  by (simp add: union-ac)
  with IHP(1) SPLIT(2) CHVALID obtain ca' cb' wa wb where IHAPP:
  c'=ca'+cb' w∈wa⊗αn fgwb 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) (#{s#}+csp+(cb-(#s#)),wa,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-xchange-context-s NTRS(3) CEHFMT CASE(2) have ((s,cb-(#s#),e,(s,cb+(cb-(#s#)))]
    fg by blast
  from gtr1-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 \( \text{fst}\ (\alpha\ \text{fg}\ e) \cap \text{mon-ww}\ \text{fg}\ \text{wa} = \{\} \) by (auto simp add: mon-c-unconc)

from cil-cn-props2[OF IHAPP(2) this] show \(?thesis\). 

qed

moreover

have mon-c fg cb \(\cap\) (mon-c fg ca \(\cup\) mon-ww fg wa) = \{\} mon-c fg ca \(\cap\) (mon-c fg cb \(\cup\) mon-ww fg (e#wb)) = \{\}

proof –

from ntr-mon-increasing-s[OF FIRSTSTEP] IHAPP(4) show mon-c fg cb 
\(\cap\) (mon-c fg ca \(\cup\) mon-ww fg wa) = \{\} by auto

from MONED CASE have mon-c fg ca \(\cap\) mon-ww fg e = \{\} by (auto simp add: mon-c-unconc)

with ntr-mon-increasing-s[OF FIRSTSTEP] IHAPP(3) show mon-c fg ca 
\(\cap\) (mon-c fg cb \(\cup\) mon-ww fg (e#wb)) = \{\} by auto

qed

ultimately show \(?thesis\) by blast

qed

qed

The next lemma is a more general version of flowgraph.ntrp-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:

| ![c1 c2 s' c'].
| \([(s,(c1+c2),w,(s',c'))\in\text{trcl} (\text{ntrp}\ fg); \valid\ fg\ ((\#s#)+c1+c2)]
| \(\Rightarrow\ \exists\ w1\ w2\ c1'\ c2'.
| \(w\in w1\ \otimes\ \text{map}\ \ENV\ w2\ \land\)
| \(c'=c1'+c2'\ \land\)
| \(((s,c1),w1,(s',c1'))\in\text{trcl} (\text{ntrp}\ fg)\ \land\)
| \((c2,w2,c2')\in\text{trcl} (\text{ntrp}\ fg)\ \land\)
| \(\text{mon-ww}\ fg\ (\text{map}\ \text{le-rem-s}\ w1)\ \cap\ \text{mon-c}\ fg\ c2 = \{\} \land\)
| \(\text{mon-ww}\ fg\ w2\ \cap\ \text{mon-c}\ fg\ ((\#s#)+c1) = \{\})

proof (induct w)

| case Nil thus \(?case\) by (auto intro: ezI[of ~ []] ezI[of - {#}])

next

| case (Cons ee w) then obtain sh ch where SPLIT: \(((s,(c1+c2),ee,(sh,ch))\in\text{trcl}\ fg\ ((sh,ch),w,(s',c'))\in\text{trcl} (\text{ntrp}\ fg)\ by\ (\text{fast\ dest:}\ \text{trcl-uncons})

from SPLIT(1) show \(?case\) proof (cases rule: gtrp_cases)

| case gtrp-loc then obtain e where CASE: ee=LOC e ((s,(c1+c2),e,(sh,ch))\in\text{trrs}\ fg\ by\ auto

from ntrs-cases-s[OF CASE(2)] obtain csp where CHFMT: ch=\((\text{csp}+c1)+c2\)

\(\wedge\ s: \#: \text{csp} \Rightarrow \exists\ p\ u\ v.\ s = [\text{entry}\ \text{fg}\ p] \land (u, \text{Spawn}\ p, v) \in\ \text{edges}\ \text{fg}\ \land\ \text{initialproc}\ \text{fg}\ p\ by\ (\text{simp\ add:}\ union-assoc,\ blast))

with c-of-initial-no-mon have CSPMONOM: 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 \(((\#s#)+c1)\) by (simp add: union-ac)

from Cons,bgps[OF VALID, OF CASE(2)] obtain w1 w2 c1' c2' where IHAPP: w \in w1 \otimes\ \text{map}\ \ENV\ w2\ c' = c1' + c2' ((sh, csp + c1), w1, s', c1') \in\ trcl (\text{ntrp}\ fg)

75
(c2, w2, c2') \in \text{trecl}(\text{ntrp} f g) \text{ mon-ww fg (map le-rem-s w1)} \cap \text{mon-c fg c2} = \\
\{ \text{mon-ww fg w2} \cap \text{mon-c fg }\{\#s\#\} \} = \{ \text{by blast} \\
\text{have ee#w } \in \text{ee#w1 }^\oplus \text{of ant fg (map ENV w2) proof (rule cil-cons1) }
\text{ from ntrp-mon-env-w-no-ctx[OF SPLIT(2), unfolded mon-env-def] have mon-ww fg (map le-rem-s (env w)) } \cap \text{mon-s fg sh } = \{ \text{. }
\text{ moreover have mon-ww fg w2 } \subseteq \text{mon-ww fg (map le-rem-s (env w)) proof }
\text{ from cil-subset-il IHAPP(1) } \text{ileq-leave have map ENV w2 } \preceq \text{w by blast }
\text{ from le-list-filter[OF this] have env (map ENV w2) } \preceq \text{w by (unfold env-def)} \text{ blast }
\text{ hence map ENV w2 } \preceq \text{env w by (unfold env-def) simp }
\text{ from le-list-map[OF this, of le-rem-s] have w2 } \preceq \text{map le-rem-s (env w) by simp }
\text{ thus ?thesis by (rule mon-ww-ileq) qed }
\text{ ultimately have mon-ww fg w2 } \cap \text{mon-s fg sh } = \{ \text{by blast }
\text{ with ntrrs-mon-s[OF CASE(2)] CASE(1) show fst (ant fg ee) } \cap \text{mon-pl (map (ant fg) (map ENV w2)) } = \{ \text{by (auto simp add: cil-ax-cons-helper) qed (rule IHAPP(1)) }
\text{ moreover have ((s,c1),ee#w1,(s',c1'))}\in\text{trecl (ntrp fg) proof - }
\text{ from ntrrs-xchange-context-s[of s c1+c2 e sh csp fg c1] CASE(2) CHFMT(1) have ((s, c1), e, sh, csp + c1) } \in \text{ntrp fg by (auto simp add: mon-c-unconc union-ac) }
\text{ with CASE(1) have ((s, c1), ee, sh, csp + c1) } \in \text{ntrp fg by (auto intro: gtrp.gtrp-loc) }
\text{ also note IHAPP(3) }
\text{ finally show ?thesis . qed }
\text{ moreover from CASE(1) ntrrs-mon-c-no-ctx[OF CASE(2)] IHAPP(5) have mon-ww fg (map le-rem-s (ee#w1)) } \cap \text{mon-c fg c2 } = \{ \text{by (auto simp add: mon-c-unconc) }
\text{ moreover from ntrrs-mon-increasing-s[OF CASE(2)] CHFMT(1) IHAPP(6) have mon-ww fg w2 } \cap \text{mon-c fg }\{\#s\#\} + c1 = \{ \text{by (auto simp add: mon-c-unconc) }
\text{ moreover note IHAPP(2,4) }
\text{ ultimately show ?thesis by blast }
\text{ next }
\text{ case gtrp-env then obtain e ss ce ssh ceh where CASE: ec=ENV e c1+c2={\#s\#}+ce sh=s ch={\#ssh\#}+ceh ((ss,\{\#s\#\}+ce),e,(ssh,\{\#s\#\}+ceh)}\in\text{ntrrs fg by auto }
\text{ from ntrrs-cases-s[OF CASE(3)] obtain csp where HFMT: \{\#s\#\}+ceh = csp + ((\{\#s\#\}+ce) \land s :: csp } \Rightarrow \exists p u v. s = [\text{entry fg }p] \land (u, \text{Spawn } p, v) \in \text{edges fg }\land \text{initialproc }fg p \text{ by (blast) }
\text{ from union-left-cancel[of \{\#s\#\} ceh csp+ce] HFMT(1) have CEHFMT: ceh=csp+ce by (auto simp add: union-ac) }
\text{ from HFMT(2) have CHNOMON: mon-c fg csp } = \{ \text{ by (blast intro!: c-of-initial-no-mon) }
\text{ from CASE(2)[symmetric] show ?thesis proof (cases rule: mset-unplusm-dist-cases) }
\text{ — 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 \( \text{HFMT}'(1) \) \( \text{CASE}(4) \) \( \text{CEHFMT} \) have \( \text{CHFMT}': \) \( ch = (\text{csp} + \{\text{#ssh}\} + (c1 - \{\text{#ss}\})) \) + \( c2 \) by (simp add: union-ac)
    have \( \text{VALID}: \) \( \text{valid fg} \) \( (\{\text{#ssh}\} + \{\text{#ss}\} + (c1 - \{\text{#ss}\})) + c2 \)
proof –
  from \( \text{ntr-valid-preserve-s} \{\text{OF gtrl-s} \} \) \( \text{Cons.prems}(2) \) \( \text{CASE}(2) \)
  have \( \text{valid fg} \) \( (\{\text{#ssh}\} + (\{\text{#ss}\} + \text{ch})) \) by (simp add: union-assoc) (auto simp add: union-ac)
  with left \( \text{CEHFMT} \) show \( ?\text{thesis} \) by (auto simp add: union-ac)
qed

from \( \text{Cons.hyp}$\{\text{OF - VALID of s' c'} \} \) \( \text{CHFMT'} \) \( \text{SPLIT}(2) \) \( \text{CASE}(3) \) obtain
\( w1 \ w2 \ c1' \ c2' \) where \( \text{IHAPP}: \ w \in w1 \ @_{\text{csp}} \text{fg} \) \( \text{map ENV w2} \ c' = c1' + c2' \)
\((s, \text{csp} + \{\text{#ssh}\} + (c1 - \{\text{#ss}\}))\), \( w1, s', c1' \) \in trcl \( (\text{ntrp} \text{fg} (c2, w2, c2')) \) \( \text{proof (rule ci-lcons1)} \)
  from \( \text{IHAPP}(6) \) have \( \text{mon-ww} \ \text{fg} \ w2 \ \cap \ \text{mon-s} \ \text{fg} \ \text{ssh} = \{} \) by (auto simp add: mon-c-unconc)
    moreove from \( \text{ntrs-mon-s} \{\text{OF CASE}(5) \} \) \( \text{CASE}(1) \) have \( \text{fst} \) \( (\text{csp} \ \text{ee}) \)
\( \subseteq \text{mon-s} \ \text{fg} \) \( \text{ssh} \) \( \text{by auto} \)
    ultimately have \( \text{fst} \) \( (\text{csp} \ \text{ee}) \) \( \cap \text{mon-ww} \ \text{fg} \ w2 = \{} \) \( \text{by auto} \)
    moreover have \( \text{mon-pl} \) \( (\text{map} \ (\text{csp} \ \text{fg}) \ (\text{map ENV w2})) = \text{mon-ww} \ \text{fg} \ w2 \)
by (simp add: ci-ck-conss-helper)
    ultimately show \( \text{fst} \) \( (\text{csp} \ \text{ee}) \) \( \cap \text{mon-pl} \) \( (\text{map} \ (\text{csp} \ \text{fg}) \ (\text{map ENV w2})) \)
\( = \{} \) by auto
qed (rule \( \text{IHAPP}(1) \))

moreover
  have \( \text{SS} \): \( ((s,c1),ee,(s,\text{csp} + \{\text{#ssh}\} + (c1 - \{\text{#ss}\}))) \in ntrp \ \text{fg} \)
proof –
  from left \( \text{HFMT}(1) \) have \( \{\text{#ssh}\} + \text{ee} = (\{\text{#ssh}\} + (c1 - \{\text{#ss}\}) + c2 \ (\{\text{#ssh}\} + \text{cch} = \text{csp} + \{\text{#ssh}\} + (c1 - \{\text{#ss}\}))) + (\text{csp} + \{\text{#ssh}\} + (c1 - \{\text{#ss}\}))\)
by (simp all add: union-ac)
  with \( \text{CASE}(5) \) \( \text{ntrs-xchange-context-s} \{\text{OF ss} \ \{\text{#ssh}\} + (c1 - \{\text{#ss}\}) + c2 \)
\( \text{ss} \ \text{csp} \ \text{fg} \ ((\{\text{#ssh}\} + (c1 - \{\text{#ss}\}))) \) \( \text{have} \)
\( ((s, \{\text{#ssh}\} + (c1 - \{\text{#ss}\})), e, \text{ss}, \{\text{#ssh}\} + (\text{csp} + (c1 - \{\text{#ss}\}))) \)
\( \in \text{ntrs} \ \text{fg} \) \( (\text{auto simp add: mon-c-unconc \text{union-ac})} \)
from \( \text{gtrp.gtrp-ENV} \{\text{OF this} \} \) \( \text{left(1)[symmetric]} \) \( \text{CASE}(1) \) show \( ?\text{thesis} \) by
  (simp add: union-ac)
qed

from \( \text{trcl.cons} \{\text{OF this \text{IHAPP}(3)} \} \) \( \text{have} \ (s, c1, ee \# w1, s', c1') \in \text{trcl} \ (\text{ntrp} \ \text{fg}) \)
  moreover
  from \( \text{ntrs-mon-c-no-ctx} \{\text{OF CASE}(5) \} \) \( \text{left \text{CASE}(1) \text{IHAPP}(5)} \) \( \text{have} \ \text{mon-ww} \ \text{fg} \ (\text{map le-rem-s} \ (\text{ee#w1})) \) \( \cap \text{mon-c} \ \text{fg} \ w2 = \{} \) \( \text{by (auto simp add: mon-c-unconc)} \)
  moreover
  from \( \text{ntrp-mon-increasing-s} \{\text{OF SS} \} \) \( \text{IHAPP}(6) \) \( \text{have} \ \text{mon-ww} \ \text{fg} \ w2 \ \cap \text{mon-c} \ \text{fg} \ ((\{\text{#ssh}\} + c1) = \{} \) \( \text{by (auto simp add: mon-c-unconc)} \)
  moreover note \( \text{IHAPP}(2,4) \)
  ultimately show \( ?\text{thesis} \) by blast

