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

May 28, 2015

Abstract

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

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

1 **Introduction** ............................................. 4

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

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

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

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

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

7 **Operational Semantics** .......................... 31
   7.1 Configurations and labels ....................... 31
   7.2 Monitors ........................................... 32
   7.3 Valid configurations .............................. 34
1 Introduction

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

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

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

Related Work In [6] we present a constraint-system-based analysis for programs with thread creation and procedures but without monitors. The abstraction from synchronization is common in this line of research: There are automata-based techniques [1, 2, 3] as well as constraint-system-based techniques [7, 6] to analyze programs with procedures and either parallel calls or thread creation, but without any synchronization. In [5, 4] analysis techniques for interprocedural parallel programs with a fixed number of initial threads and nested locks are presented. These nested locks are not syntactically bound to the program structure, but assumed to be well-nested, that is any unlock statement is required to release the lock that was acquired last by the thread. Moreover, there is no support for reentrant
locks\(^1\). 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: \(mon-pl\) (map \(\alpha\) \(w\)) is the set of monitors used by a word \(w\) w.r.t. the abstraction \(\alpha\)

**definition**

\[
mon-pl \, w \equiv foldl \, (op \, \cup) \, \{\} \, (map \, (\lambda e. \, (\text{fst} \, e \cup \text{snd} \, e)) \, w)
\]

**lemma** \(mon-pl-empty\![simp]: \, mon-pl \, [\] = \{\}\)

\(^1\)Reentrant locks can always be simulated by non-reentrant ones, at the cost of a worst-case exponential blowup of the program size
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
| (w ⊗ α) w = {w}
| (w ⊗ α w) = {w}

— Interleaving with the empty word results in the empty 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; 
] ⇒ 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]:

6
\[
\begin{align*}
& \text{lemma} \ cil-induct-fix \ [\text{case-names both-empty left-empty right-empty append}]: [ \\
& \quad \forall \alpha. \ P \alpha \; [],; \\
& \quad \forall \alpha \; \text{ad ae}. \ P \alpha \; []; \ (\text{ad} \; \# \; \text{ae}); \\
& \quad \forall \alpha \; \text{z aa}. \ P \alpha \; (z \; \# \; \text{aa} \); []; \\
& \quad \forall \alpha \; e1 \; w1 \; e2 \; w2. \ [ \\
& \qquad \text{fst} \; (\alpha \; e1) \; \cap \; \text{mon-pl} \; (\text{map} \; \alpha \; (e2 \; \# \; w2)) = \{\} ] \implies P \; \alpha \; w1 \; (e2 \; \# \; w2); \\
& \qquad \text{fst} \; (\alpha \; e2) \; \cap \; \text{mon-pl} \; (\text{map} \; \alpha \; (e1 \; \# \; w1)) = \{\} ] \implies P \; \alpha \; (e1 \; \# \; w1) \; w2] \\
& \implies P \; \alpha \; \text{wa} \; w1 \; w2. \\
& \end{align*}
\]

apply (induct \text{wa} \alpha \; \text{wb} \; \text{rule: cil.induct})
apply (case-tac \text{w})
apply \text{auto}
done

\textbf{lemma} cil-induct-fixα: [ \\
\quad \text{P} \; \alpha \; []; \\
\quad \forall \alpha \; \text{ad ae}. \ P \alpha \; []; \ (\text{ad} \; \# \; \text{ae}); \\
\quad \forall \alpha \; \text{z aa}. \ P \alpha \; (z \; \# \; \text{aa} \); []; \\
\quad \forall \alpha \; \text{e1} \; \text{w1} \; \text{e2} \; \text{w2}. \ [ \\
\qquad \text{fst} \; (\alpha \; \text{e1}) \; \cap \; \text{mon-pl} \; (\text{map} \; \alpha \; (\text{e2} \; \# \; \text{w2})) = \{\} ] \implies P \; \alpha \; \text{e1} \; \text{w1} \; (\text{e2} \; \# \; \text{w2}); \\
\qquad \text{fst} \; (\alpha \; \text{e2}) \; \cap \; \text{mon-pl} \; (\text{map} \; \alpha \; (\text{e1} \; \# \; \text{w1})) = \{\} ] \implies P \; \alpha \; (\text{e1} \; \# \; \text{w1}) \; \text{w2}] \\
\implies P \; \alpha \; \text{v} \; \text{w} \\
\text{apply (induct \text{v} \alpha \; \text{w} \; \text{rule: cil.induct})} \\
\text{apply (case-tac \text{w})} \\
\text{apply \text{auto}} \\
\text{done}
lemma cil-induct-fix[
it{case-names both-empty left-empty right-empty append}]: 
\[ P \alpha [] [] ; \]
\[ \forall ad \ ac. \ P \alpha [] \ (ad \neq ac) ; \]
\[ \forall z \ a. \ P \alpha (z \neq a) [] ; \]
\[ \forall e1 w1 e2 w2. \]
\[ \text{fst} (\alpha e1) \cap \text{mon-pl} (\text{map} \ \alpha (e2 \neq w2)) = {} \Longrightarrow P \alpha w1 (e2 \neq w2) ; \]
\[ \text{fst} (\alpha e2) \cap \text{mon-pl} (\text{map} \ \alpha (e1 \neq w1)) = {} \Longrightarrow P \alpha (e1 \neq w1) w2 \]
\[ \Longrightarrow P \alpha (e1 \neq w1) (e2 \neq w2) \]
\[ \Longrightarrow P \alpha \ wa \ wb \]
apply (induct wa \alpha \ wb rule: cil.induct)
apply (case-tac w)
apply auto
done

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

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

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

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

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

lemma cil-cons1: \[ [[w\in wa \otimes \alpha \wb ; \text{fst} (\alpha e) \cap \text{mon-pl} (\text{map} \ \alpha \ wb) = {}] ] \]
\[ \Longrightarrow e\neq w \in e\neq wa \otimes \alpha \ wb \]
by (cases wb) auto
2.2 Properties of consistent interleaving