77
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

```isar
  case right
  with HFMT(1) CASE(4) CEHFMT have CHFMT': ch=c1 + (csp+{#ssh#}+(c2-{#ss#}))
  by (simp add: union-ac)
  have VALID: valid fg ({#s#} + c1 + ((csp+{#ssh#}+(c2-{#ss#}))))
  proof –
    from ntr-valid-preserve-s[OF gtrI-s, OF CASE(5)] Cons.prems(2) CASE(2)
    have valid fg ({#ssh#} + ({#s#} + ceh)) by (simp add: union-assoc) (auto simp add: union-ac)
  with right CEHFMT show ?thesis by (auto simp add: union-ac)
  qed
from Cons.hyps[OF - VALID,of s' c'] CHFMT' SPLIT(2) CASE(3) obtain
w1 w2 c1' c2' where IHAPP: w ∈ w1 ⊗αn fg map ENV w2 c' = c1' + c2'
  ((s, c1), w1, s', c1') ∈ trcl (ntrp fg) (csp + {#ssh#} + (c2 - {#ss#})),
  w2, c2' ∈ trcl (ntr fg)
  mon-ww fg (map le-rem-s w1) ∩ mon-c fg (csp + {#ssh#} + (c2 - {#ss#})) = {}
  by blast
  have ee # w ∈ w1 ⊗αn 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 ntr-s-mon-s[OF CASE(5)] CASE(1) have fst (αn fg ee)
    ≤ mon-s fg ssh by auto
  ultimately have fst (αn fg ee) ∩ mon-ww fg (map le-rem-s w1) = {}
  by auto
  moreover have mon-pl (map (αn fg) w1) = mon-ww fg (map le-rem-s w1)
  by (unfold αn-def') (simp add: cil-an-cons-helper[symmetric])
  ultimately show fst (αn fg (ENV e)) ∩ mon-pl (map (αn fg) w1) = {}
using CASE(1) by auto
  qed (rule IHAPP(1))
overmore
  have SS: (c2,c,csp + {#ssh#} + (c2 - {#ss#})∈ntr fg proof –
    from right HFMT(1) have {#s#} + ceh=({#s#} + c1+(c2-{-#ss#})} #s# by (simp-all add: union-ac)
      with CASE(5) ntr-exchange-context-s[OF SS {#s#} + c1+(c2-{-#ss#})] e ssh csp fg c2-{-#ss#}) have
      ((s, c2 - {-#ss#}), e, ssh, csp+ (c2 - {-#ss#})) ∈ ntr fg by (auto simp add: mon-c-unconc union-ac)
    from gtrI-s[OF this] right(1)[symmetric] show ?thesis by (simp add: union-ac)
  qed
from trcl.cons[OF this IHAPP(4)] have (c2, e # w2, c2') ∈ trcl (ntr fg) .
overmore
from ntr-mon-increasing-s[OF SS] IHAPP(5) have mon-ww fg (map le-rem-s w1) ∩ mon-c fg c2 = {}
by (auto simp add: mon-c-unconc)
overmore
```
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:** \( w \in \text{wa} \otimes_{\text{an}} f g \text{wb} \text{ and } B: (ca, wa, ca') \in \text{trcl} (\text{ntr} f g) \)

\((cb, wb, cb') \in \text{trcl} (\text{ntr} f g)\)

mon-c \( fg \text{ ca} \cap (\text{mon-c} fg \text{ cb} \cup \text{mon-ww} f g \text{ wb}) = \{\} \)

mon-c \( fg \text{ cb} \cap (\text{mon-c} fg \text{ ca} \cup \text{mon-ww} f g \text{ wa}) = \{\} \)

shows \((ca + cb, w, ca' + cb') \in \text{trcl} (\text{ntr} f g)\)

**proof** —

— We have to generalize and rewrite the goal, in order to apply Isabelle’s induction method

**from** **A** have \( \forall ca \ ca'. (ca, wa, ca') \in \text{trcl} (\text{ntr} f g) \land (cb, wb, cb') \in \text{trcl} (\text{ntr} f g) \land \text{mon-c} f g \text{ ca} \cap (\text{mon-c} f g \text{ cb} \cup \text{mon-ww} f g \text{ wb}) = \{\} \land \text{mon-c} f g \text{ cb} \cap (\text{mon-c} f g \text{ ca} \cup \text{mon-ww} f g \text{ wa}) = \{\} \) \implies \((ca + cb, w, ca' + cb') \in \text{trcl} (\text{ntr} f g)\)

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

**proof** *(induct rule: \text{ntr-unsplit})*

— If both words are empty, the proposition is trivial

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

**next** — The first macrostep of the combined path was taken from the left operand of the interleaving

**case** *(left \( e \ w' w1' w2) \text{ thus} \ ?\text{case} \text{ proof} (\text{intro allI impI})*

**case** *(goal1 ca cb) hence **I:** \( w' \in w1' \otimes_{\text{an}} f g \text{ w2} \text{ fst } (\text{an} f g e) \cap \text{mon-pl (map (an f g) w2)} = \{\} *

!ca cb.

\[[(ca, w1', ca') \in \text{trcl} (\text{ntr} f g)\];

\((cb, w2, cb') \in \text{trcl} (\text{ntr} f g)\)

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

mon-c \( f g \text{ cb} \cap (\text{mon-c} f g \text{ ca} \cup \text{mon-ww} f g \text{ w1'}) = \{\} \)

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

\((ca, e \# w1', ca') \in \text{trcl} (\text{ntr} f g) (cb, w2, cb') \in \text{trcl} (\text{ntr} f g)\)

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

mon-c \( f g \text{ cb} \cap (\text{mon-c} f g \text{ ca} \cup \text{mon-ww} f g \text{ (e \# w1')}) = \{\} \) \text{ by blast+} — Split the left path after the first step

then obtain **cah where** \( \text{SPLIT:} (ca, e, ca) \in \text{ntr} f g (ca, w1', ca') \in \text{trcl} (\text{ntr} f g) \) \text{ by (fast dest: trcl-uncors)}

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

**from** \text{ntr-add-context-bl} \( \text{OF SPLIT}(1) \), where **cn=cb** \( \text{I}(7) \) have \((ca + cb, e, cah + cb) \in \text{ntr} f g \text{ by auto} \)

also

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

**have** \((cah + cb, w', ca' + cb') \in \text{trcl} (\text{ntr} f g) \text{ proof} —

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

with **I**(7) **have** \( \text{MON-CB:} \text{ mon-c} f g \text{ cb} \cap (\text{mon-c} f g \text{ cah} \cup \text{mon-ww} f g \text{ w1'})} \)
\[\{\} \text{ by } \text{auto}\]
\[\text{from } I(3)[\text{OF SPLIT}(2) \land I(5) \land \text{MON-CAH MON-CB}] \text{ show } ?\text{thesis} \ .\]
\[\text{qed}\]
\[\text{finally show } ?\text{case} \ .\]
\[\text{qed}\]
\[\text{next}\]
\[\text{— The first macrostep of the combined path was taken from the right path — this case is done completely analogous}\]
\[\text{case } (\text{right } e \ w' \ w2' \ w1) \text{ thus } ?\text{case proof } (\text{intro all impl})\]
\[\text{case } (\text{goal1 } c \ a \ c b) \text{ hence } I: \ w' \in w1 \otimes_{\text{an}} f g \ w2' \text{ fst } (\text{an } f g \ e) \cap \text{mon-pl (map } (\text{an } f g) \ w1) = \{\}\]
\[\text{!!ca } c b.\]
\[\[(ca, w1, ca') \in \text{trcl } (\text{ntr } f g);\]
\[\text{cb, w2', cb'} \in \text{trcl } (\text{ntr } f g);\]
\[\text{mon-c fg } ca \cap (\text{mon-c fg } cb \cup \text{mon-ww fg } w2') = \{\};\]
\[\text{mon-c fg } cb \cap (\text{mon-c fg } ca \cup \text{mon-ww fg } w1) = \{\} \implies (ca + cb, w', ca' + cb') \in \text{trcl } (\text{ntr } f g)\]
\[\text{(ca, w1, ca') \in \text{trcl } (\text{ntr } f g) (cb, e\# w2', cb') \in \text{trcl } (\text{ntr } f g)\]
\[\text{mon-c fg } ca \cap (\text{mon-c fg } cb \cup \text{mon-ww fg } (e\# w2')) = \{\} \]
\[\text{mon-c fg } cb \cap (\text{mon-c fg } ca \cup \text{mon-ww fg } w1) = \{\} \text{ by blast+}\]
\[\text{then obtain } cbh \text{ where } \text{SPLIT}: (cb, e, cbh) \in \text{ntr } f g \ (cb, w2', cb') \in \text{trcl } (\text{ntr } f g)\]
\[\text{by (fast dest: trcl-unscons)}\]
\[\text{from } \text{ntr-add-context-s}[\text{OF SPLIT}(1), \text{where } cn=ca] \ I(6) \text{ have } (ca + cb, e, ca + cbh) \in \text{ntr } f g \text{ by (auto simp add: union-commute)}\]
\[\text{also}\]
\[\text{have } (ca + cbh, w', ca' + cb') \in \text{trcl } (\text{ntr } f g) \text{ proof —}\]
\[\text{from } I(2, 6, 7) \text{ntr-mon-s}[\text{OF SPLIT}(1)] \text{ have } \text{MON-CBH}: \text{mon-c fg } cbh \cap (\text{mon-c fg } ca \cup \text{mon-ww fg } w1) = \{\} \text{ by (cases e) (auto simp add: cil-an-cons-helper)}\]
\[\text{with } I(6) \text{ have } \text{MON-CA: mon-c fg } ca \cap (\text{mon-c fg } cbh \cup \text{mon-ww fg } w2') = \{\} \text{ by auto}\]
\[\text{from } I(3)[\text{OF I(4) SPLIT}(2) \land \text{MON-CA MON-CBH}] \text{ show } ?\text{thesis} \ .\]
\[\text{qed}\]
\[\text{finally show } ?\text{case} \ .\]
\[\text{qed}\]
\[\text{with } B \text{ show } ?\text{thesis by blast}\]
\[\text{qed}\]

\text{lemma } (\text{in flowgraph}) \text{ ntrp-unsplit:}\]
\text{assumes } A: w \in w a \otimes \text{and } f g (\text{map } \text{ENV } w b) \text{ and}\\B: (s, ca), w a, (s', ca') \in \text{trcl } (\text{ntr } f g)\\(cb, wb, cb') \in \text{trcl } (\text{ntr } f g)\\\text{mon-c fg } ((\#s\#) + ca) \cap (\text{mon-c fg } cb \cup \text{mon-ww fg } wb) = \{\}\\\text{mon-c fg } cb \cap (\text{mon-c fg } ((\#s\#) + ca) \cup \text{mon-ww fg } (\text{map } \text{le-rem-s } wa)) = \{\}\\\text{shows } ((s, ca + cb), w, (s', ca' + cb')) \in \text{trcl } (\text{ntr } f g)\\\text{proof } —\\\text{fix } wb'\\\text{have } w \in w a \otimes \text{and } f g w b' \implies
∀s ca cb wb. w\prime = \text{map ENV} \ w \land
\((s,\text{ca}),w,(s',\text{ca}')\) ∈ trcl (ntrp fg) \land (\text{cb},w,\text{cb}') \in \text{trcl} (\text{ntrp} fg) \land \text{mon-c fg}
\((\#s\#)+ca) \land (\text{mon-c fg} \ cb \cup \text{mon-ww fg} \ wb) = \{} \land \text{mon-c fg cb} \cap (\text{mon-c fg}
\((\#s\#)+\text{ca}) \cup \text{mon-ww fg} \ (\text{map le-rem-s} \ \text{wa})) = \{} \rightarrow
\((s,\text{ca}+\text{cb}),w,(s',\text{ca}'+\text{cb}')\) ∈ trcl (ntrp fg)

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

\text{case empty thus \ ?case by simp}

\text{next}

\text{case (left e w' w1' w2) thus \ ?case proof (intro allI impl)}

\text{case (goal1 s ca cb wb) hence I: w' ∈ w1' ⊗_\text{arr} fg \ w2 \ \text{fst (arr fg e)} \cap
\text{mon-pl (map (arr fg) w2) = \{}}

!!s ca cb wb. [w2 = map ENV \ wb;
\((s, \text{ca}), w', s', \text{ca}' \) ∈ trcl \ (ntrp fg);
\((\text{cb}, \text{wb}, \text{cb}') \) ∈ trcl \ (\text{ntrp fg});
\text{mon-c fg} \ ((\#s\#) + \text{ca}) \cap (\text{mon-c fg} \ cb \cup \text{mon-ww fg} \ wb) = \{\};
\text{mon-c fg cb} \cap (\text{mon-c fg} \ ((\#s\#) + \text{ca}) \cup \text{mon-ww fg} \ \text{(map le-rem-s} \ w1') = \{\})

\[\rightarrow\] \ ((s, \text{ca} + \text{cb}), \ w', \ s', \text{ca}'+ \text{cb}') \in trcl (ntrp fg)

\text{w2 = map ENV} \ wb
\((s, \text{ca}), e \neq w1', s', \text{ca}' \) ∈ trcl \ (ntrp fg)
\((\text{cb}, \text{wb}, \text{cb}') \) ∈ trcl \ (\text{ntrp fg});
\text{mon-c fg} \ ((\#s\#) + \text{ca}) \cap (\text{mon-c fg} \ cb \cup \text{mon-ww fg} \ wb) = \{};
\text{mon-c fg cb} \cap (\text{mon-c fg} \ ((\#s\#) + \text{ca}) \cup \text{mon-ww fg} \ \text{(map le-rem-s} \ (e \neq w1')) = \{\})

\text{by blast+}

\text{then obtain sh cah where SPLIT: ((s,ca),e,(sh,cah))∈ntrp fg ((sh,cah),w1',(s',ca'))∈trcl}
\text{(ntrp fg) by (fast dest: trcl-uncs)}

\text{from ntrp-add-context-s(OF SPLIT(1)), of cb} \ I(8) \ \text{have ((s, ca + cb), e, sh, cah + cb) ∈ ntrp fg by auto}

\text{also have ((sh,cah+cb),w1',(s',ca'+cb'))∈trcl (ntrp fg) proof (rule I(3))}

\text{from ntrp-mon-s(OF SPLIT(1)] I(2,4,7,8) show 1: mon-c fg ((\#sh\#)}
\text{+ cah) \cap (\text{mon-c fg cb} \cup \text{mon-ww fg} \ wb) = \{\})

\text{by (cases e) (case-tac a, simp add: cil-arr-cons-helper, fastforce simp add: cil-arr-cons-helper)+}

\text{from I(8) I show mon-c fg cb} \cap (\text{mon-c fg} \ ((\#sh\#} + \text{ca}) \cup \text{mon-ww fg} \ \text{(map le-rem-s} \ w1')) = \{\}) \ \text{by auto}

\text{qed (auto simp add: I(4,6) SPLIT(2))}

\text{finally show ?case .}

\text{qed}

\text{next}

\text{case (right ee w' w2' w1) thus \ ?case proof (intro allI impl)}

\text{case (goal1 s ca cb wb) hence I: w' ∈ w1 ⊗_\text{arr} fg \ w2' \ \text{fst (arr fg e)} \cap
\text{mon-pl (map (arr fg) w1) = \{}}

!!s ca cb wb. [w2' = map ENV \ wb;
\((s, \text{ca}), w1, s', \text{ca}' \) ∈ trcl \ (ntrp fg);
\((\text{cb}, \text{wb}, \text{cb}') \) ∈ trcl \ (\text{ntrp fg});
\text{mon-c fg} \ ((\#s\#) + \text{ca}) \cap (\text{mon-c fg cb} \cup \text{mon-ww fg} \ wb) = \{\});
\[ \text{mon-c fg cb} \cap (\text{mon-c fg} (\{\#s\#\} + \text{ca}) \cup \text{mon-ww fg} (\text{map le-rem-s w1})) = \{\} \]

\[ I \mapsto ((s, \text{ca} + \text{cb}), w', s', ca' + cb') \in \text{trcl} (\text{ntrp fg}) \]

\[ ee#w2' = \text{map ENV wb} \]

\[ ((s, \text{ca}), w1, s', ca') \in \text{trcl} (\text{ntrp fg}) \]

\[ (cb, \text{wb}, cb') \in \text{trcl} (\text{ntrp fg}) \]

\[ \text{mon-c fg} (\{\#s\#\} + \text{ca}) \cap (\text{mon-c fg cb} \cup \text{mon-ww fg wb}) = \{\} \]

\[ \text{mon-c fg cb} \cap (\text{mon-c fg} (\{\#s\#\} + \text{ca}) \cup \text{mon-ww fg} (\text{map le-rem-s w1})) = \{\} \]

\[ \text{by fastforce} \]

\[ \text{from I(4) obtain e wb' where EE: \text{wb}'=e#wb' ee=ENV e w2'=map ENV wb' by (cases wb, auto)} \]

\[ \text{with I(6) obtain cbh where SPLIT: (cb,e,cbh)\in ntr fg (cbh,wb',cb')\in trcl (ntr fg)} \]

\[ \text{by (fast dest: trcl-uncors)} \]

\[ \text{have ((s, \text{ca} + \text{cb}), ee, (s, \text{ca} + \text{cbh})) \in \text{ntrp fg proof}} \]

\[ \text{from gtrE[OF SPLIT(1)] obtain sb cbh sbh cehb where NTRS: cb =} \]

\[ \{\#s\#\} + \text{ceh} \]

\[ \text{from ntr-add-context-s[OF NTRS(3), of \{\#s\#\}+ca] EE(1) I(7) have} \]

\[ ((s, \{\#s\#\} + (ca+ceh)), e, \text{sbh}, \{\#s\#\} + (ca+ceh)) \in \text{ntr fg by (auto simp add: union-ac)} \]

\[ \text{from gtrp-env[OF this] NTRS(1,2) EE(2) show \?thesis by (simp add: union-ac)} \]

\[ \text{qed} \]

\[ \text{also have ((s,ca+cbh),w',(s',ca'+cb'))\in trcl (ntrp fg) proof (rule I(3))} \]

\[ \text{from ntr-mon-s[OF SPLIT(1)] I(2,4,7,8) EE(2) show 1: \text{mon-c fg cehb}} \]

\[ \cap (\text{mon-c fg} (\{\#s\#\} + \text{ca}) \cup \text{mon-ww fg} (\text{map le-rem-s w1})) = \{\} \]

\[ \text{by (cases e) (simp add: cil-cah-cons-helper, fastforce simp add: cil-cah-cons-helper)} \]

\[ \text{from I(7) 1 EE(1) show mon-c fg (\{\#s\#\} + \text{ca}) \cap (\text{mon-c fg cbh} \cup} \]

\[ \text{mon-ww fg wb')} = \{\} \text{ by auto} \]

\[ \text{qed (auto simp add: EE(3) I(5) SPLIT(2))} \]

\[ \text{finally show \?case .} \]

\[ \text{qed} \]

\[ \text{qed} \]

\[ \text{with A B show \?thesis by blast} \]

\[ \text{qed} \]

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) \(\rightarrow\)

\( (ca+cb,w',e)\in\text{trcl (ntr fg)} \quad \leftrightarrow \quad (\exists ca' cb' \quad \text{wa \ wb.} \)

\( e' = \text{ca' + cb'} \land \)

\( w' \in (\text{wa} \otimes \text{ca fg wb}) \land \)

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

\( \text{mon-c fg cb} \cap (\text{mon-c fg cb + mon-ww fg wb}) = \{\} \land \)

83
(ca,wa,ca')∈trcI (ntr fg) ∧ (cb,wb,cb')∈trcI (ntr fg))

by (blast intro: ntr-split ntr-unsplit)

— Here is the corresponding version for executions with an explicit local thread

**Theorem (in flowgraph) ntrp-interleave:**

valid fg ((#s#)+c1+c2) ⇒
((s,c1+c2),w,(s',c1'))∈trcl (ntrp fg) \[→\]
(∃ w1 w2 c1' c2'.
  w ∈ w1 ⊗ anl fg (map ENV w2) ∧
  c'=c1'+c2' ∧
  ((s,c1),w1,(s',c1'))∈trcl (ntrp fg) ∧
  (c2,w2,c2')∈trcl (ntr fg) ∧
  mon-ww fg (map le-rem-s w1) ∩
  mon-c fg c2 = {} ∧
  mon-ww fg w2 ∩ mon-c fg ((#s#)+c1) = {})

apply (intro iffI)
apply (blast intro: ntrp-split)
apply (auto intro! : ntrp-unsplit simp add: valid-unconc)
done

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

**Corollary (in flowgraph) ntr2ntrp:**

(\[(c,w,\{#s#\}+ce)\in trcl (ntr fg);\]
  mon-c fg ((#s#)+c1) ∩ (mon-c fg c ∪ mon-ww fg w)={})
\[⇒\]((s,c1+c'),map ENV w,(s',c1'))∈trcl (ntrp fg)

using ntrp-unsplit[where wa=[], simplified] by fast

8.5.4 Reverse splitting

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

**Lemma (in flowgraph) ntr-reverse-split:**

(\[(c,w,\{#s#\}+ce)\in trcl (ntr fg);\]
  valid fg c \[⇒\]
  ∃ s ce w1 w2 ce1' ce2'.
  c={#s#}+ce ∧
  ce'=ce1'+ce2' ∧
  w∈w1 ⊗ on fg w2 ∧
  mon-s fg s ∩ (mon-c fg ce ∪ mon-ww fg w2) = {} ∧
  mon-c fg ce ∩ (mon-s fg s ∪ mon-ww fg w1) = {} ∧
  ((#s#),w1,\{#s'\}+ce1')∈trcl (ntrp fg) \[∧\]
  (ce, w2, ce2')∈trcl (ntr fg))

— The proof works by induction on the initial configuration. Note that configurations consist of finitely many threads only
— FIXME: An induction over the size (rather then over the adding of some fixed element) may lead to a smoother proof here.

**proof (induct c rule: multiset-induct)**

— If the initial configuration is empty, we immediately get a contradiction

**case empty hence False by auto thus ?case ..**

**next**

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

**case (add ce s)**

— We split the path by this initial configuration

**from ntr-split[OF add.prems(1,2)] obtain ce1' ce2' w1 w2 where**

**SPLIT: \{\#s\#\} + ce' = ce1' + ce2' w \in w1 \cap \alpha n fg w2**

**mon-c fg ce \cap (mon-s fg s \cup mon-wu fg w1) = {}**

**mon-s fg s \cap (mon-c fg ce \cup mon-wu fg w2) = {}**

**\{(\#s\#\},w1,ce1') \in trcl (ntr fg)**

\(ce,ce2',w2\) \in trcl (ntr fg)

by auto

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

**from SPLIT(1) show ?case proof (cases rule: mset-unplus-dist-cases)**

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

**case left**

**with SPLIT show ?thesis by fastforce**

**next**

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

**case right with SPLIT(6) have C: (ce,w2,\{\#s\#\}+(ce2' - \{\#s\#\})) \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 \cap mon-c fg ce = {}**

by (simp-all add: valid-unconc)

**from add.hyps[OF C VALID(1)] obtain st cet w21 w22 ce21' ce22' where**

**IHAPP:**

**ce=\{\#st\#\} + cet**

**ce2' - \{\#s\#\} = ce21' + ce22'**

**w2 \in w21 \cap \alpha n fg w22**

**mon-s fg st \cap (mon-c fg cet \cup mon-wu fg w22) = {}**

**mon-c fg cet \cap (mon-s fg st \cup mon-wu fg w21) = {}**

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

\((ce,w22,ce22') \in 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 w2 w21] SPLIT(2) IHAPP(3) obtain wl where**

**CASSOC: w \in w21 \cap \alpha n fg w2 w21 \cap \alpha n fg w22 by (auto simp add: cil-commute)**

**from CASSOC IHAPP(1,3,4,5) SPLIT(3,4) have COMBINE: (\{\#s\#\} + cet, wl, ce1' + ce22') \in trcl (ntr fg)**

by (rule-tac ntr-unsplit[OF CASSOC(2) SPLIT(5) IHAPP(7)]) (auto simp add: mon-c-unconc mon-ww-cil)

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

**mon-c fg (\{\#s\#\} + cet) \cap (mon-s fg st \cup mon-wu fg w21) = {}**

by (auto simp add: mon-c-unconc mon-ww-cil)

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

85
ce' = ce21' + (ce1' + ce22') by (simp-all add: union-ac)
moreover note IHAPP(6) CASSOC(1)
ultimately show thesis by blast
qed
qed
end

9 Constraint Systems

theory ConstraintSystems
imports Main AcquisitionHistory Normalization
begin

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

Constraint systems are widely used in static program analysis. There least
solution describes the desired analysis information. In its generic form, a
constraint system \( R \) is a set of inequations over a complete lattice \((L, \sqsubseteq)\)
and a set of variables \( V \). An inequation has the form \( R[v] \sqsupseteq \text{rhs} \), 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:

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

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
\((\mathcal{P}(\mathcal{T}a), \subseteq)\) for some type \( 'a \). This admits a convenient formalization in Is-
abelle/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 \( S\text{-cs fg } k \):

\(^3\)Variables are identified by control nodes here
\[(u, \text{Call } q, v) \in \text{edges } fg; (u, M, P) \in S\text{-cs } fg \ k; \\
\quad (\text{return } fg \ q, Ms, Ps) \in S\text{-cs } fg \ k; P' \leq \#P + Ps; \text{ size } P' \leq k \] \\
\implies (v, mon fg q \cup M \cup Ms, P') \in S\text{-cs } fg \ k

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

9.1 Same-level paths

9.1.1 Definition

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

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

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

\begin{verbatim}
lemma (in flowgraph) S-sound: \(!p \forall v \exists P.
\forall\{e\},w,((v,[c])\} \in trcl (trss fg); size P \leq k; (\lambda p. \text{entry fg p}) \# P \leq c '\
\implies (v,mon-w fg w,P) \in S-cs fg k
proof (induct w rule: length-compl-rev-induct)
\hspace{0.5cm}case Nil thus \textit{by} (auto intro: S-init)
next
\hspace{0.5cm}case (snoc w e) then obtain sh ch where SPLIT: ((\{entry fg p\},\{\#\}),w,(sh,ch)) \in trcl (trss fg) ((sh,ch),e,([v],[c])) \in trss fg by (fast dest: trcl-rev-uncons)
\hspace{0.5cm}from SPLIT(2) show \textit{?case proof} (cases rule: trss.cases)
\hspace{0.5cm}\hspace{0.5cm}case trss-base then obtain u a where CASE: e=\text{LBase} a sh=[u] ch=e'
\hspace{0.5cm}\hspace{0.5cm}by auto
\hspace{0.5cm}\hspace{0.5cm}with snoc.hyps[of w p u c'] OF - - snoc.prenms(2,3) SPLIT(1) have (u,mon-w fg w,P) \in S-cs fg k by blast
\hspace{0.5cm}\hspace{0.5cm}moreover from CASE(1) have mon-e fg e = {} by simp
\hspace{0.5cm}\hspace{0.5cm}ultimately show \textit{?thesis using S-base[of CASE(4)]} by (auto simp add: mon-w-unconc)
next
\hspace{0.5cm}case trss-ret then obtain q where CASE: e=\text{LRet} sh=\text{return} fg q \#[v] ch=e'
\hspace{0.5cm}by auto
\hspace{0.5cm}with SPLIT(1) have ((\{entry fg p\},\{\#\}),w,[\text{return} fg q,v],c') \in trcl (trss fg) by simp
\hspace{0.5cm}from trss-find-call[of this] obtain at ct w1 w2 where FC:
\hspace{0.5cm}w=w1 \oplus \text{LCall} q \#[w2]
\hspace{0.5cm}((\{entry fg p\},\{\#\}),w1,([ut],ct)) \in trcl (trss fg)
\hspace{0.5cm}((ut,ct),\text{LCall} q,([\text{return} fg q,v],ct)) \in trss fg
\hspace{0.5cm}(ut,\text{Call} q,v) \in edges fg
\hspace{0.5cm}((\{entry fg q\},ct),w2,([\text{return} fg q],c')) \in trcl (trss fg).
\hspace{0.5cm}from trss-drop-all-context[of FC(5)] obtain esp' where SLP: c'=ct+esp'
\hspace{0.5cm}((\{entry fg q\},\{\#\}),w2,([\text{return} fg q],esp')) \in trcl (trss fg) by (auto simp add: union-ac)
\end{verbatim}
from FC(1) have LEN: \( length \ w1 \leq length \ w \) \( length \ w2 \leq length \ w \) by auto
from mset-map-split-orig-le SLP(1) snoc.prems(3) obtain P1 P2 where PSPLIT: \( P = P1 + P2 \) \((\lambda p. \{\text{entry } fg \ p\})' \# P1 \leq c't \ (\lambda p. \{\text{entry } fg \ p\})' \# P2 \leq csp'\) by blast
with snoc.prems(2) have PSIZE: size P1 \leq k size P2 \leq k by auto
from snoc.hyps[OF LEN(1) FC(2) PSIZE(1) PSPLIT(2)] snoc.hyps[OF LEN(2) SLP(2) PSIZE(2) PSPLIT(3)] have IHAPP: \((\text{ut, mon-w } fg \ w1, P1)\) \(\in\) S-cs fg k \((\text{return } fg \ q, \text{mon-w } fg \ w2, P2)\) \(\in\) S-cs fg k.
from S-call[OF FC(4)] IHAPP mset-le-eq-refl[OF PSPLIT(1)] snoc.prems(2)] FC(1) CASE(1) show \((\text{v, mon-w } fg \ (w@]}\ c)), P) \(\in\) S-cs fg k by (auto simp add: mon-w-unconc Un-ac)
next
case trss-spawn then obtain u q where CASE: \( e = L\text{Spawn } q sh = [u] \ c' = \{\# entry fg q\} \# + ch \ (u, \text{Spawn } q, v)\) \(\in\) edges fg by auto
from mset-map-split-orig-le CASE(3) snoc.prems(3) obtain P1 P2 where PSPLIT: \( P = P1 + P2 \) \((\lambda p. \{\text{entry } fg \ p\})' \# P1 \leq \{\# entry fg q\} \# \ (\lambda p. \{\text{entry } fg \ p\})' \# P2 \leq ch \) by blast
with snoc.prems(2) have PSIZE: size P2 \leq k by simp
from snoc.hyps[OF \cdot - - PSIZE PSPLIT(3)] SPLIT(1) CASE(2) have IHAPP: \((\text{u,mon-w } fg \ w,P2)\) \(\in\) S-cs fg k by blast
have PCOND: \( P \leq \{\# q\} + P2 \) proof
from PSPLIT(2) have P1 \leq \{\# q\} by (auto elim!: mset-le-single-cases mset-map-single-rightE)
with PSPLIT(1) show ?thesis by simp
qed
from S-spawn[OF CASE(4)] IHAPP PCOND snoc.prems(2)] CASE(1) show \((\text{v, mon-w } fg \ (w@]}\ c)), P) \(\in\) S-cs fg k by (auto simp add: mon-w-unconc)
qed
qed

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

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

lemma (in flongraph) S-precise: \((\text{v,M,P})\)\(\in\)S-cs fg k
\(\Rightarrow\) \(\exists \ p \ c' w.\)
\([([\text{entry } fg \ p],\{\#\}),w,([v],c')]\)\(\in\)trcl (trss fg) \(\wedge\)
size P \(\leq\) k \(\wedge\)
\((\lambda p. \{\text{entry } fg \ p\})' \# P \leq c' \wedge\)
M = mon-w fg w
proof (induct rule: S-cs.induct)
case (S-init p) have \(\text{case } (\text{([entry } fg \ p],\{\#\}),\text{,[([entry } fg \ p],\{\#\})])\)\(\in\)trcl (trss fg) by simp-all
thus ?case by fastforce