— Consistent interleaving is a restriction of interleaving

**lemma cil-subset-il**: \( w \otimes \alpha \subseteq w \otimes w' \)

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

**apply** simp-all

**apply** safe

**apply** auto

**done**

**lemma cil-subset-il'**: \( w \in w_1 \otimes w_2 \implies w \in w_1 \otimes w_2 \)

**using** cil-subset-il by (auto)

— Consistent interleaving preserves the set of letters of both operands

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

**by** (induct rule: cil-set-induct-fix alpha)

**auto**

— Consistent interleaving preserves the length of both operands

**lemma cil-length**: \( \forall w \in w_1 \otimes w_2. \text{length } w = \text{length } w_1 + \text{length } w_2 \)

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

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

**lemma cil-ileq**: \( w \in w_1 \otimes w_2 \implies w_1 \preceq w \land w_2 \preceq w \)

**by** (intro conjI cil-subset-il ileq-interleave)

— Consistent interleaving is commutative and associative

**lemma cil-commute**: \( w \otimes w' = w' \otimes w \)

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

**lemma cil-assoc1**: !\\ \exists \! w_1 w_2 w_3. \[ w \in w_1 \otimes w_2 \implies \exists \! w. w \in w_1 \otimes w_2 \otimes w_3 \]

**proof** (induct 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') **with** Cons.prems(2) **have** e#w' \in w_1 \otimes w_2 **by** simp

**thus** ?thesis **proof** (cases rule: cil-cons-cases[case-names left' right'])

**case** (left' w')

**from** Cons.hyps[OF - left(2) left'(2)] **obtain** wr where IHAPP: \( w \in w_1' \otimes w_2 \)

**obtain** e#w \in \# w_1' \otimes w_2 **proof** (rule cil-cons[OF IHAPP(1)])

**from** left left' cil-mon-pl[OF IHAPP(2)] **show** \( \text{fst } (\alpha e) \cap \text{mon-pl } (\text{map } \alpha \)

**
\[ \text{IHAPP} \]

\[ \text{wr} \] = \{ \} by auto

\text{qed}

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

next

\text{case (right' w2')} from \text{Cons.hyps}[\text{OF - left'}(2) right'(2)] obtain \text{wr} where

\text{IHAPP}: \ w \in w_1 \otimes_{\alpha} \text{wr} \ w \in w_2' \otimes_{\alpha} w_3 by blast

from \text{IHAPP}(2) left have \ e \# \text{wr} \in e \# w_2' \otimes_{\alpha} w_3 by (auto intro: cil-cons1)

moreover from \text{right'} \text{IHAPP}(1) have \ e \# w \in w_1 \otimes_{\alpha} e \# \text{wr} by (auto intro: cil-cons2)

ultimately show \text{thesis using right'} by blast

qed

next

\text{case (right w3')} from \text{Cons.hyps}[\text{OF - right'}(2) \text{Cons.prems}(2)] obtain \text{wr} where

\text{IHAPP}: \ w \in w_1 \otimes_{\alpha} \text{wr} \ w \in w_2 \otimes_{\alpha} w_3' by blast

from \text{IHAPP}(2) right cil-mon-pl[\text{OF Cons.prems}(2)] have \ e \# \text{wr} \in w_2 \otimes_{\alpha} e \# w_3' by (auto intro: cil-cons2)

moreover from \text{IHAPP}(1) right cil-mon-pl[\text{OF Cons.prems}(2)] have \ e \# w \in w_1 \otimes_{\alpha} e \# \text{wr} by (auto intro: cil-cons2)

ultimately show \text{thesis using right} by blast

qed

\text{qed}

\text{lemma cil-assoc2:}

assumes \( A: \ w \in w_1 \otimes_{\alpha} \text{wr} \) and \( B: \ w \in w_2 \otimes_{\alpha} w_3 \)

shows \( \exists w. \ w \in w_1 \otimes_{\alpha} w_3 \land w \in w_1 \otimes_{\alpha} w_2 \)

\text{proof –}

from \text{A have} \( A': \ w \in w \otimes_{\alpha} w_1 \) by (simp add: cil-commute)

from \text{B have} \( B': \ w \in w \otimes_{\alpha} w_2 \) by (simp add: cil-commute)

from cil-assoc1[\text{OF A' B'}] obtain \text{wl} where \( w \in w_3 \otimes_{\alpha} w_1 \land w_1 \in w_2 \otimes_{\alpha} w_1 \)

by blast

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

\text{qed}

— Parts of the abstraction can be moved to the operands

\text{lemma cil-map:} \ w \in w_1 \otimes_{(\alpha f)} w_2 \Longrightarrow \text{map f w} \in \text{map f w} \otimes_{\alpha} \text{map f w} \o _2

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

\text{case empty thus \{case by auto}

next

\text{case (left e w' w1' w2)}

have \( f e \# \text{map f w'} \in f e \# \text{map f w1'} \otimes_{\alpha} \text{map f w2} \) proof (rule cil-cons1)

from left(2) have \( \text{fst}( (\alpha f) f) \cap \text{mon-pl} (\text{map } \alpha (\text{map f w2})) = \{ \} \) by (simp only: map-map[symmetric])

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

\text{qed (rule left(3))}

thus \text{thesis by simp}

next

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

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

end

3 Acquisition Histories

theory AcquisitionHistory
imports ConsInterleave
begin

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

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

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

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

3.1 Definitions

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

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

\textbf{lemma} \( ah\)-cases\([cases \ set]: [h \in ah; h m = \{} \Longrightarrow P ; m \in h m \Longrightarrow P] \Longrightarrow P \)

\textbf{by} (\textit{unfold} \( ah\)-\textit{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 \( m1 \).