89
next

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

\text{note IHAPP(1)}

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

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

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

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

\text{next}

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

\((([\text{entry } fg \ p], \{\#\}), w1, [u], csp1) \in \text{trcl (trss } fg) \text{ size } P \leq k (\lambda p. [\text{entry } fg \ p]) \quad \# \ P \leq csp1 M = \text{mon-w } fg \ w1 \text{ by blast} \]

\text{from S-call obtain csp2 } w2 \text{ where SL-PATH:} 

\((([\text{entry } fg \ q], \{\#\}), w2, [\text{return } fg \ q], csp2) \in \text{trcl (trss } fg) \text{ size } Ps \leq k (\lambda p. [\text{entry } fg \ p]) \quad \# \ Ps \leq csp2 Ms = \text{mon-w } fg \ w2 \text{ by (blast dest: trss-er-path-proc-const)}

\text{from trss-c-no-mon[OF REACHING-PATH(1)] trss-c-no-mon[OF SL-PATH(1)] have NOMON:} \quad \text{mon-c } fg \ csp1 = \{\} \quad \text{mon-c } fg \ csp2 = \{\} \quad \text{by auto}

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

\text{note REACHING-PATH(1)}

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

\text{also from trss-add-context[OF trss-stack-comp[OF SL-PATH(1)]]} \text{ NOMON}

\text{have} (([\text{entry } fg \ q,v], csp1), w2, ([\text{return } fg \ q, v], csp1+csp2)) \in \text{trcl (trss } fg) \text{ by (simp add: union-ac)}

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

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

\text{qed}

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

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

\text{from mset-map-le[OF S-call(6)] have } \# P' \leq \# P + \# Ps \text{ by (auto simp add: mset-map-union)}

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

\ldots \leq csp1+csp2 

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

\text{qed}

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

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

\text{next}

\text{case (S-spawn } u \ q \ v \ M \ P P') \text{ then obtain } p \ c' \ w \text{ where IHAPP:} 

\((([\text{entry } fg \ p], \{\#\}), w, [u], c') \in \text{trcl (trss } fg) \text{ size } P \leq k (\lambda p. [\text{entry } fg \ p]) \quad \# \ P \leq c' M = \text{mon-w } fg \ w \text{ by blast} \]

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

finally have $((\text{entry } fg p), \{\#\}), w \in [\text{LSpawn } q, [v], \{\text{entry } fg q\} \# + c') \in \text{trcl } (\text{trss } fg)$.

moreover from IHAPP(4) have $M \equiv \text{mon-w } fg \ (w \in [\text{LSpawn } q])$ by (simp add: mon-w-anconc)

moreover have $(\lambda p. [\text{entry } fg p]) \ (# P' \leq \{\text{entry } fg q\} \# + c' \ (\text{is } ?f \ '# \ - \leq \ -) proof –

from mset-map-le[of $S$-spawn(4)] have $??P' \leq \{\text{entry } fg q\} \# + ?f \ '# P$ by (auto simp add: mset-map-union)

also from mset-le-mono-add[of - IHAPP(3)] have $\ldots \leq \{\text{entry } fg q\} \# + c'$ by (auto intro: IHAPP(3))

finally show ?thesis .

qed

moreover note $S$-spawn(5)

ultimately show ?case by blast

qed

— Finally we can state the soundness and precision as a single theorem

**theorem (in flowgraph) $S$-sound-precise:**

$$(v, M, P) \in S\text{-cs } fg \ k \iff$

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

$\text{size } P \leq k \land (\lambda p. [\text{entry } fg p]) \ (# P' \leq c' \land M \equiv \text{mon-w } fg \ w )$

using $S$-sound $S$-precise by blast

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

**lemma (in flowgraph) $S$-sound-ntrp:**

assumes $A$: $((u,\{\#\}), \text{eel}, (sh, ch)) \in \text{ntrp } fg$ and

CASE: $!!p \ u' \ v \ w.$

$\text{eel} = \text{LOC } (\text{LCall } p \# \ w);$  
$\text{(u, LCall } p \# u) \in \text{edges } fg;$
$sh = [v, u']$;
$\text{proc-of } fg \ v = p;$
$\text{mon-c } fg \ ch = \{\};$
$!!s. s\#: ch \Longrightarrow \exists p \ u \ v. s = [\text{entry } fg p] \land$
$(u, \text{Spawn } p, v) \in \text{edges } fg \land$
$\text{initialproc } fg \ p;$
$!!P. (\lambda p. [\text{entry } fg p]) \ (# P \leq \ ch \Longrightarrow$
$(v, \text{mon-w } fg \ w, P) \in S\text{-cs } fg \ (\text{size } P)$

$\models \Longrightarrow \ Q$

shows $\ Q$

proof –

from $A$ obtain $ee$ where $EE$: $\text{eel} = \text{LOC } ee$ $((u,\{\#\}), \text{ee}, (sh, ch)) \in \text{entrs } fg$ by (auto elim: gtrp.cases)

have $\text{CHMT}$: $!!s. s\#: ch \Longrightarrow \exists p \ u \ v. s = [\text{entry } fg p] \land (u, \text{Spawn } p, v) \in \text{edges } fg$
$\land \text{initialproc } fg \ p$ by (auto intro: ntrp-c-cases-s[of EE(2)])

91
with c-of-initial-no-mon have CHNOMON: mon-c fg ch = {} by blast
from EE(2) obtain p u' v w where FIRSTSPLIT: ee=LCall p\# w \langle ([\{u\},\{\}\}], LCall p, ([\{ entry fg p, w\},\{\\}])\rangle \in trss fg sh\langle [v,u'], ([\{ entry fg p\},\{\}], w, ([v], ch))\rangle in trcl (trss fg) by (auto elim!: ntrs.cases[simplified])
  from FIRSTSPLIT have EDGE: (u, Call p, u') \in edges fg by (auto elim!: trss.cases)
  have PROC-OF-V: proc-of fg v = p by simp
  have !!P. (\lambda p. ![\{ entry fg p\}] (\#) P \leq ch \Longrightarrow (v, mon-w fg w, P) \in S\_cs fg (size P)
    proof –
      fix P assume (\lambda p. ![\{ entry fg p\}] (\#) P \leq ch
      from S\_sound[OF FIRSTSPLIT(4)] this, of size P show ?thesis P by simp
    qed
  with EE(1) FIRSTSPLIT(1,3) EDGE PROC-OF-V CHNOMON CHFMT show Q by (rule-tac CASE) auto
  qed

lemma (in flowgraph) S\_precise-ntrp:
assumes ENTRY: (v, M, P) \in S\_cs fg k and
  P: proc-of fg v = p and
  EDGE: (u, Call p, u') \in edges fg
shows \exists w ch.
  \langle ([\{u\},\{\}\}], LOC (LCall p\# w, ([v,u'], ch))\rangle \in ntrp fg \land
  size P \leq k \land
  M = mon-w fg w \land
  mon-n fg v = mon fg p \land
  (\lambda p. ![\{ entry fg p\}] (\#) P \leq ch \land
    mon-c fg ch = \{\})
proof –
  from P S\_precise[OF ENTRY, simplified] trss-bot-prec-const[where s=\[] and
  s'=\[], simplified] obtain usl ch where
    SLPATH: \langle ([\{ entry fg p\}, \{\}], usl, [v], ch)\rangle \in trcl (trss fg) size P \leq k (\lambda p. ![\{ entry fg p\}] (\#) P \leq ch \land
      M = mon-w fg usl \land
    from mon-n-same-proc[OF trss-bot-prec-const[where s=\[] and
      s'=\[], simplified, OF SLPATH(1)]] have MON-V: mon-n fg v = mon fg p by (simp)
    from trss-cases[OF SLPATH(1), simplified] have CHFMT: \land s, s' : \# ch \Longrightarrow \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
  — From the constraints prerequisites, we can construct the first step
  have FS: \langle ([\{u\},\{\}], LCall p\# usl, ([v,u'], ch))\rangle \in ntrs fg proof (rule ntrs-step[where r=\[], simplified])
  from EDGE show \langle ([\{u\}, \{\}], LCall p, ![\{ entry fg p, u\}], \{\})\rangle \in trss fg by (auto intro: trss-call)
  qed (rule SLPATH(1))
  hence FSP: \langle ([\{u\}, \{\}], LOC (LCall p\# usl), ([v,u'], ch))\rangle \in ntrp fg by (blast intro: gtrp-loc)
  from FSP SLPATH(2,3,4) CHNOMON MON-V show ?thesis by blast
  qed
9.2 Single reaching path

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

9.2.1 Constraint system

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

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

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

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

9.2.2 Soundness and precision

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

The proof follows from Lemma \textit{flowgraph.ntr-reverse-split} rather directly.

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

\[
\begin{align*}
\forall u \ s' \ c'. \ [((\{u\},\{\#\}),w,(s',c')) \in \text{trcl} (\text{ntrp } fg); \ atU U (\{\#s'\#\}+c')] \\
\rightarrow \exists Ml \ Me \ h. \\
(u,Ml,Me,h) \in \text{RU-cs } fg U \\
Ml \subseteq \text{mon-loc } fg w \\
Me \subseteq \text{mon-env } fg w \\
h \leq \alpha ah (\text{map } (\alpha nl fg) w)
\end{align*}
\]

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

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

— For a reaching path of length zero, the proposition follows immediately by the constraint RU-init

case Nil thus \?case by auto (auto intro! : RU-init)

next
case (Cons eel wwl)

— For a non-empty path, we regard the first step and the rest of the path

then obtain $sh \ ch$ where SPLIT:

\[
\begin{align*}
((\{u\},\{\#\}),eel,(sh,ch)) & \in \text{ntrp } fg \\
((sh,ch),wwl,(s',c')) & \in \text{trcl} (\text{ntrp } fg)
\end{align*}
\]

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: $\text{cel} = \text{LOC (LCall } p \ # \ w) (u, \ Call p, u') \in \text{edges } fg \ sh = [v, u']$

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: $\land s, s': ch \rightarrow \exists p \ u \ v. \ s=[\text{entry } fg \ p] \land (u, \text{Spawn } p,v) \in \text{edges } fg \land \text{initialproc } fg \ p$

— For the same-level path, we find a corresponding entry in the $S$-cs-constraint
system
and S-ENTRY-PAT: ∃P. (λp. [entry fg p]) ⌈# P ≤ ch → (v, mon-w fg w, P) ∈ S-cs fg (size P)

by (rule S-sound-ntrp[of SPLIT(1)]) blast

from ntrp-valid-preserve-s[of SPLIT(1)] have HVALID: valid fg ([#sh#] + ch) by simp
— We split the remaining path by the local thread and the spawned threads, getting two interleavable paths, one from the local thread and one from the spawned threads

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

LESLIT:

wwl∈wwl w2

(wwl ∈ trcl (ntrp fg)

(wwl, w2, c2') ∈ trcl (ntrp fg)

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

— 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)

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

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

— This does not affect the configuration being at U

from atU-xchange-stack left DECOMP(1) have ATU: atU U ((#v'##rr#)+c1') by fastforce

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

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

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

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

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

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

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

qed

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

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

moreover have mon fg p ∪ mon-w fg w ∪ MI ⊆ mon-loc fg (eel#wwl) using
mon-loc-ileq[OF ILEQ] IHAPP(2) FS-FMT(1) by fastforce
moreover have Me ⊆ mon-env fg (eel#wwl) using mon-env-ileq[OF ILEQ, of fg] IHAPP(3) by auto
moreover have ah-update h (mon fg p, mon-w fg w) (MI ∪ Me) ≤ ah (map (anl fg) (eel#wwl)) proof (simp add: ah-update-cons)
show ah-update h (mon fg p, mon-w fg w) (MI ∪ Me) ≤ ah-update (ah (map (anl fg) w1)) (mon-pl (map (anl fg) wwl)) proof (rule ah-update-mono)

from IHAPP(4) have h ≤ ah (map (anl fg) w1).
also from ah-ileq[OF le-list-map[OF ILEQ]] have ah (map (anl fg) w1) ≤ ah (map (anl fg) wwl).
finally show h ≤ ah (map (anl fg) wwl).

next
from FS-FMT(1) show (mon fg p, mon-w fg w) = anl fg eel by auto
next
from IHAPP(2,3) have (MI ∪ Me) ⊆ mon-pl (map (anl fg) w1) by (auto simp add: mon-pl-of-anl)
also from mon-pl-ileq[OF le-list-map[OF ILEQ]] have ... ⊆ mon-pl (map (anl fg) wwl).
finally show (MI ∪ Me) ⊆ mon-pl (map (anl fg) wwl).

qed
qed
ultimately show ?thesis by blast

next
case right — U was reached from the spawned threads
from cil-ileq[OF LESPLIT(1)] le-list-length[of map ENV w2 wwl] have ILEQ: map ENV w2≤wwl and LEN: length w2 ≤ length wwl by (auto)
from HVVALID have CHVALID: valid fg ch mon-s fg sh ∩ mon-c fg ch = {}
by (auto simp add: valid-uncnc)
— We first identify the actual thread from that U was reached
from ntr-reverse-split-atU[OF CHVALID(1) right LESPLIT(4)] obtain q wr cr' where RI: [entry fg q] #: ch wr≤w2 cr'≤c2' atU U cr' (∋[#][entry fg q]#),wr,cr')∈trcl (ntr fg) by (blast dest: CHFMT)
— In order to apply the induction hypothesis, we have to convert the reaching path to loc/env semantics
from ntr-reverse-split-atU[OF CHVALID(1) right LESPLIT(4)] obtain sr' cre' wwr
where RI-NTRP: cr'={#sr' #} + cre' wr=map le-rem-s wwr ([entry fg q],#),wrr,(sr',cre')∈trcl (ntrp fg) by blast
from LEN le-list-length[of RI(2)] RI-NTRP(2) have LEN': length wwr ≤ length wwl by simp
— The induction hypothesis yields a constraint system entry
from Cons.hyps[OF LEN' RI-NTRP(3)] RI-NTRP(1) RI(4) obtain MI Me h where IHAPP: (entry fg q, MI, Me, h)∈RU-cs fg U MI ⊆ mon-loc fg wwr Me ⊆ mon-env fg wwr h ≤ ah (map (anl fg) wwr) by auto
— We also have an entry in the same-level path constraint system that contains
the thread from that $U$ was reached

from \textit{S-ENTRY-PAT}[\textit{of \{#q#\}}, simplified] \textit{RI(1)} have \textit{S-ENTRY}: \langle v, \textit{mon-w fg w}, \{#q#\} \rangle \in \textit{S-cs fg 1 by auto}

— Before we can apply the \textit{RU-spawn-constraint}, we have to analyze the monitors

\textbf{have} \textit{MON-MLE-ENV}: $Ml \cup Me \subseteq \textit{mon-env fg wwl}$ \textbf{proof} —
from \textit{IHAPP(2,3)} have \textit{MHl} \textit{Ml} \textit{Me} \textit{Me} \subseteq \textit{mon-loc fg wwr} \cup \textit{mon-env fg wwr} by auto

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

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

\textbf{finally show} \textit{thesis}.

\textbf{qed}

— Finally we can apply the \textit{RU-spawn-constraint} that yields us an entry for the reaching path from $a$

from \textit{RU-spawn}[OF \textit{FS-FMT}(2,4)] \textit{S-ENTRY} - \textit{IHAPP(1)} \textit{MON-UP-MLE}

have $\langle u, \textit{mon-fg p} \cup \textit{mon-w fg w}, Ml \cup Me, \textit{ah-update h} (\textit{mon-fg p}, \textit{mon-w fg w}) \rangle \in \textit{RU-cs fg U by simp}

— Next we have to assemble the rest of the proposition

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

\textbf{moreover have} \textit{MHl} \textit{Ml} \textit{Me} \subseteq \textit{mon-env fg} (eel#wwl) using \textit{MON-MLE-ENV}

by auto

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

\textbf{proof} (simp add: \textit{ah-update-cons})

\textit{have MAP-HELPER}: \textit{map (anl fg)} wwr \leq \textit{map (anl fg)} wwl \textbf{proof} —
from \textit{RI-NTRP(2)} have \textit{map (anl fg)} wwr = \textit{map (anl fg)} wr by (simp add: \textit{anl-anl})

also from \textit{le-list-map}[OF \textit{RI(2)}] have ... \leq \textit{map (anl fg)} w2.

also have ... = \textit{map (anl fg)} (\textit{map ENV w2}) by simp

also from \textit{le-list-map}[OF \textit{ILEQ}] have ... \leq \textit{map (anl fg)} wwl.

\textbf{finally show} \textit{thesis}.

\textbf{qed}

\textbf{show} \textit{ah-update h} (\textit{mon-fg p}, \textit{mon-w fg w}) (\textit{Ml} \cup \textit{Me}) \leq \textit{ah-update} (\alpha\textit{ah} (\textit{map (anl fg)} wwl)) (\textit{anl fg} eel) (\textit{map-pl (map (anl fg) wwl)}) \textbf{proof} (rule \textit{ah-update-mono})

from \textit{IHAPP(4)} have $h \leq \alpha\textit{ah} (\textit{map (anl fg)} wwr)$.

also have ... \leq \alpha\textit{ah} (\textit{map (anl fg)} wwl) by (rule \alpha\textit{ah-ileq}[OF \textit{MAP-HELPER}])

\textbf{finally show} $h \leq \alpha\textit{ah} (\textit{map (anl fg)} wwl)$.

\textbf{next}

from \textit{FS-FMT(1)} \textbf{show} (\textit{mon-fg p}, \textit{mon-w fg w}) = \textit{anl fg eel} by simp
Now we prove a statement about the precision of the least solution. As in the precision proof of the S-cs constraint system, we construct a path for the entry on the conclusion side of each constraint, assuming that there already exists paths for the entries mentioned in the antecedent.

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

**Lemma (in flowgraph)** RU-precise: \((u, Ml, Me, h) \in RU-cs fg U\) implies \(\exists w s' c'. (([u], \{\#\}, w, (s', c')) \in trcl (ntrp fg) \land atU U ((\#s'\# + c') \land mon-loc fg w = Ml \land mon-env fg w = Me \land αah (map (αnl fg) w) = h)\)

**Proof** (induct rule: RU-cs.induct)
- The RU-init constraint is trivially covered by the empty path
- Case (RU-init) thus ?case by (auto intro: ext[of - []])

**Next**
- Call constraint
- Case (RU-call \(up u' v M P Ml Me h\))
- Then obtain \(w s' c'\) where IHAPP: \(((v), \{\#\}, w, s', c') \in trcl (ntrp fg) \land atU U ((\#s'\# + c') \land mon-loc fg w = Ml \land mon-env fg w = Me \land αah (map (αnl fg) w) = h)\) by blast
- From RU-call.hyps(2) S-precise[OF RU-call.hyps(3), simplified] trss-bot-proc-cons[where \(s = []\) and \(s' = []\), simplified] obtain \(wsl ch\) where SLPATH: 
  \(((\{entry fg p\}, \{\#\}), wsl, [v], ch) \in trcl (trss fg) M = mon-w fg wsl\)
  by fastforce
- From trss-cases[OF SLPATH(1), simplified] have CHFMT: \(\land s. s : \# ch \Rightarrow \exists p. s = [entry fg p] \land (\exists u v. (u, Spawn p, v) \in edges fg) \land initialproc fg p\) by blast
- With c-of-initial-no-mon have CHNOMON: \(\exists p. ch = {}\) by blast
- From the constraints prerequisites, we can construct the first step
- Have FS: \((([u], \{\#\}), LCall p\#wsl, ([v, u'], ch)) \in ntrp fg\) proof (rule ntrss-step[where 
  \(r = [],\) simplified])
- From RU-call.hyps(1) show \((([u], \{\#\}), LCall p, [entry fg p, u'], \{\#\}) \in trss fg\) by (auto intro: trss-call)
qed (rule SLPATH(1))

hence FSP: \(((\{u\}, \{\#\}), \text{LOC}(\text{LCall} \ p \ # \ wsl), (\{v, u'\}, ch))\)\ in \ ntrp \ fg \ by \ (\text{blast intro: gtrp-loc})

also
   — The rest of the path comes from the induction hypothesis, after adding the rest of the threads to the context
have \(((\{v, u'\}, ch), w, s' \circ [u'], c' + ch)\)\ in \ trcl \ (ntrp \ fg) \ proof (rule ntrp-add-context[OF ntrp-stack-comp[OF IHAPP(1)], where r=[u'], where cn=ch, simplified])

from RU-call.hyps(1,6) IHAPP(4) show \监控 run fg \ u' \ intersect \ mon-env \ fg \ w = \{\} \ by \ auto

from CHNOMON show \mon-wf \ fg \ \text{map le-rem-s} \ w \ intersect \ \mon-c \ fg \ ch = \{\} \ by \ auto

qed

finally have \(((\{u\}, \{\#\}), \text{LOC}(\text{LCall} \ p \ # \ wsl) \# w, s' \circ [u'], c' + ch)\)\ in \ trcl \ (ntrp \ fg) \ .

— It is straightforward to show that the new path satisfies the required properties for its monitors and acquisition history

moreover from IHAPP(2) have \text{at} U \ (\{\# \ s' \circ [u'] \} \cup (c' + ch)) \ by \ auto

moreover have \mon-loc \ fg \ \text{(LOC}(\text{LCall} \ p \ # \ wsl) \# w) \ = \ \mon-fg \ p \ intersect \ M \ intersect \ \Mon-Ml using SLPATH(2) IHAPP(3) by auto

moreover have \mon-env \ fg \ \text{(LOC}(\text{LCall} \ p \ # \ wsl) \# w) \ = \ \Mon-Me using IHAPP(4)

by auto

moreover have \text{oaah} \ (\text{map} \ (\text{cnil} \ fg) \ \text{(LOC}(\text{LCall} \ p \ # \ wsl) \# w)) \ = \ \text{ah-update} \ h \ (\mon-fg \ p, M) \ (\text{Ml} \ intersect \ \Mon-Me) \ proof —

have \text{oaah} \ (\text{map} \ (\text{cnil} \ fg) \ \text{(LOC}(\text{LCall} \ p \ # \ wsl) \# w)) \ = \ \text{ah-update} \ (\text{oaah} \ (\text{map} \ (\text{cnil} \ fg) \ w)) \ (\mon-fg \ p, \mon-w \ fg \ wsl) \ (\text{mon-pl} \ (\text{map} \ (\text{cnil} \ fg) \ w)) \ by \ (\text{auto simp add: ah-update-cons})

also have \ldots = \text{ah-update} \ h \ (\mon-fg \ p, M) \ (\text{Ml} \ intersect \ \Mon-Me) \ proof —

from IHAPP(5) have \text{oaah} \ (\text{map} \ (\text{cnil} \ fg) \ w) \ = \ h \ .

moreover from SLPATH(2) have \mon-fg \ p, \mon-w \ fg \ wsl) \ = \ (\mon-fg \ p, M) \ by \ (\text{simp add: mon-pl-of-cnil})

moreover from IHAPP(3,4) have \mon-pl \ (\text{map} \ (\text{cnil} \ fg) \ w) \ = \ \text{Ml} \ intersect \ \Mon-Me by (auto simp add: mon-pl-of-cnil)

ultimately show \?thesis by simp

qed

finally show \?thesis .

qed

ultimately show \?case by blast

next
   — Spawn constraint
   case (RU-spawn \ p \ u \ u' \ v \ M \ p \ q \ \Mon-Me \ h) \ then \ obtain \ w \ s' \ c' \ where \ IHAPP:
      \(((\text{entry} \ fg \ q), \{\#\}), \ w, s', c') \ in \ trcl \ (ntrp \ fg) \ \text{at}\ U \ (\{\#s'\#\} \cup c') \ \mon-loc \ fg \ w \ = \ \Mon-Me \ intersect \ wsl \ fg \ w = \ \Mon-Me \ intersect \ wsl \ by \ blast

from RU-spawn.hyps(2) S-precise[OF RU-spawn.hyps(3), simplified] trss-bot-proc-const[where s=\[] and s'=\[] simplifed] obtain wsl \ ch \ where

SLPATH: \(((\text{entry} \ fg \ p), \{\#\}), \ \text{wsl}, \ [v], ch) \ in \ trcl \ (trss \ fg) \ M \ = \ \text{mon-wf} \ wsl \ size \ P \ \leq \ 1 \ \text{by fastforce}

with RU-spawn.hyps(4) obtain che \ where \ PFMT: \ P=(\#q\#) \ ch \ = \ (\#[\text{entry} \ fg \ q]\#) \ + \ che \ by \ (\text{auto elim!: mset-size-le1-cases mset-le-addE})
from trss-cases[OF SLPATH(1), simplified] have CHNOMT: \( \wedge s. s \in [\text{entry } fg p] \land (\exists u v. (u, Spawn p, v) \in \text{edges } fg) \land \text{initialproc } fg p \) by blast

with c-of-initial-no-mon have CHNMON: \( \text{mon-c } fg \ ch = \{\} \) by blast

have FS: \( \{[u],[\#]\}, \text{LCall } p\#\text{wsl},([v,u'],ch)\} \subseteq \text{ntrs } fg \) proof (rule ntrs-step[where \( r=[] \), simplified])

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

qed (rule SLPATH(1))

hence FSP: \( ([u],[\#]), \text{LOC } (\text{LCall } p\#\text{wsl}),([v,u'],ch)\} \subseteq \text{ntrp } fg \) by (blast intro: grtp-loc)

also have \( \{([v,u'], ch), \text{map ENV } (\text{map le-rem-s } w), [v,u'], \text{che} + \{\#s'\#\} + c')\} \in \text{trcl } (\text{ntrp } fg) \)

proof —

from IHAPP(3,4) have \( \text{mon-ww } fg \) \( (\text{map le-rem-s } w) \subseteq Ml \cup Mc \) by (auto simp add: mon-ww-of-le-rem)

with RU-spawn.hyps(1,2,7) have \( (\text{mon-n } fg v \cup \text{mon-n } fg u') \cap \text{mon-ww } fg \) \( (\text{map le-rem-s } w) = \{\} \) by (auto simp add: mon-n-def edges-part)

with ntr2ntrp[OF grtp2gtr[OF IHAPP(1)], of \([v,u']\) che] PFMT(2) CHNMON show \( \text{thesis } \) by (auto simp add: union-ac mon-c-unconc)

qed

finally have \( \{([u],[\#]), \text{LOC } (\text{LCall } p \# \text{wsl}) \# \text{map ENV } (\text{map le-rem-s } w), [v,u'], \text{che} + \{\#s'\#\} + c')\} \in \text{trcl } (\text{ntrp } fg) \).

moreover from IHAPP(2) have \( \text{atU } U \) \( \{\#[v,u']\#\} + (\text{che} + \{\#s'\#\} + c')\)

by simp

moreover have \( \text{mon-loc } fg \) \( (\text{LOC } (\text{LCall } p \# \text{wsl}) \# \text{map ENV } (\text{map le-rem-s } w)) = \text{mon } fg p \cup M \) using SLPATH(2) by (auto simp del: map-map)

moreover have \( \text{mon-env } fg \) \( (\text{LOC } (\text{LCall } p \# \text{wsl}) \# \text{map ENV } (\text{map le-rem-s } w)) = Ml \cup Mc \) using IHAPP(3,4) by (auto simp add: mon-ww-of-le-rem simp del: map-map)

moreover have \( \text{ovh } (\text{map } (\text{onl } fg)) \) \( (\text{LOC } (\text{LCall } p \# \text{wsl}) \# \text{map ENV } (\text{map le-rem-s } w)) = \text{ah-update } h (\text{mon } fg p, M) \) (\( Ml \cup Mc \)) proof —

have \( \text{ovh } (\text{map } (\text{onl } fg)) \) \( (\text{LOC } (\text{LCall } p \# \text{wsl}) \# \text{map ENV } (\text{map le-rem-s } w)) = \text{ah-update } (\text{ovh } (\text{map } (\text{onl } fg)) \text{map le-rem-s } w)) \) \( (\text{mon } fg p, \text{mon-w } fg \text{ wsl}) \) \( (\text{mon-pl } (\text{map } (\text{onl } fg)) \text{map le-rem-s } w)) \)

by (simp add: ah-update-cons o-assoc)

also have \( \ldots = \text{ah-update } h (\text{mon } fg p, M) \) (\( Ml \cup Mc \)) proof —

from IHAPP(5) have \( \text{ovh } (\text{map } (\text{onl } fg)) \) \( (\text{map le-rem-s } w) = h \) by (simp add: ovh-onl)

moreover from SLPATH(2) have \( \text{mon-values } (\text{map } (\text{onl } fg)) \) \( (\text{map le-rem-s } w) = (\text{mon } fg p, M) \) by simp

moreover from IHAPP(3,4) have \( \text{ovh } (\text{map } (\text{onl } fg)) \) \( (\text{map le-rem-s } w) \) = \( Ml \cup Mc \) by (auto simp add: mon-pl-onl ovh-onl)

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

\textbf{inductive-set}

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

\textbf{where}

\begin{align*}
\text{RUV-call:} &\Rightarrow (u, \text{Call } p, u') \in \text{edges } fg; \text{proc-of } fg \ v = p; \ (v, M, P) \in S-cs \ fg \ 0; \\
&\ (v, Ml, Me) \in RUV-cs \ fg \ U \ V; \ \text{mon-n } fg \ u \cap \ Me = \{\} \ ] \\
\Rightarrow &\ (u, \text{mon } fg \ p \cup M \cup Ml, Me) \in RUV-cs \ fg \ U \ V
\end{align*}

\begin{align*}
\text{RUV-spawn:} &\Rightarrow (u, \text{Call } p, u') \in \text{edges } fg; \text{proc-of } fg \ v = p; \ (v, M, P) \in S-cs \ fg \ 1; \ q :# \ P; \\
&\ (\text{entry } fg \ q, Ml, Me) \in RUV-cs \ fg \ U \ V; \\
&\ (\text{mon-n } fg \ u \cup \text{mon } fg \ p) \cap (Ml \cup Me) = \{\} \ ] \\
\Rightarrow &\ (u, \text{mon } fg \ p \cup M, Ml \cup Me) \in RUV-cs \ fg \ U \ V
\end{align*}

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

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

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

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

- **RUV\_call**: Both, $U$ and $V$ are reached from the initial thread.
- **RUV\_spawn**: Both, $U$ and $V$ are reached from a single spawned thread.
- **RUV\_split\_le**: $U$ is reached from the initial thread, $V$ is reached from a spawned thread.
- **RUV\_split\_el**: $V$ is reached from the initial thread, $U$ is reached from a spawned thread.
- **RUV\_split\_ee**: Both, $U$ and $V$ are reached from different spawned threads.