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

\textbf{where} \( h1 \ [\ast] h2 \equiv \neg(\exists m1 \ m2. \ m1 \in h2 m2 \land m2 \in h1 m1) \)

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

\textbf{lemma} \( ah-il\)-lockset-disjoint:

\( [h1 \in ah; h2 \in ah; h1 \ [\ast] h2] \Longrightarrow h1 m = \{} \lor h2 m = \{} \)

\textbf{by} (\textit{unfold} \( ah-il\)-\textit{def}) (\textit{auto elim:} \( ah\)-\textit{cases})

Of course, acquisition history interleavability is commutative

\textbf{lemma} \( ah-il\)-commute: \( h1 \ [\ast] h2 \Longrightarrow h2 \ [\ast] h1 \)

\textbf{by} (\textit{unfold} \( ah-il\)-\textit{def}) \textit{auto}

3.3 Used monitors

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

\textbf{definition} \( mon-ah :: ('m \Rightarrow 'm set) \Rightarrow 'm set \)

\textbf{where} \( mon-ah h \equiv \bigcup \{ h(m) \mid m. \ True \} \)
3.4 Ordering

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

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

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

lemma ah-leq-il: \[ h1; h1' \leq h1; h2' \leq h2 \] = \[ h1' h2' \]

by (unfold ah-il-def le-fun-def [where 'b='a set]) blast+

lemma ah-leq-il-left: \[ h1 h2; h1' \leq h1 \] = \[ h1' h2 \]

ah-leq-il-right: \[ h1 h2; h2' \leq h2 \]

by (unfold ah-il-def le-fun-def [where 'b='a set]) blast+

3.5 Acquisition histories of executions

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

primrec α ah :: ('m set × 'm set) list ⇒ 'm ⇒ 'm set where

α ah [] m = {}

| α ah (e#w) m = (if m∈fst e then fst e ∪ snd e ∪ mon-pl w else α ah w m)

— α ah generates valid acquisition histories

lemma α ah-ah: α ah w ∈ ah

apply (induct w)

apply (unfold ah-def)

apply simp

apply (fastforce split: split-if-asm)

done

lemma α ah-hd: [m∈fst e; x∈fst e ∪ snd e ∪ mon-pl w] =⇒ x∈α ah (e#w) m

by auto

lemma α ah-tl: [m∉fst e; x∈α ah w m] =⇒ x∈α ah (e#w) m

by auto

lemma α ah-cases[cases set, case-names hd tl]: [ x∈α ah w m; ![e w'. [w=e#w'; m∈fst e; x∈fst e ∪ snd e ∪ mon-pl w'] =⇒ P; ![e w'. [w=e#w'; m∉fst e; x∈α ah w' m] =⇒ P ] =⇒ P ]

by (cases w) (simp-all split: split-if-asm)

lemma α ah-cons-cases[cases set, case-names hd tl]: [ x∈α ah (e#w') m; [m∈fst e; x∈fst e ∪ snd e ∪ mon-pl w'] =⇒ P; [m∉fst e; x∈α ah w' m] =⇒ P ]

13
\[ \Rightarrow P \]

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

\textbf{lemma} \textit{mon-ah-subset}: \textit{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

\textbf{lemma} \textit{\(\alpha \text{ah-ileq}: w_1 \preceq w_2 \Rightarrow \alpha \text{ah } w_1 \leq \alpha \text{ah } w_2\)}

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

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

\textbf{next}

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

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

fix \(m x\)

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

with \(\text{drop}(2)\) \textbf{have} \(x \in \alpha \text{ah } l \ m\) \textbf{by} \((\text{unfold le-fun-def [where } 'b'= 'a set], auto})\)

\textbf{moreover hence} \(x \in \text{mon-pl } l\) \textbf{using} \(\text{mon-ah-subset [unfolded mon-ah-def] by fast\)}

ultimately \textbf{show} \(x \in \alpha \text{ah}\ (a \# l) \ m\) \textbf{by auto\)

\textbf{qed}

\textbf{next}

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

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

fix \(m x\)

assume \(A: x \in \alpha \text{ah } (a \# l') \ m\)

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

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

\textbf{case} \(\text{hd}\)

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

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

\textbf{next}

\textbf{case} \(\text{tl}\)

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

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

\textbf{qed}\)

\textbf{next}\)

\textbf{qed}\)

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

\textbf{lemma} \textit{ah-interleavable1}: \(w \in w_1 \otimes_\alpha w_2 \Rightarrow \alpha \text{ah}\ (\text{map } \alpha w_1) \ [\ast]\ \alpha \text{ah}\ (\text{map } \alpha w_2)\)

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

\textbf{proof} \((\text{induct } w \alpha w_1 w_2 \text{ rule: cil-set-induct-fixa})\)

\textbf{case} empty \textbf{show} \(?\text{case}\) \((\text{simp add: ah-il-def})\) — The base case is trivial by the definition of \(\text{op}\ [\ast]\)

\textbf{next}

— Case: First step comes from the left word
case (left e w' w1' w2) show ?case

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

lemma ah-interleavable2:
assumes A: \( \alpha h (\map \alpha w1) [\star] \alpha h (\map \alpha w2) \)
shows \( w1 \otimes_\alpha w2 \neq {} \)
— This lemma is shown by induction on the sum of the word lengths
proof
— To apply this induction in Isabelle, we have to rewrite the lemma a bit
{ fix n
have !!w1 w2. \[ \[ \alpha h (\map \alpha w1) [\star] \alpha h (\map \alpha w2); n=\text{length } w1 + \text{length } w2 \] \[ \[ \Rightarrow \] w1 \otimes_\alpha w2 \neq {} \]
proof (induct n rule: nat-less-induct [case_names I])
— We first rule out the cases that one of the words is empty
case (I n w1 w2) show ?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
— In this case, we check whether the first step of one of the words can
safely be executed without blocking any steps of the other word
show ?thesis proof (cases fst (\( \alpha e1 \)) \[ \cap \] mon-pl (\( \map \alpha w2 \)) = {})
— The first step of the first word can safely be executed
— From the induction hypothesis, we get that there is a consistent
interleaving of the rest of the first word and the second word
have w1' \[ \otimes_\alpha w2 \neq {} \] proof
— From the induction hypothesis, we get that there is a consistent
interleaving of the rest of the first word and the second word
have w1' \[ \otimes_\alpha w2 \neq {} \] proof
— From the induction hypothesis, we get that there is a consistent
interleaving of the rest of the first word and the second word
by fast
moreover from CONS1 I.prems(2) have length w1' + length w2 < n
by simp
ultimately show ?thesis using I.hyps by blast
qed
— And because the first step of the first word can be safely executed, we
can prepend it to that consistent interleaving
with cil-cons1[\( \alpha h - \text{True} \)] CONS1 show ?thesis by blast
next
case False note C1=this
show ?thesis proof (cases fst (\( \alpha e2 \)) \[ \cap \] mon-pl (\( \map \alpha w1 \)) = {})
— The first step of the second word can safely be executed
— This case is shown analogously to the latter one
have \( w_1 \otimes_{\alpha} w_2' \not= \{ \} \) proof

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

by fast

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

ultimately show \(?\text{thesis}\) using \( \text{I.hyps} \) by blast

next
case False note \( C2 = \text{this} \) — Neither first step can safely be executed.