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

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

### 9.3.2 Soundness and precision

**Lemma (in flowgraph) RUV\_sound**: 

\[
\left( \left( [([u],[\#]),w,(s',c')] \right) \in \text{trcl} \left( ntrp \ fg \right) ; \ atUV \ U \ V \ \{(\#s'+c')\} \right) \\
\Rightarrow \exists M_l, M_e. \ \ \\
(\left( u, M_l, M_e \right) \in \text{RU}-cs \ fg \ U \ V \ \& \ \\
M_l \subseteq \text{mon-loc} \ fg \ w \ \& \\
M_e \subseteq \text{mon-env} \ fg \ w)
\]

— The soundness proof is done by induction over the length of the reaching path

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

— In case of the empty path, a contradiction follows because a single-thread configuration cannot simultaneously be at two control nodes

- **case Nil** hence **False** by **simp** **thus** ?case ..

**next**

- **case** (Cons ee ww) **then obtain** sh ch **where** SPLIT: 

\[
\left( (\left( [\binom{u}{\#}),w,(s',c') \right) \in \text{trcl} \left( ntrp \ fg \right) \right) \left( (\left( [sh,\binom{\#}{ch}),w,(s',c') \right) \in \text{trcl} \left( ntrp \ fg \right) \right) \right) \text{ by (**fast dest**: trcl-uncons)}
\]

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

LE_SPLIT: \( w w \in w1 \odot_{\text{cmd}} \text{fg} \text{map ENV} w2 c' = c1' + c2' (\left( [sh, \{\#\}) \right), w1, s', c1') \in \text{trcl} \left( ntrp \ fg \right) \left( ch, w2, c2' \right) \in \text{trcl} \left( ntrp \ fg \right) \text{mon-ww fg} \text{mon-c fg} \text{ch} = \{\}\text{ mon-ww fg} \text{w2} \cap \text{mon-s fg} \text{sh} = \{\}\text{ by blast}

**obtain** p u' v w **where**

FS-FMT: \( ee = \text{LOC} \left( \text{LCall} \ p \ # w \right) \left( u, \text{Call} \ p, u' \right) \in \text{edges} \ fg \text{sh} = [v, u]\)

**proc-of fg v = p mon-c fg ch = \{\}\text{ by simp}
and CHFMT: \( \forall s, s : \# \, ch \implies \exists p \, u \, v. \, s = [\text{entry} \, fg \, p] \land (u, \text{Spawn} \, p, v) \in \text{edges} \, fg \land \text{initialproc} \, fg \, p \)

and S-ENTRY-PAT: \( \forall P. \, (\lambda p. \, [\text{entry} \, fg \, p]) \, '\# \, P \leq ch \implies (v, \text{mon-w} \, fg \, w, P) \in \text{S-cs} \, fg \, (\text{size} \, P) \)

by (rule S-sound-ntrp[\text{OF SPLIT}(1)]) blast
from ntrp-mon-env-w-no-ctz[\text{OF SPLIT}(2)] \text{FS-FMT}(3,4) \text{edges-part}[\text{OF FS-FMT}(2)]

have MON-PU: \( \text{mon-env} \, fg \, w \cap (\text{mon} \, fg \, p \cup \text{mon-n} \, fg \, w, P) \in \text{S-cs} \, fg \) by (auto simp add: mon-n-def)

from cil-ileq[\text{OF LESPLIT}(1)] \text{mon-loc-ileq}[of \, w1 \, \text{ww} \, fg] \text{mon-ileq}[of \, w1 \, \text{ww} \, fg] \text{have MON1-LEQ: mon-loc} \, \text{fg} \, w1 \subseteq \text{mon-loc} \, \text{fg} \, w1 \text{mon-ww} \, \text{fg} \, \text{ww} \text{by auto}

from cil-ileq[\text{OF LESPLIT}(1)] \text{mon-ileq}[of \, \text{map} \, \text{ENV} \, w2 \, \text{ww} \, fg] \text{have MON2-LEQ: mon-ww} \, \text{fg} \, w2 \subseteq \text{mon-ww} \, \text{fg} \, \text{ww} \text{by simp}

from LESPLIT(3) \text{FS-FMT}(3) \text{ntrp-stack-decomp}[of \, v \, \text{[]} \, [\#] \, \text{w1} \, s \, c1', \text{simplied}] \text{obtain} v' \, \text{rr} \text{where DECOMP-LOC: s'=}v' \# \text{rr}\text{[]}\text{w1},(v' \# \text{rr},c1')\in \text{trcl} \text{(ntrp} \, fg) \text{by (simp, blast)}

from Cons.prenses(2) \text{LESPLIT}(2) \text{have atUV} \, \text{U} \, \text{V} \, ((\#s'\#)+c1') \text{by (simp add: union-ac)}

thus \text{?case proof} \text{(cases rule: atUV-union-cases)}

case \text{left with DECOMP-LOC(1)} \text{have ATUV: atUV} \, \text{U} \, \text{V} \, ((\#v'\#\#\#)+c1')

by simp

from Cons.hyps[\text{OF - DECOMP-LOC(2) ATUV}] \text{cil-length}[\text{OF LESPLIT}(1)]

\text{obtain Ml Me where IHAPP: (v, Ml, Me) \in RUV-cs \, fg} \, U \, \text{V} \, \text{Ml} \subseteq \text{mon-loc} \, \text{fg} \, w1 \, \text{Me} \subseteq \text{mon-ww} \, \text{fg} \, \text{w1} \text{by auto}

from RUV-call[\text{OF FS-FMT}(2,4) \text{S-ENTRY-PAT}[\text{of} \, \{\#,\}, \text{simplied}] \text{IHAPP}(1)]

\text{have (u, mon fg p \cup mon-w} \, \text{fg} \, w \cup \text{Ml, Me) \in RUV-cs \, fg} \, U \, \text{V} \, \text{using IHAPP(3)} \text{MON-PU MON1-LEQ by fastforce}

\text{moreover have mon fg p \cup mon-w fg w \cup Ml \subseteq mon-loc fg (ee\#ww) using FS-FMT(1) IHAPP(2) MON1-LEQ by auto}

\text{moreover have Me \subseteq mon-ww fg (ee\#ww) using IHAPP(3) MON1-LEQ by auto}

ultimately show \text{?thesis by blast}

next

case \text{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' \text{ where R-STACKS: } \{\#s1'\#\}+\{\#s2'\#\} \leq c2' \text{ atU-s} \, \text{U} \, \text{s1'} \text{atU-s} \, \text{V} \, \text{s2'} \text{by (unfold atUV-def) auto}

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

\text{obtain q ceh} \, w21 \, w22 \, \text{ce21' where REVSGOV: ch=}\{\#[\text{entry} \, fg \, q\#]\}+\text{ceh} \, \{\#s2'\#\}+\text{ce22'} \text{=ce21'+ce22'} \, w2 \in w21 \, \text{of} \, \text{fg}\, w22\text{mon fg q} \cap (\text{mon-c} \, \text{fg} \, \text{ceh} \, \cup \, \text{mon-ww} \, \text{fg} \, w22)=\{} \text{mon-c} \, \text{fg} \, \text{ceh} \, \cap \, (\text{mon} \, \text{fg} \, q \, \cup \, \text{mon-ww} \, \text{fg} \, w21) = \{} \}

((\text{\{\#[\text{entry} \, fg \, q\#]\}+\text{ce21'})\in \text{trcl} \text{(ntrp} \, fg)) \text{(cale,ww,ce22')}\text{\in trcl} \text{(ntrp} \, fg))

\text{proof —}

case \text{goal1}

from ntr-reverse-split[\text{of} \, \text{ch} \, w2 \, s1' \, \{\#s2'\#\}+\text{ce22'} \text{ntrp-valid-preserv-s[}\text{OF}
SPLIT(1), simplified] C2'FMT LESPLIT(4)

obtain \texttt{ch} \texttt{cch} \texttt{w21} \texttt{w22} \texttt{ce21}' \texttt{ce22}' where
\texttt{ch=}\{\# \texttt{sch}\#\} + \texttt{cch} \{\#s2'\#\} + \texttt{ce2}' = \texttt{ce21}' + \texttt{ce22}' w2\in w21 \odot \texttt{mon-s} \texttt{fg} \texttt{seh} \cap (\texttt{mon-c} \texttt{fg} \texttt{cch} \cup \texttt{mon-ww} \texttt{fg} w22) = \{\} \texttt{mon-c} \texttt{fg} \texttt{cch} \cap (\texttt{mon-s} \texttt{fg} \texttt{seh} \cup \texttt{mon-ww} \texttt{fg} w21) = \{\}
\{(\# \texttt{sch}\#), \texttt{w21}, \{\#s1'\#\} + \texttt{ce21}'\} \in \texttt{trcl} (\texttt{ntr fg}) (\texttt{cch}, w22, ce22') \in \texttt{trcl} (\texttt{ntr fg}) 

by (auto simp add: valid-anconc)

moreover from this(1) CHFMT[\texttt{of seh}] obtain \texttt{q} where \texttt{seh=} \texttt{[entry fg q]}

by auto

ultimately have \texttt{ch=} \texttt{[#}}\texttt{entry fg q}\texttt{#} + \texttt{cch} \{\#s2'\#\} + \texttt{ce2}' = \texttt{ce21}' + \texttt{ce22}' w2\in w21 \odot \texttt{mon-fg} \texttt{w2 mon-fg} \texttt{q} \cap (\texttt{mon-c} \texttt{fg} \texttt{cch} \cup \texttt{mon-ww} \texttt{fg} w22) = \{\} \texttt{mon-c} \texttt{fg} \texttt{cch} \cap (\texttt{mon-fg} \texttt{q} \cup \texttt{mon-ww} \texttt{fg} w21) = \{\}
\{(\# \texttt{entry fg q}\texttt{#})\}, \texttt{w21}, \{\#s1'\#\} + \texttt{ce21}' \in \texttt{trcl} (\texttt{ntr fg}) (\texttt{cch}, w22, ce22') \in \texttt{trcl} (\texttt{ntr fg}) by auto

thus \texttt{thesis} using goal1 by (blast)

qed

— For applying the induction hypothesis, it will be handy to have the reaching path in loc/env format:

\texttt{from ntr.sgt2grtrp[where c=} \texttt{#}, simplified, OF REVSPPLIT(6)} obtain \texttt{sq'} 
\texttt{csp-q \texttt{ww21} where}
\texttt{R-CONV: \{\#s1'\#\} + \texttt{ce21}' = \texttt{#sq'\#} + \texttt{csp-q \texttt{w21} = map le-rem-s \texttt{ww21}} (((\texttt{[entry fg q]\texttt{#}, \{\#\}}), \texttt{w21}, \texttt{sq'}, \texttt{csp-q}) \in \texttt{trcl} (\texttt{ntrp fg}) by blast}

\texttt{from cil-ileq[OF REVSPPLIT(3)] mon-ww-ileq[of w21 \texttt{w2 fg}] mon-ww-ileq[of w22 \texttt{w2 fg}] have MON2N-LEQ: mon-ww \texttt{fg w21} \subseteq mon-ww \texttt{fg w22} mon-ww \texttt{fg w22} \subseteq mon-ww \texttt{fg w2 by auto}

\texttt{from REVSPPLIT(2) show \texttt{thesis} proof (cases rule: mset-anplusm-dist-cases[case-names \texttt{left' right'}])}

\texttt{case left' — Both nodes are reached from the same thread
have ATUV: atUV U V \{\#sq'\#\} + \texttt{csp-q} using right C2'FMT R-STACKS(2,3)
by (subst R-CONV(1)[symmetric], subst left'(1)) simp}

\texttt{from Cons.hups[OF - R-CONV(3) ATUV] cil-length[OF REVSPPLIT(3)] cil-length[OF LESPLIT(1)] R-CONV(2) obtain MI Me where IHAPP: (entry fg q, MI, Me) \in RUV-cs fg U V MI \subseteq mon-loc fgww21 Me \subseteq mon-env fgww21 by auto

\texttt{from REVSPPLIT(1) S-ENTRY-PAT[of \{\#q\#\}, simplified] have S-ENTRY: (v, mon-w fg w, \{\#q\#\}) \in \texttt{S-cs fg I by simp
\have MON-COND: (mon-n fg u \cup mon fg p) \cap (MI \cup Me) = \{\} proof —
\texttt{from R-CONV(2) have mon-ww fg w21 = mon-loc fg w21 \cup mon-ww fg w22 by (simp add: mon-ww-of-le-rem)
with IHAPP(2,3) MON2N-LEQ(1) MON-PU MON2-LEQ show \texttt{thesis
by blast

\texttt{qed
\texttt{from RUV-spawn[OF FS-FMT(2) FS-FMT(4) S-ENTRY - IHAPP(1)] MON-COND] have (u, mon fg p \cup mon-w fg w, MI \cup Me) \in RUV-cs fg U V by simp
moreover have mon fg p \cup mon-w fg w \subseteq mon-loc fg (ee\#ww) using FS-FMT(1) by auto
moreover have MI \cup Me \subseteq mon-env fg (ee\#ww) using IHAPP(2,3)
R-CONV(2) MON2N-LEQ(1) MON2-LEQ by (auto simp add: mon-ww-of-le-rem)
ultimately show \( \text{thesis by blast} \) next

\textbf{case right'} — The nodes are reached from different threads

from \texttt{R-STACKS(2,3)} have \( \text{ATUV\textendash} \text{RU(U \{\{#sq\'\}\}\plus csp-q}) \) \texttt{atU \text{V ce22'}}

by (--) (\texttt{subst \text{R-CONV}(1)[\text{symmetric}], simp, subst right'(1), simp})

— We have to reverse-split the second path again, to extract the second interesting thread

\texttt{obtain q' w22' ce22e' where REVSLIT': [entry fg q'] :: ceh w22' \leq w22 ce22e' \leq ce22' atU \text{V ce22e'} ((\{entry fg q\}, w22', ce22e') \in \text{trcl} \ (\text{ntr fg}) \ \text{proof})

—

case goal1

from \texttt{ntr-reverse-split-atU[\text{OF\textendash} \text{ATUV(2) REVSLIT(7)}] \text{ntrp-valid-preserve-s[\text{OF SPLIT(1), simplified}] REVSLIT(1)} \text{obtain sq'' w22' ce22e' where}

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

moreover from \texttt{CHFMT[of sq'']} \text{REVSLIT(1) this(1)} \text{obtain q' where}

\(sq'':=[\text{entry fg q}'] \) by auto

ultimately show \( \text{thesis using goal1 by blast} \)

\texttt{qed}

from \texttt{ntrs,gt\textsubscript{2}tpr[where \( c=\{\#\}, \text{ simplified, OF REVSLIT'(5)} \) \text{obtain sq'' ce22e' w22' where R-CONV': ce22e' = \{#sq''\#\} + ce22e' w22' = \text{map le-rem-s w22' ([\text{entry fg q’}, \{\#\}], w22’, (sq'', ce22e’)) \in \text{trcl} \ (\text{ntrp fg}) \text{ by blast}}}