This is exactly the situation from that we can extract a conflicting pair

- from \( \text{CONS1} C2 \) obtain \( m_1 \) \( m_2 \) where \( m_1 \in \text{fst} \) \((\alpha \) e1 \) \(m_1 \in \text{mon-pl} \) \((\text{map } \alpha \) \(w_2) \) \(m_2 \in \text{fst} \) \((\alpha \) e2 \) \(m_2 \in \text{mon-pl} \) \((\text{map } \alpha \) \(w_1) \) by blast

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

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

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

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

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

\textbf{theorem} \( \alpha \)-interleavable:

\((\alpha \)ah \((\text{map } \alpha \) \(w_1) [\ast] \alpha \)ah \((\text{map } \alpha \) \(w_2)\) \(\leftarrow\rightarrow (w_1 \otimes_{\alpha} w_2 \not= \{\})\))

\textbf{using} \( \alpha \)-interleavable1 \( \alpha \)-interleavable2 by blast

\textbf{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.

\textbf{definition}

\( \text{ah-update} :: \) \(('m \Rightarrow \text{set}) \Rightarrow (\text{'m set } \ast \text{'m set}) \Rightarrow \text{'m set } \ast \) \(('m \Rightarrow \text{'m set})\)

where

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

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:** \( ah (e \# w) = ah-update (\alpha ah) e \) (mon-pl \( w \))

by (auto intro!: ext simp add: ah-update-def)

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

**lemma ah-update-mono:** \[ \[ h \leq h', F = F', M \subseteq M' \] \] \( \Rightarrow \) ah-update \( h \) \( F \) \( M \) \( \leq \) ah-update \( h' \) \( F' \) \( M' \)

by (auto simp add: ah-update-def le-fun-def [where 'b= 'a set!])

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

by (auto simp add: ah-update-def le-fun-def [where 'b= 'a set!])

end

4 Labeled transition systems

theory LTS
imports Main
begin

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

4.1 Definitions

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

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

Transitive reflexive closure

**inductive-set**

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

for \( t \)

where

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

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

4.2 Basic properties of transitive reflexive closure

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

by (auto elim: trcl.cases)

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

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

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

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

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

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

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

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

lemma trcl-unconcat: \!! c . \[(c,w1@w2,c')\in\text{trcl t}] \Rightarrow \exists \text{ch} . (c,w1,\text{ch})\in t \land (\text{ch},w2,c')\in\text{trcl t}\)  
proof (induct w1)  
  case Nil hence \((c,[]\in\text{trcl t} \land (c,w2,c')\in\text{trcl t}) by \text{auto}\)  
thus \(?case by \text{fast}\) next

  case (Cons a w1) note \text{IHP} = this  
  hence \((c,a\#(w1@w2),c')\in\text{trcl t}) by \text{simp}\nwith trcl-uncons obtain chh where \((c,a,\text{chh})\in t \land (\text{chh},w1@w2,c')\in\text{trcl t}) by \text{fast}\n
moreover with \text{IHP} obtain ch where \((\text{chh},w1,\text{ch})\in\text{trcl t} \land (\text{ch},w2,c')\in\text{trcl t}) by \text{fast}\n
ultimately have \((c,a\#w1,\text{ch})\in\text{trcl t} \land (\text{ch},w2,c')\in\text{trcl t}) by \text{auto}\nthus \(?case by \text{fast}\) qed
4.2.1 Appending of elements to paths

**lemma** trcl-rev-cons: \( (c, e, c') \in \text{trcl } T \) \( \Rightarrow \) \( (c, w \oplus [e], c') \in \text{trcl } T \)
by (auto dest: trcl-concat iff add: trcl-single)

**lemma** trcl-rev-uncons: \( (c, w \oplus [e], c') \in \text{trcl } T \)
\( \Rightarrow \exists ch. (c, w, ch) \in \text{trcl } T \land (ch, e, c') \in T \)
by (force dest: trcl-unconcat)

**lemma** trcl-rev-induct\[\text{induct set, consumes 1, case-names } \text{empty snoe}]: \![c', e]. [\]
\( (c, w, c') \in \text{trcl } S \)
\( \Rightarrow \) \( P \)
by (induct w rule: rev-induct) (auto dest: trcl-rev-uncons)

**lemma** trcl-rev-cases: \![c, e, c']. [\]
\( (c, w, c') \in \text{trcl } S \)
\( \Rightarrow \) \( P \)
by (induct w rule: rev-induct) (simp, blast dest: trcl-rev-uncons)

**lemma** trcl-cons2: \![c, e, c']. [\]
\( c, (e, f, c') \in T \)
\( \Rightarrow \) \( (c, [e, f], e') \in \text{trcl } T \)
by auto

4.2.2 Transitivity reasoning setup

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

**declare** cons\[trans\]
**declare** trcl-concat\[trans\]
**declare** trcl-rev-cons\[trans\]

4.2.3 Monotonicity

**lemma** trcl-mono: \(!A B. A \subseteq B \Rightarrow \text{trcl } A \subseteq \text{trcl } B\)\napply (clarsimp)
apply (erule trcl.induct)
apply auto
done

**lemma** trcl-inter-mono: \( x \in \text{trcl } (S \cap R) \Rightarrow x \in \text{trcl } S \land x \in \text{trcl } R \)
proof
assume \( x \in \text{trcl } (S \cap R) \) show \( x \in \text{trcl } S \) by auto
next
assume \( x \in \text{trcl } (S \cap R) \) show \( x \in \text{trcl } R \) by auto
qed
4.2.4 Special lemmas for reasoning about states that are pairs

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

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

4.2.5 Invariants

**lemma** trcl-prop-trans[cases set, consumes 1, case-names empty steps]:

\[
\begin{align*}
(c,w,c') \in \text{trcl } S; \\
[\text{c=c'; \ w=[]}] \implies P; \\
[\text{c} \in \text{Domain } S; \text{c} \in \text{Range (Range } S)] \implies P
\end{align*}
\]

apply (erule-tac trcl-rev-cases)
apply auto
apply (erule trcl.cases)
apply auto
done

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**

\[
gtr gtrs \equiv \{ (\#s\#) + c, e, \{\#s'\#\} + c' | (s,c), e, (s',c') \in gtrs \}
\]

**lemma** gtrl-s: ((s,c), e, (s',c')) \in gtrs \implies (\#s\#) + c, e, \{\#s'\#\} + c' \in gtr gtrs

by (unfold gtr-def, auto)

**lemma** gtrl: ((s,c), w, (s',c')) \in gtr gtrs

\implies (\#s\#) + c, w, \{\#s'\#\} + c' \in \text{trcl (gtr gtrs)}

by (induct rule: trcl-pair-induct) (auto dest: gtrl-s)
lemma \textit{gtrE}: \[\begin{array}{l}
(c, e, c') \in \text{gtr } T; \\

\text{!!s ce s' ce'. } [ c = \{\#s\#\} + ce; c' = \{\#s'\#\} + ce'; ((s, ce), e, (s', ce')) \in gtrs] \implies P \\
\end{array}\]

by (unfold gtr-def) blast

lemma \textit{gtr-empty-conf-s[simp]}:

\((\#), w, c') \in \text{gtr } S \rightarrow ((\#), w, c') \in \text{gtr } S \rightarrow (w = \[] \wedge c' = \{\}) \rightarrow (w = \[] \wedge c = \{\})

by (auto elim: gtrE)

lemma \textit{gtr-empty-conf1[simp]}:

\(((\#), w, c') \in \text{trcl } (\text{gtr } S) \rightarrow ((\#), w, c') \in \text{trcl } (\text{gtr } S) \rightarrow (w = \[] \wedge c' = \{\})

by (induct w) (auto dest: trcl-unscons)

lemma \textit{gtr-empty-conf2[simp]}:

\(((c, w, \{\}) \in \text{gtr } S) \rightarrow ((c, w, \{\}) \in \text{gtr } S) \rightarrow (w = \[] \wedge c = \{\})

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

lemma \textit{gtr-find-thread}:

\((c, e, c') \in \text{gtr } gtrs; \\

\text{!!s ce s' ce'. } [ c = \{\#s\#\} + ce; c' = \{\#s'\#\} + ce'; ((s, ce), e, (s', ce')) \in gtrs] \implies P \\
\end{array}\]

by (unfold gtr-def) auto

lemma \textit{gtr-step-cases[cases set, case-names loc other]}:

\((\{\#s\#\} + ce, e, c') \in \text{gtr } gtrs; \\

\text{!!s' ce'. } [ c' = \{\#s'\#\} + ce'; ((s, ce), e, (s', ce')) \in gtrs] \implies P; \\

\text{!!cc ss ss' ce'. } [ ce = \{\#ss\#\} + cc; c' = \{\#ss'\#\} + ce'; ((ss, \{\#ss\#\} + cc), e, (ss', \{\#ss'\#\} + cc)) \in gtrs] \implies P \\
\end{array}\]

by (auto elim!: gtr-find-thread mset-single-cases)

lemma \textit{gtr-rev-cases[cases set, case-names loc other]}:

\((c, e, \{\#s\#\} + ce) \in \text{gtr } gtrs; \\

\text{!!s ce. } [ c = \{\#s\#\} + ce; ((s, ce), e, (s', ce')) \in gtrs] \implies P; \\

\text{!!cc ss ss' ce. } [ c = \{\#ss\#\} + cc; ce' = \{\#ss'\#\} + cc; ((ss, ce), e, (ss', \{\#ss'\#\} + cc)) \in gtrs] \implies P \\
\end{array}\]

by (auto elim!: gtr-find-thread mset-single-cases)

5.2 Invariants

lemma \textit{gtr-preserve-s}:

\((c, e, c') \in \text{gtr } T; \\

P \rightarrow \text{!!s e s' c' e'. } [P (\{\#s\#\} + c); ((s, ce), e, (s', ce')) \in T] \implies P (\{\#s'\#\} + c') \\
\end{array}\]

by (unfold gtr-def) blast

lemma \textit{gtr-preserve}:

\((c, w, c') \in \text{trcl } (\text{gtr } T); \\

P \rightarrow \text{!!s e s' c' e. } [P (\{\#s\#\} + c); ((s, ce), e, (s', ce')) \in T] \implies P (\{\#s'\#\} + c') \\
\end{array}\]

P \rightarrow \text{!!s e s' c' e. } [P (\{\#s\#\} + c); ((s, ce), e, (s', ce')) \in T] \implies P (\{\#s'\#\} + c') \\
\end{array}\]

by (unfold gtr-def) blast
\[ P \left( \{ \#s\# \} + c \right); (s, c, e, (s', c')) \in T \] \implies P \left( \{ \#s'\# \} + c' \right)

---

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]:
  \[ \left( (s, c), e, (s', c') \right) \in gtrs; \#csp. c' = csp + c \implies P \right] \implies P

---

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'#} + cc) ∈ gtrs gtrs and
LOC: !s ce. \[ c = \{#s#'\} + cc; ((s, cc), e, (s', ce')) \in gtrs ] \implies P and
SPAWN: !ss ss' ce csp.
\[ c = \{#ss#\} + cc; ce' = \{#ss#'\} + csp + ce; ((ss, cc), e, (ss', {#s#'}) + csp + ce) \in gtrs ]
\implies P and
ENV: !ss ss' ce csp.
\[ c = \{#ss#\} + \{#s#'\} + cc; ce' = \{#ss#'\} + csp + ce; ((ss, \{#s#'\} + ce), e, (ss', csp + \{#s#'\} + ce)) \in gtrs ]
\implies P

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': \$ = \{#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-unplusm-dist-cases)

case left note CC=this
with CASE' have ce' = \{#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#'\} + (cc - \{#s#'\}) by (simp add: union-assoc)
moreover from CC(2) CASE'(2) have ce' = \{#ss#'\} + csp + (cc - \{#s#'\}) by (simp add: union-assoc)
moreover from CC(2) have \{#s#'\} + cc = csp + \{#s#'\} + (cc - \{#s#'\}) by (simp add: union-ac)
ultimately show \$thesis using CASE'(3) CASE(3) CC(1) ENV by auto
qed