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

\texttt{from \texttt{RU-sound[OF \text{R-CONV(3) ATUV(1)}] \text{obtain Ml Me h where RU: (entry fg q, Ml, Me, h) \in \text{RU-cs fg U Ml \subseteq mon-loc fg w21 Me \subseteq mon-env fg w21 h \leq oah (map (oah fg) w21) by blast}}}

\texttt{from \texttt{RU-sound[OF \text{R-CONV(3), of V}] REVSLIT'(4) R-CONV'(1) \text{obtain Ml' Me' h' where RV: (entry fg q', Ml', Me', h') \in \text{RU-cs fg V Ml' \subseteq mon-loc fg w22' Ml' \subseteq mon-env fg w22' h' \leq oah (map (oah fg) w22') by auto}}}

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

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

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

\texttt{from \texttt{ah-interleaveable[OF REVSLIT(3)] \text{have oah (map (oah fg) w21)}}}

\(\bullet\) \texttt{ah (map (oah fg) w22)}

\texttt{thus h [\*] h'}

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

\texttt{note RU'(4)}

\texttt{also have map (oah fg) w22 \leq map (oah fg) w21 using R-CONV(2)}

\texttt{by (simp add: oah-anil)}

\texttt{hence oah (map (oah fg) w22) \leq oah (map (oah fg) w21) by (rule oah-ileq)}

106
finally show \( h \leq \alpha ah(\text{map}(\alpha n f g) w21) \).

next

note \( RV(4) \)
also have \( \text{map}(\alpha n l f g) w22' \leq \text{map}(\alpha n f g) w22 \) using \( R-CONV'(2) \)
\( \text{REVSPLIT}'(2) \) by (simp add: \( \alpha n-l[\text{symmetric}] \) le-list-map map-map[symmetric]
del: map-map)
hence \( \alpha ah(\text{map}(\alpha n f g) w22') \leq \alpha ah(\text{map}(\alpha n f g) w22) \) by (rule 
\( \alpha ah-ileq)\)
finally show \( h' \leq \alpha ah(\text{map}(\alpha n f g) w22) \).

qed

qed (simp)

moreover have \( \text{mon-fg} p \cup \text{mon-w} f g w \subseteq \text{mon-loc} f g \ (ee\#ww) \) using
\( FS-FMT(1) \) by auto
moreover have \( \text{MI} \cup \text{Me} \cup \text{Me}' \subseteq \text{mon-env} f g \ (ee\#ww) \) using \( RV(2,3) \)
\( \text{RU}(2,3) \) mon-wu-ileq[OF \( \text{REVSPLIT}'(2) \), of \( f g \)] \( \text{MON2-LEQ} \) \( R-CONV(2) \) \( R-CONV'(2) \)
\( \text{MON2-LEQ} \) by (simp add: mon-ww-of-le-rem) blast
ultimately show \( \text{?thesis} \) by blast

qed

next

case \( \text{lr} \) — The first node is reached from the local thread, the second one from
a spawned thread

from \( \text{RU-sound}(OF \ \text{DECOMP-LOC}(2), \ \text{of} \ \text{U}) \ \text{br}(1) \) \( \text{DECOMP-LOC}(1) \) obtain
\( \text{MI} \ \text{Me} \ h \) where \( \text{RU}': \ (v, \ \text{MI}, \ \text{Me}, \ h) \in \text{RU-cs} \ f g \ U \ \text{MI} \subseteq \text{mon-loc} f g \ w1 \ \text{Me} \subseteq \text{mon-env} f g \ w1 \ h \leq \alpha ah(\text{map}(\alpha n l f g) w1) \) by auto
obtain \( \text{MI}' \ \text{Me}' \ h' q' \) where \( \text{RV}': \ (\text{entry} \ f g \ q', \ \text{MI}', \ \text{Me}', \ h') \in \text{RU-cs} \ f g \ V \ \text{MI}' \subseteq \text{mon-ww} f g \ w2 \ \text{Me}' \subseteq \text{mon-ww} f g \ w2 \ h' \leq \alpha ah(\text{map}(\alpha n f g) w2) \)
proof —


case \( \text{goal1} \) — We have to extract the interesting thread from the spawned threads in
order to get an entry in \( \text{RU} \ f g \ V \)

obtain \( q' \ w2' \ c2i' \) where \( \text{REVSPLIT}': \ (\text{entry} \ f g \ q') :\ # \ w2' \leq w2 \ c2i' \leq c2' \ \text{atU} \ V \ c2i' (\{\text{entry} \ f g \ q',\ #\}\), \( w2', c2i'\} \in \text{trcl}(\text{ntrp} \ f g) \) using \( \text{ntr-reverse-split-atU}(OF - \text{br}(2) \ \text{LESPLIT}(4)) \) \( \text{ntrp-valid-preserve-s}(OF \ \text{SPLIT}(1), \ \text{simplified}) \) \( \text{CHFMT} \) by (simp add: valid-unconc) blast
from \( \text{ntrs} \text{gtr}2\text{trp}[\text{where} ' \{\#\}] \) simplified, \( \text{OF} \ \text{REVSPLIT}(5) \) obtain
\( s2i' \ c2i' \ w2' \) where \( \text{R-CONV}': \ c2i' = \{\# s2i' \#\} + c2i' w2' = \text{map} \ \text{le-rem-s} \ w2' ((\{\text{entry} \ f g \ q', \ #\}) \), \( w2', s2i', c2i'\} \in \text{trcl}(\text{ntrp} \ f g) \).

from \( \text{RU-sound}(OF \ \text{R-CONV}(3), \ \text{of} \ \text{V}) \ \text{REVSPLIT}(4) \ \text{R-CONV}(1) \) obtain
\( \text{MI}' \ \text{Me}' \ h' \) where \( \text{RV}': \ (\text{entry} \ f g \ q', \ \text{MI}', \ \text{Me}', \ h') \in \text{RU-cs} \ f g \ \text{V} \ \text{MI}' \subseteq \text{mon-loc} f g \ w2' \ \text{Me}' \subseteq \text{mon-env} f g \ w2' \ h' \leq \alpha ah(\text{map}(\alpha n l f g) w2') \) by auto
moreover have \( \text{mon-loc} f g \ w2' \subseteq \text{mon-ww} f g \ w2 \) \( \text{mon-env} f g \ w2' \subseteq \text{mon-ww} f g \ w2 \) using mon-ww-ileq[OF \( \text{REVSPLIT}(2) \), of \( f g \)] \( R-CONV(2) \) by (auto simp add: mon-ww-of-le-rem)
morerover have \( \alpha ah(\text{map}(\alpha n l f g) w2') \leq \alpha ah(\text{map}(\alpha n f g) w2) \) using
\( \text{REVSPLIT}(2) \ \text{R-CONV}(2) \) by (auto simp add: \( \alpha n-l[\text{symmetric}] \) le-list-map map-map[symmetric]
del: map-map intro: \( \alpha ah-ileq \) del: predicate2I)
ultimately show \( \text{?thesis} \) using \( \text{goal1} \ \text{REVSPLIT}(1) \) by (blast intro: order-trans) qed
from \( \text{S-ENTRY-PAT}[\{\# q'\#\}] \) simplified \( \text{RV}(1) \) have \( \text{S-ENTRY}': (v,
mon-w fg w, \{#q\} ∈ S-cs fg t by simp

have \(u, \text{mon-w fg w \cup \text{mon-w fg w \cup Me \cup Ml' \cup Me' }}\) ∈ RUV-cs fg U V
proof (rule RUV-split-le[of FS-FMT(2,4) S-ENTRY - RU(1) RV(2)])

from MON-PU MON1-LEQ MON2-LEQ RU(3) RV(3,4) show (mon-n fg u \cup \text{mon-w fg p}) \cap (Me \cup Ml' \cup Me') = \{\} by blast

next

from ah-interleavable1[of LESPLIT(1)] have ah (map (mon fg) w1) \[*\]
aah (map (\text{mon fg}) w2) by simp
thus \[*\] h\' using RV(4) RV(5) by (auto elim: ah-leq-il)
qed (simp)

moreover have mon fg p \cup \text{mon-w fg w \cup Me} \subseteq \text{mon-loc fg} \ (ee \# ww) using FS-FMT(1) MON1-LEQ RU(2) by (simp) blast

moreover have Me \cup Ml' \cup Me' ∈ \text{mon-env fg} \ (ee \# ww) using MON1-LEQ MON2-LEQ RU(3) RV(3,4) by (simp) blast
ultimately show \#thesis by blast
next

case rl — The second node is reached from the local thread, the first one from a spawned thread. This case is symmetric to the previous one

from RU-sound[of DECOMP-LOC(2), of V] rl(1) DECOMP-LOC(1) obtain Ml Me h where RV: \((v, \text{Ml}, \text{Me}, h)\) ∈ RU-cs fg V Ml \subseteq \text{mon-loc fg} w1 Me \subseteq \text{mon-env fg} w1 h \leq \text{ah} (map (\text{mon fg}) w1) by auto

obtain Ml' Me' h' q' where RU: \([\text{entry fg q'} : ]\# ch (\text{entry fg q'}, \text{Ml'}, \text{Me'}, h')\) ∈ RU-cs fg U Ml' \subseteq \text{mon-ww fg w2 Me'} \subseteq \text{mon-ww fg w2 h'} \leq \text{ah} (map (\text{mon fg}) w2) \#proof -
case goal1

— We have to extract the interesting thread from the spawned threads in order to get an entry in RU fg V

obtain q'' w2' c2'i where REVSPLIT: \([\text{entry fg q''} : ]\# ch w2' \leq w2 c2'i \leq c2' at U U c2'i \#\text{entry fg q''}\} w2', c2'i) \in \text{trcl (ntrp fg)}

using ntr-reverse-split-atU[of - rl(2) LESPLIT(4)] ntrp-valid-preserves[OF SPLIT(1), simpl] CHFMT by (simp add: valid-unconc) blast

from ntr-nfg2trwp[where c=\{\#\}, simplified, OF REVSPLIT(5)] obtain s2'i c2'ie w2' where R-CONV: \(c2'i=\{s2'i\} + c2'ie\ w2'=\text{map le-rem-s w2'} ([\text{entry fg q''} : \{\#\}], w2', s2'i, c2'ie) \in \text{trcl (ntrp fg)}\).

from RU-sound(of R-CONV(3), of U) REVSPLIT(4) R-CONV(1) obtain Ml' Me' h' where RU: \(\text{entry fg q'}, \text{Ml'}, \text{Me'}, h'\) ∈ RU-cs fg U Ml' \subseteq \text{mon-loc fg} w2' Me' \subseteq \text{mon-env fg w2' h'} \leq \text{ah} (map (\text{mon fg}) w2') by auto

moreover have mon-loc fg w2' \subseteq \text{mon-ww fg w2} \text{mon-env fg w2} \subseteq \text{mon-ww fg w2} using mon-ww-ileq[of REVSPLIT(2), of fg] R-CONV(2) by (auto simp add: mon-ww-of-le-rem)

moreover have ah (map (\text{mon fg}) w2') \leq ah (map (\text{mon fg}) w2) using REVSPLIT(2) R-CONV(2) by (auto simp add: ah-ileq del: predicate2I)
ultimately show \#thesis using goal1 REVSPLIT(1) by (blast intro: order-trans)

qed

from S-ENTRY-PAT[of \{#q\}, simplified] RU(1) have S-ENTRY: \((v, \text{mon-w fg w, \{#q\}})\) ∈ S-cs fg t by simp

have \(u, \text{mon-fg p \cup mon-w fg w \cup Ml, Me \cup Ml' \cup Me' \in RU-cs fg U V}\)
proof (rule RUV-split-el[of FS-FMT(2,4) S-ENTRY - RU(1) RU(2)])
from MON-PU MON1-LEQ MON2-LEQ RV(3) RU(3,4) show (mon-n fg u ∪ mon fg p) ∩ (Me ∪ M'l' ∪ Me') = {} by blast

next

from ah-interleave{I[OF LESPLIT(1)] have αah (map (and fg) w1) [s] αah (map (or fg) w2) by simp

thus h [s] h' using RV(4) RU(5) by (auto elim: ah-leq-il)

qed (simp)

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

moreover have Me ∪ Ml' ∪ Me' ⊆ mon-env fg (ee # ww) using MON1-LEQ MON2-LEQ RV(3) RU(3,4) by (simp) blast

ultimately show ?thesis by blast

qed

lemma (in flowgraph) RUV-precise: (u,Ml,Me)∈RUVCs fg U V

⇒ w s' c',

(((u),{#}),w,(s',c'))∈trcl (ntrp fg) ∧

atUV U V (((#s'##)+c') ∧

mon-loc fg w = Ml ∧

mon-env fg w = Ml

proof (induct rule: RUV-cs.induct)

case (RUVCall u p u' v M P Mi Me) then obtain ww s' c' where IH: (((v),{#}), ww, s', c') ∈ trcl (ntrp fg) atUV U V (((#s'##)+c') mon-loc fg w = Ml

mon-env fg w = Ml by blast

from S-precise-ntrp[OF RUVCALL(3,2,1), simplified] obtain w ch where FS: (((u),{#}), LOC (LCall p ≠ w), [v, u'], ch) ∈ ntrp fg P = {#} M = mon-w fg w mon-n fg v = mon fg p mon-c fg ch = {} by blast

note FS(1)

also have (((v),{#}), ch), ww, s' @ [u'], c' + ch) ∈ trcl (ntrp fg)

using ntrp-add-context[OF ntrp-stack-comp[OF IH(1), of [u']], of ch, simplified] FS(5) IH(4) RUV-call.hyps(6) mon-n-same-proc[OF edges-part[OF RUV-call.hyps(1)]] by simp

finally have (((u),{#}), LOC (LCall p ≠ w) ≠ ww, s' @ [u'], c' + ch) ∈ trcl (ntrp fg) .

moreover from IH(2) have atUV U V (((#s' @ [u']##)+(c'+ch)) by auto

moreover have mon-loc fg (LOC (LCall p ≠ w) ≠ ww) = mon fg p ∪ M ∪ Mi using IH(3) FS(3) by auto

moreover have mon-env fg (LOC (LCall p ≠ w) ≠ ww) = Me using IH(4)

by auto

ultimately show ?case by blast

next

case (RUVCALL Spawn u p u' v Mi Me) then obtain ww s' c' where IH: (((entry fg q),{#}), ww, s', c') ∈ trcl (ntrp fg) atUV U V (((#s'##)+c') mon-loc fg w = Ml

mon-env fg w = Ml by blast

from S-precise-ntrp[OF RUVCALL-Spawn(3,2,1), simplified] mset-sizeIelem[OF - RUVCALL-Spawn(4)]

obtain w ch where

FS: (((u),{#}), LOC (LCall p ≠ w), [v, u'], {[#entry fg q]#} + che) ∈ ntrp fg P={#q#} M = mon-w fg w mon-n fg v = mon fg p mon-c fg (((#entry fg