5.4 Explicit local context

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

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

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

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

5.4.1 Lifted step datatype

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

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

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

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

Standard simplification lemmas

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

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

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

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

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

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

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

**Lemma le-rem-simps [simp]:**

le-rem-s (LOC a) = a
le-rem-s (ENV a) = a
by (unfold le-rem-s-def, auto)

**Lemma le-rem-id-simps [simp]:**

le-rem-s ◦ LOC = id
le-rem-s ◦ ENV = id
by (auto intro: ext)

**Lemma le-rem-id-map [simp]:**

map le-rem-s (map LOC w) = w
map le-rem-s (map ENV w) = w
by auto

**Lemma env-map-env [simp]:**

env (map ENV w) = map ENV w
by (unfold env-def)

**Lemma env-map-loc [simp]:**

env (map LOC w) = []
by (unfold env-def)

**Lemma loc-map-env [simp]:**

loc (map ENV w) = []
by (unfold loc-def)

**Lemma loc-map-loc [simp]:**

loc (map LOC w) = map LOC w
by (unfold loc-def)

5.4.2 Definition of the loc/env-semantics

type-synonym 's el-conf = ('s × 's multiset)

inductive-set

gtrp :: ('s el-conf, 'l) LTS ⇒ ('s el-conf, 'l el-step) LTS
for S
where

gtrp-loc: ((s,c),e, (s',c')) ∈ S ⇒ ((s,c),LOC e, (s',c')) ∈ gtrp S
| gtrp-env: ((s, {#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**

((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#)+((#s#)+c), e, (#ss' #)+((#s#)+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 gtrp2gtr:**

\[(s,c), w.(s',c') \in \mathsf{trcl}(\mathsf{gtrp} T) \rightarrow ((\#s') + c, \mathsf{map}\ \mathsf{le-rem-s} w.(\#s' + c') \in \mathsf{trcl}(\mathsf{gtr} T)\]

**by** (induct rule: \(\mathsf{trcl-pair-induct}\)) (auto dest: \(\mathsf{gtrp2gtr}\))

**lemma** (in \(\mathsf{env-no-step}\)) \(\mathsf{gtrp2gtr}\) [cases set, case-names \(\mathsf{gtrp}\)]:

assumes \(A:((\#s') + c, w.(s', c') \in \mathsf{gtr}\) gtrs\)

and CASE: \(!!s' ce' ee. (c' = (\#s') + ce' ; w = \mathsf{le-rem-s} ee; ((s,c), ee.(s', ce')) \in \mathsf{gtrp} \) gtrs\]

shows \(P\)

using \(A\)

**proof** (cases rule: \(\mathsf{gtr-step-cases}\))

**case** \((\text{loc } s' ce')\) hence \(((s,c), \mathsf{LOC}\ e.(s', ce')) \in \mathsf{gtrp} \) gtrs by \(\text{blast intro: gtrp-loc}\)

with \(\text{loc}(1)\) show \(\text{?thesis}\) by \(\text{rule-tac CASE}\) auto

**next**

case \((\text{other cc ss ss'} cc')\) from \(\mathsf{env-no-step-s[OF other(3)]}\) obtain \(\mathsf{csp}\) where \(\mathsf{CE'}\) FMT: \(c' = \mathsf{csp} + ((\#s') + cc)\).

with \(\text{other(3)}\) have \(((ss,\#s') + cc), (ss',\#s') + (cc + (csp + cc)) \in \mathsf{gtrp}\) by (auto simp add: union-ac)

from \(\mathsf{gtrp-loc[OF this]}\) \(\text{other(1)}\) have \(((s,c), \mathsf{ENV}\ e, s, (\#ss') + (cc + (csp + cc)) \in \mathsf{gtrp}\) gtrs by simp

moreover from other \(\mathsf{CE'}\) FMT have \(c' = (\#s') + (\#ss' + cc))\) by (simp add: union-ac)

ultimately show \(\text{?thesis}\) by \(\text{rule-tac CASE}\) auto

**qed**

**lemma** (in \(\mathsf{env-no-step}\)) \(\mathsf{gtrp2gtr}\) [cases set, case-names \(\mathsf{gtrp}\)]:

assumes \(A:((\#s') + c, w.(c', w) \in \mathsf{trcl}(\mathsf{gtrp} gtrs)\)

and CASE: \(!!s' ce' w w. (c' = (\#s') + ce' ; w = \mathsf{le-rem-s} w w; ((s,c), w w.(s', ce')) \in \mathsf{trcl}(\mathsf{gtrp} gtrs)\]

shows \(P\)

**proof**

have \(!!s c. ((\#s') + c, w. c') \in \mathsf{trcl}(\mathsf{gtrp} gtrs) \rightarrow \exists s' ce' w w. (c' = (\#s') + ce' \wedge w = \mathsf{le-rem-s} w w \wedge ((s,c), w w.(s', ce')) \in \mathsf{trcl}(\mathsf{gtrp} gtrs)\)

**proof** (induct \(w\))

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

**next**

case \((\text{Cons } c w)\) then obtain \(\text{ch}\) where \(\text{SPLIT}: ((\#s') + c, e, c) \in \mathsf{trcl}(\mathsf{gtrp} gtrs) (c h, c') \in \mathsf{trcl}(\mathsf{gtrp} gtrs)\) by \(\text{fast dest: trcl-unccons}\)

from \(\mathsf{gtrp2gtr-s[OF SPLIT(1)]}\) obtain \(\text{sh ceh ee}\) where \(\text{FS}: c h = (\#sh) + c e h = \mathsf{le-rem-s} ee ((s, c), ee, sh, ceh) \in \mathsf{gtrp} gtrs\) by blast

moreover from \(\text{FS(1) SPLIT(2)}\) \(\text{Cons.hyps}\) obtain \(s' ce' w w\) where \(\text{IH}: c' = (\#s') + ce' w = \mathsf{le-rem-s} w w (\text{((sh, ceh, ee, s', ce'))} \in \mathsf{trcl}(\mathsf{gtrp} gtrs)\) by blast

ultimately have \(((s,c), ee, w w.(s', ce')) \in \mathsf{trcl}(\mathsf{gtrp} gtrs) e \# w = \mathsf{map}\ \mathsf{le-rem-s}\ (ee \# w)\) by auto

with \(\text{IH(1)}\) show \(\text{?case by iprover}\)

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

5.4.4 Invariants

lemma gtrp-preserve-s:
  assumes A: \(((s,c),e,(s',c'))\)\in gtrp T
  and INIT: P \((\#s\#)+c\)
  and PRES: !s c s' c' e. \[P \((\#s\#)+c\); ((s,c),e,(s',c'))\in T\]
  shows P \((\#s'\#)+c'\)
proof –
  from gtrp-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'))\in T\]
  shows P \((\#s'\#)+c'\)
proof –
  from gtrp-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**

\[ (p, ba) \text{edgeAnnot} = \text{Base 'ba | Call 'p | Spawn 'p} \]

**type-synonym**

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

**record**

\[ (n, p, ba, m) \text{flowgraph-rec} = \]

- **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**

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

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

**locale** **flowgraph** =

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

### 6.2 Basic properties

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

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

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

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

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

6.3 Extra assumptions for flowgraphs

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

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

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

definition
isolated-ret fg p ==
(\forall u l. \neg(u,l,return fg p)\in edges fg) \land entry fg p \neq return fg p

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

locale eflowgraph = flowgraph +
— Initial procedure’s entry node isn’t equal to its return node
assumes initial-no-ret: initialproc fg p \implies entry fg p \neq return fg p
— The only outgoing edges of initial procedures’ entry nodes are call edges to procedures with isolated return node
assumes initial-call-no-ret: [initialproc fg p; (entry fg p,l,v)\in edges fg]
\implies \exists p'. l=Call p' \land isolated-ret fg p'
6.4 Example Flowgraph

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

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

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

end

7 Operational Semantics

theory Semantics imports Main Flowgraph ~~/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.

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

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

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

7.2 Monitors

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

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

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

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

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

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

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

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

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

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

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

by (simp-all add: mon-e-def)

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

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

**lemma**
\[ mon-s-alt: mon-s \ fg \ s =\bigcup \{ \ mon fg \ ′ proc-of \ fg \ ′ set \ s \} \]

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

**lemma**
\[ mon-c-alt: mon-c \ fg \ c =\bigcup \{ mon-s \ fg \ ′ set-of \ c \} \]

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

**lemma**
\[ mon-w-alt: mon-w \ fg \ w =\bigcup \{ mon-e \ fg \ ′ set \ w \} \]

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

**lemma**
\[ mon-sI: \[n\in set \ s; m\in mon-n \ fg \ n\] ⇒ m\in mon-s \ fg \ s \]

by (unfold mon-s-def, auto)

**lemma**
\[ mon-sD: m\in mon-s \ fg \ s ⇒ \exists n\in set \ s . m\in mon-n \ fg \ n \]
lemma mon-n-same-proc:
  \( \text{proc-of } fg \ n = \text{proc-of } fg \ n' \iff \text{mon-n } fg \ n = \text{mon-n } fg \ n' \)

by (unfold mon-n-def, simp)

lemma mon-s-same-proc:
  \( \text{proc-of } fg ' \ \text{set } s = \text{proc-of } fg ' \ \text{set } s' \iff \text{mon-s } fg \ s = \text{mon-s } fg \ s' \)

by (unfold mon-s-alt, simp)

lemma (in flowgraph) mon-of-entry[simp]:
  \( \text{mon-n } fg (\text{entry } fg \ p) = \text{mon } fg \ p \)

by (unfold mon-n-def, simp add: entry-valid)

lemma (in flowgraph) mon-of-ret[simp]:
  \( \text{mon-n } fg (\text{return } fg \ p) = \text{mon } fg \ p \)

by (unfold mon-n-def, simp add: return-valid)

lemma mon-c-single[simp]:
  \( \text{mon-c } fg \ \{\#s\#\} = \text{mon-s } fg \ s \)

by (unfold mon-c-def) auto

lemma mon-s-single[simp]:
  \( \text{mon-s } fg \ [n] = \text{mon-n } fg \ n \)

by (unfold mon-s-def) auto

lemma mon-s-empty[simp]:
  \( \text{mon-s } fg \ [] = \{\} \)

by (unfold mon-s-def) auto

lemma mon-c-empty[simp]:
  \( \text{mon-c } fg \ \{\} = \{\} \)

by (unfold mon-c-def) auto

lemma mon-s-unconc:
  \( \text{mon-s } fg (a@b) = \text{mon-s } fg a \cup \text{mon-s } fg b \)

by (unfold mon-s-def) auto

lemma mon-s-uncons[simp]:
  \( \text{mon-s } fg (a#as) = \text{mon-n } fg a \cup \text{mon-s } fg as \)

by (rule mon-s-unconc[where a=[a], simplified])

lemma mon-c-unconc:
  \( \text{mon-c } fg (a+b) = \text{mon-c } fg a \cup \text{mon-c } fg b \)

by (unfold mon-c-def) auto

lemma mon-cl: \( [s:#c; m\in\text{mon-s } fg \ s] \Rightarrow m\in\text{mon-c } fg \ c \)

by (unfold mon-c-def, auto)

lemma mon-cD: \( [m\in\text{mon-c } fg \ c] \Rightarrow \exists \ s. s:#c \land m\in\text{mon-s } fg \ s \)

by (unfold mon-c-def, auto)

lemma mon-s-mono: set s \subseteq set s' \Rightarrow \text{mon-s } fg \ s \subseteq \text{mon-s } fg \ s'

by (unfold mon-s-def) auto

lemma mon-c-mono: c\leq c' \Rightarrow \text{mon-c } fg c \subseteq \text{mon-c } fg c'

by (unfold mon-c-def) (auto intro: mset-le-trans-elem)

lemma mon-w-empty[simp]:
  \( \text{mon-w } fg \ [] = \{\} \)

by (unfold mon-w-def, auto)

lemma mon-w-single[simp]:
  \( \text{mon-w } fg \ [e] = \text{mon-e } fg \ e \)

by (unfold mon-w-def, auto)