109
\(q\#\) + \text{che} = \emptyset \text{ by (auto elim: mset-le-addE)}

moreover

have \(((v, u'), \text{che} + \{#entry fg q\#\}), \text{map ENV (map le-rem-s \(ww\)), (v, u'), \text{che} + \{#s'\#\} + c')\in \text{trcl (ntrp fg)}\)

using ntrp2ntrp[OF grtp2grt[of IH(1)], of \([v, u']\) \text{che} IH(3,4) RUV-spawn(7) FS(4,5) mon-n-same-proc[of OF edges-part[of OF RUV-spawn(1)]]

by (auto simp add: mon-c-unconc mon-ww-of-le-rem)

ultimately have \(((u, \{\#\}), \text{LOC (LCall \(p\#\) \# \text{map ENV (map le-rem-s \(ww\)), (v, u'), \text{che} + \{#s'\#\} + c'))\in \text{trcl (ntrp fg)}\) by (auto simp add: union-ac)

moreover have \(atUV U V ((\#v,u'\#) + (che + \{#s'\#\} + c'))\) using IH(2)

by auto

moreover have mon-loc fg (LOC (LCall \(p\#\) \# \text{map ENV (map le-rem-s \(ww\})) = \text{mon-fg} p \cup M\) using FS(3) by (simp del: map-map)

moreover have mon-env fg (LOC (LCall \(p\#\) \# \text{map ENV (map le-rem-s \(ww\})) = Ml \cup Mc\) using IH(3,4) by \(\text{auto simp add: mon-ww-of-lem simp del: map-map}\)

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

next

case (RUVT-split-le u p u' v M P q Mc Me h M' Me' h')

— Get paths from precision results

from S-precise-ntrp[OF RUV-split-le(3,2,1), simplified] mset-size1elem[OF -RUVT-split-le(4)] obtain w \text{che where}

FS: \(((u, \{\#\}), \text{LOC (LCall \(p\#\) \# \text{map ENV (map le-rem-s \(ww\)), (v, u'), \{#entry fg q\#\} + \text{che}\})\in ntrp fg P = \{#q\#\} M = \text{mon-ww fg w mon-n fg v = mon fg p mon-c fg \{#entry fg q\#\} + \text{che}\}) = \emptyset\) by (auto elim: mset-le-addE)

from RUVT-precise[OF RUV-split-le(5)] obtain \(ww1 s1' c1' \text{where P1: ((v, \{\#\}), \(ww1, s1', c1' \in \text{trcl (ntrp fg)}\) atUV U ((\#s1'\#) + c1') mon-loc fg \(ww1 = Ml\) mon-env fg \(ww1 = Ml\) mon-evn fg \(ww2 = Ml'\) mon-ww fg \(ww2 = Ml'\) \text{oh} (\text{map (envl fg) \(ww2\)) = h by blast}

from RUVT-precise[OF RUV-split-le(6)] obtain \(ww2 s2' c2' \text{where P2: ((entry fg q), \{\#\}), \(ww2, s2', c2' \in \text{trcl (ntrp fg)}\) atUV V ((\#s2'\#) + c2') mon-loc fg \(ww2 = Ml'\) mon-env fg \(ww2 = Ml'\) \text{oh} (\text{map (envl fg) \(ww2\)) = h by blast}

— Get combined path from the acquisition history interleavability, need to remap loc/env-steps in second path

from \(P2(5)\) have \(\text{ah} (\text{map (envl fg) \text{map ENV (map le-rem-s \(ww2\))}) = h' by (simp add: atn-env l-assoc}

with \(P1(5)\) RUV-split-le(8) obtain \(ww \text{where IL: \(ww\in} \text{ww1\oplus}_2 \text{\text{map ENV (map le-rem-s \(ww2\))}}\) using ah-interleave2 by (force)

— Use the ntrp-unsplit-theorem to combine the executions

from ntrp-unsplit(where ca=\{\#\},OF IL P1(1) grtp2grt[of P2(1)], simplified]

have \(((v, \{#entry fg q\#\}), \(ww, s1', c1' + (\{#s2'\#\} + c2') \in \text{trcl (ntrp fg)}\)

using FS(4,5) RUV-split-le(7)

by (auto simp add: mon-c-unconc mon-ww-of-le-rem P2(3,4))

from ntrp-add-context[OF ntrp-stack-comp[of \(\text{OF this, of [u']}, [u'], \text{of che}] \text{have \(((v) \at\ [u'], \{#entry fg q\#\} + \text{che}, \(ww, s1' @ [u'], c1' + (\{#s2'\#\} + c2') + \text{che}\}) \in \text{trcl (ntrp fg)}\)

using mon-n-same-proc[of OF edges-part[of OF RUV-split-le(1)] mon-loc-cil[of OF IL, of fg] mon-env-cil[of OF IL, of fg] FS(4,5) RUV-split-le(7)] by \(\text{auto simp add: mon-c-unconc P1(3,4) P2(3,4) mon-ww-of-le-rem simp del: map-map}\)

with FS(1) have \(((u, \{\#\}), \text{LOC (LCall \(p\#\) \# \(ww, (s1' @ [u'], c1' +}

110
\[ ((\#s2\#') + c2') + (che)) \in trcl (ntrp fg) \text{ by simp} \]

moreover have atUV U V \((\#s1' @ [u]\#') + (c1' + (\#s2\#') + c2') + (che)) \]
using \( P1(2) P2(2) \) by auto

moreover have mon-loc fg (LOC (LCall p # w) # ww) = mon fg p \cup M \cup Ml
using \( FS(3) \) \( P1(3) \) mon-loc-cil[OF IL, of fg] \text{ by (auto simp del: map-map)}

moreover have mon-env fg (LOC (LCall p # w) # ww) = Me \cup Ml' \cup Me' \text{ using } P1(4) P2(3,4) mon-env-cil[OF IL, of fg] \text{ by (auto simp del: mon-ww-of-le-rem simp del: map-map)}

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

next

case (RUV-split-el u p u' v M P q Ml Me h Ml' Me' h') — This is the symmetric case to RUV-split-le, it is proved completely analogously, just need to swap \( U \) and \( V \).

— Get paths from precision results

from \( S\text{-precise-ntrp} \text{[OF RUV-split-el(3,2,1), simplified]} \) mset-size1elem[OF -
RUV-split-el(4)] obtain \( w \) che where

\[ FS: ([[u], {#}]), LOC (LCall p # w), [v, u'], {#[entry fg q][#] + (che')} \in ntrp fg P=[(#q#')] M = mon-w fg w mon-n fg v = mon fg p mon-c fg ((#entry fg q)[#] + (che')) = {} \text{ by (auto elim: mset-le-addE)} \]

from \( RU\text{-precise}[OF RUV-split-el(5)] \) obtain \( wwl s1' c1' \text{ where } P1: ([[v], {#}]), wwl, s1', c1' \in trcl (ntrp fg) atUV V ((#s1' #') + c1') mon-loc fg wwl = Ml mon-loc fg wwl = (map (entry fg q)[#]) = h' by blast

from \( RU\text{-precise}[OF RUV-split-el(6)] \) obtain \( w w2 s2' c2' \text{ where } P2: (((entry fg q)[#], {#})), w w2, s2', c2' \in trcl (ntrp fg) atUV V ((#s2' #') + c2') mon-loc fg w w2 = Ml' mon-loc fg w w2 = Me' aah (map (entry fg q)[#]) = h' by blast

— Get combined path from the acquisition history interleavability, need to remap loc/env-steps in second path

from \( P2(5) \) have aah (map (entry fg q)[#]) (map ENV (map le-rem-s w w2))) = h' by

(simp add: an-ansi a-assoc)

with \( P1(5) \) RUV-split-el(8) obtain \( w w \) where IL: \( ww \in w w1 \cup (\text{map ENV (map le-rem-s w w2)}) \text{ using ah-interleaveable2 by (force)} \)

— Use the ntrp-unsplit-theorem to combine the executions

from \( ntrp\text{-unsplit}[where cae=\{#,}\text{OF IL P1(1)} \) gtrp2gtr[OF P2(1)], simplified] \)

have \( ([[v], {#[entry fg q][#]}]), w w, s1', c1' + ((#s2' #') + c2') \in trcl (ntrp fg) \text{ using } FS(4,5) \text{ RUV-split-el(7)} \)

by (auto simp add: mon-c-unconc mon-ww-of-le-rem P2(3,4))

from \( ntrp\text{-add-context}[OF ntrp\text{-stack-comp}[OF this, of [u]], of che] \) have \( \{[v] @ [u'], {#[entry fg q][#]} + (che'), w w, s1' @ [u'], c1' + ((#s2' #') + c2') + (che) \in trcl (ntrp fg) \text{ using mon-n-same-proc[OF edges-part[OF RUV-split-el(1)]] mon-loc-cil[OF IL, of fg]} \text{ mon-env-cil[OF IL, of fg]} \text{ FS(4,5) RUV-split-el(7)} \text{ by (auto simp add: mon-c-unconc P1(3,4) P2(3,4) mon-ww-of-le-rem simp del: map-map)} \)

with \( FS(1) \) have \( \{[v], {#}\}, LOC (LCall p # w) # ww, (s1' @ [u'], c1' + ((#s2' #') + c2') + (che)) \in trcl (ntrp fg) \text{ by simp} \)

moreover have atUV U V ((#s1' @ [u]\#') + (c1' + (\#s2\#') + c2') + (che)) \text{ using } P1(2) P2(2) \text{ by auto} \]

moreover have mon-loc fg (LOC (LCall p # w) # ww) = mon fg p \cup M \cup Ml
using \( FS(3) \) \( P1(3) \) mon-loc-cil[OF IL, of fg] \text{ by (auto simp del: map-map)}

moreover have mon-env fg (LOC (LCall p # w) # ww) = Me \cup Ml' \cup Me' us-
ing $P_1(4)$ $P_2(3,4)$ mon-env-cil[OF IL, of fg] by (auto simp add: mon-ww-of-le-rem simp del: map-map)

ultimately show ?case by blast

next

case (RUV-split-ee u p u' v M P q q' Ml Me h Ml' Me' h')

— Get paths from precision results

from $S$-precise-ntrp[OF RUV-split-ee(3,2,1), simplified] mset-size2elem[OF - RUV-split-ee(4)] obtain w c where

$FS$: $([u], \{\#\})$, LOC (LCall p \# w), [v, u'], \{[#entry fg q]#\} + \{[#entry fg q']#\} + che) \in \ntrp fg P={#q#}+{#q'#} M = mon-w fg w mon-n fg v = mon fg p mon-c fg (([#entry fg q]#)+[#entry fg q']#)+che) = \{}

by (auto elim: mset-le-addE)

from RUV-split-ee(5) obtain wu1 s1' c1' where $P_1$: ([#entry fg q], \{\#\}), wu1, s1', c1') \in trcl (ntrp fg) atU U ([#s1'#] + c1') mon-loc fg wu1 = Ml mon-env fg wu1 = Me oah (map (and fg) wu1) = h by blast

from RUV-split-ee(6) obtain wu2 s2' c2' where $P_2$: ([#entry fg q'], \{\#\}), wu2, s2', c2') \in trcl (ntrp fg) atU V ([#s2'#] + c2') mon-loc fg wu2 = Ml' mon-env fg wu2 = Me' oah (map (and fg) wu2) = h' by blast

— Get interleaved paths, project away loc/env information first

from $P_1(5)$ $P_2(5)$ have oah (map (on fg) (map le-rem-s wu1)) = h oah (map (on fg) (map le-rem-s wu2)) = h' by (auto simp add: on-and o-assoc)

with RUV-split-ee(8) obtain ww where IL: ww \in (map (le-rem-s wu1))oand fg (map (le-rem-s wu2)) using ah-interleaveable2 by (force simp del: map-map)

— Use the ntr-unsplit-theorem to combine the executions

from ntr-unsplit[OF IL gtrp2gtr[OF P1(1)] gtrp2gtr[OF P2(1)], simplified] have PC: \{[#entry fg q]#, \{\#\}, \text{map ENV ww, [v, u], che + ([#s1'#] + c1') + ([#s2'#] + c2')} \} \in trcl (ntrp fg) using FS(5) by (auto simp add: mon-c-anconc)

— Prepend first step

from ntr2ntrp[OF PC(1), of [v,u'] che] have \{([v, u'], \text{che + ([#entry fg q]#} + \{[#entry fg q']#\}), \text{map ENV ww, [v, u'], che + ([#s1'#] + c1') + ([#s2'#] + c2'\})\} \in trcl (ntrp fg) by (auto simp add: union-ac)

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-anconc mon-ww-of-le-rem $P_1(3,4)$ $P_2(3,4)$)

with FS(1) have \{([v], \{\#\}), LOC (LCall p \# w) \# map ENV ww, (\{v, u\}, che + ([#s1'#] + c1' + ([#s2'#] + c2'))) \} \in trcl (ntrp fg) by (auto simp add: union-ac)

moreover have atU V ([#\{v, u\}#] + (che + ([#s1'#] + c1' + ([#s2'#] + c2')))) using $P_1(2)$ $P_2(2)$ by auto

moreover have mon-loc fg (LOC (LCall p \# w) \# map ENV ww) = mon fg p \cup M using FS(3) by auto

moreover have mon-ww fg (LOC (LCall p \# w) \# map ENV ww) = Ml \cup Me \cup Ml' \cup Me' using mon-ww-cil[OF IL, of fg] by (auto simp add: $P_1(3,4)$ $P_2(3,4)$ mon-ww-of-le-rem)

ultimately show ?case by blast

qed
10 Main Result

theory MainResult
imports ConstraintSystems
begin

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

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

- The flowgraph and the assumptions made on it in the flowgraph- and eflowgraph-locales. Note that we show in Section 6.4 that there is at least one non-trivial model of eflowgraph.
- The reference point semantics (refpoint) and the transitive closure operator (trcl).
- The definition of atUV.
- All dependencies of the above definitions in the Isabelle standard libraries.

theorem (in eflowgraph) RUV-is-sim-reach:
\( \exists w c'. (\{\#\text{entry fg (main fg)}\}\#), w, c') \in \text{trcl (refpoint fg)} \land \text{atUV U V c'} \)

The proof uses the soundness and precision theorems wrt. to normalized paths (flowgraph.RUV-sound, flowgraph.RUV-precise) as well as the normalization result, i.e. that every reachable configuration is also reachable using a normalized path (eflowgraph.normalize) and, vice versa, that every normalized path is also a usual path (ntr-is-tr). Finally the conversion between our working semantics and the semantic reference point is exploited (flowgraph.refpoint-eq).

proof
case goal1 then obtain w c' where C: (\{\#\text{entry fg (main fg)}\}\#), w, c') \in \text{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 (\{\#\text{entry fg (main fg)}\}\#), ww, c') \in \text{trcl (ntr fg)} by blast

from ntrs.gtr2gtrp[where c:=\#], simplified, OF this] obtain s' c' ww where 1: c'=(\#s'\#)+c e' ww = map le-rem-s ww l ((\{entry fg (main fg)\}, \#)), ww, s', c') \in \text{trcl (ntrp fg)} by blast

with C(2) have 2: atUV U V (\{\#s'\#\}+c') by auto

from RUV-sound[OF 1, OF 2] show \exists Ml Me. (entry fg (main fg), Ml, Me) \in RUV-cs fg U V by blast

next
case goal2 then obtain Ml Me where C: (entry fg (main fg), Ml, Me) \in RUV-cs fg U V by blast

from RUV-precise[OF C] obtain ww s' c' where P: ((\{entry fg (main fg)\}, \#)), ww, s', c') \in \text{trcl (ntrp fg)} atUV U V (\{\#s'\#\}+c') by blast
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