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

by (unfold mon-w-def) auto

lemma mon-w-uncons[simp]:
  \( \text{mon-w } fg (e#w) = \text{mon-e } fg e \cup \text{mon-w } fg w \)

by (rule mon-w-unconc[where wa=[e], simplified])
\textbf{lemma} \textit{mon-w-ileq}: \( w \preceq w' \implies \text{mon-w} \ fg \ w \subseteq \text{mon-w} \ fg \ w' \)
\begin{itemize}
  \item \textit{by} (induct rule: \textit{less-eq-list-induct}) \textit{auto}
\end{itemize}

\section{Valid configurations}

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

\textbf{definition}
\begin{align*}
\text{valid} \ fg \ c \ &= \ \forall s \ s'. \ \{ \#s\# \} + \{ \#s'\# \} \leq c \implies \text{mon-s} \ fg \ s \cap \text{mon-s} \ fg \ s' = \{} \\
\end{align*}

\textbf{lemma} \textit{valid-empty}[\textit{simp, intro}]\!: \text{valid} \ fg \ \{ \# \}
\begin{itemize}
  \item \textit{by} (unfold \textit{valid-def}, \textit{auto})
\end{itemize}

\textbf{lemma} \textit{valid-single}[\textit{simp, intro}]\!: \text{valid} \ fg \ \{ \#s\# \}
\begin{itemize}
  \item \textit{by} (unfold \textit{valid-def mset-le-def}) \textit{auto}
\end{itemize}

\textbf{lemma} \textit{valid-split1}:
\begin{align*}
\text{valid} \ fg \ (c+c') \implies \text{valid} \ fg \ c \land \text{valid} \ fg \ c' \land \text{mon-c} \ fg \ c \cap \text{mon-c} \ fg \ c' = \{} \\
\end{align*}
\begin{itemize}
  \item \textit{apply} (unfold \textit{valid-def})
  \item \textit{apply} (\textit{auto simp add: mset-le-incr-right})
  \item \textit{apply} (\textit{drule mon-cD}+)
  \item \textit{apply} \textit{auto}
  \item \textit{apply} (\textit{subgoal-tac \{\#s\#\} + \{\#sa\#\} \leq c+c'})
  \item \textit{apply} (\textit{auto dest: mset-le-mono-add iff add: mset-le-single-conv[symmetric] simp del: mset-le-single-conv})
  \item \textit{done}
\end{itemize}

\textbf{lemma} \textit{valid-split2}:
\begin{align*}
\text{valid} \ fg \ (c+c') \iff \text{valid} \ fg \ c \land \text{valid} \ fg \ c' \land \text{mon-c} \ fg \ c \cap \text{mon-c} \ fg \ c' = \{} \\
\end{align*}
\begin{itemize}
  \item \textit{apply} (unfold \textit{valid-def})
  \item \textit{apply} (\textit{intro impI allI})
  \item \textit{apply} (\textit{erule mset-2dist2-cases})
  \item \textit{apply simp-all}
  \item \textit{apply} (\textit{blast intro: mon-ci}+)
  \item \textit{done}
\end{itemize}

\textbf{lemma} \textit{valid-unconc}:
\begin{align*}
\text{valid} \ fg \ (c+c') \iff (\text{valid} \ fg \ c \land \text{valid} \ fg \ c' \land \text{mon-c} \ fg \ c \cap \text{mon-c} \ fg \ c' = \{} \\
\end{align*}
\begin{itemize}
  \item \textit{by} (\textit{blast dest: valid-split1 valid-split2})
\end{itemize}

\textbf{lemma} \textit{valid-no-mon}: \text{mon-c} \ fg \ c = \{} \implies \text{valid} \ fg \ c
\begin{itemize}
  \item \textit{proof} (\textit{unfold valid-def, intro allI impI})
  \item \textit{fix} \ s \ s'
  \item \textit{assume} \ A: \text{mon-c} \ fg \ c = \{} \textbf{and} \ B: \{\#s\#\} + \{\#s'\#\} \leq c
  \item \textit{from} \text{mon-c-mono[OF B, of fg]} \ A \textbf{have} \text{mon-s} \ fg \ s = \{} \textit{mon-s} \ fg \ s' = \{} \textit{by} (\textit{auto simp add: mon-c-unconc})
  \item \textit{thus} \text{mon-s} \ fg \ s \cap \text{mon-s} \ fg \ s' = \{} \textit{by blast}
\end{itemize}
\textbf{qed}
7.4 Configurations at control points

— A stack is at $U$ if its top node is from the set $U$

**primrec atU-s :: 'n set ⇒ 'n list ⇒ bool where**

\[ atU-s U [] = False \]

\[ atU-s U (u # r) = (u ∈ U) \]

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

by (induct s) auto

— A configuration is at $U$ if it contains a stack that is at $U$

**definition**

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

**lemma atU-empty[simp]:** \( ¬ atU U \{ # \} \)

by (unfold atU-def, auto)

**lemma atU-single[simp]:** \( atU U \{ # s # \} = atU-s U s \)

by (unfold atU-def, auto)

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

by (auto)

**lemma atU-exchange-stack: atU U (\{ # u # r # \} + c) ⇒ atU U (\{ # u # r' # \} + c) \)

by (simp)

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

**definition**

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

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

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

**Lemma** atUV-union[simp]:
\[
atUV U V (c1 + c2) \iff (atUV U V c1) \lor (atUV U V c2) \lor (atU U c1 \land atUV V c2) \lor (atU V c1 \land atU U c2)
\]
apply (unfold atUV-def atU-def)
apply (auto elim \#:mset-2dist2-cases intro \#:mset-le-incr-right iff add \#:mset-le-mono-add-single)
apply (subst union-commute)
apply (auto iff add \#:mset-le-mono-add-single)
done

**Lemma** atUV-union-cases[case-names left right lr rl], consumes 1: \[
\begin{align*}
\text{atUV U V (c1 + c2); } & \implies P; \\
\text{atUV U V c1; } & \implies P; \\
\text{atUV V c2; } & \implies P; \\
\text{atU U c1; atUV V c2; } & \implies P; \\
\text{atU V c1; atU U c2; } & \implies P
\end{align*}
\]
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-base**: \([ (u,Base a,v) \in \text{edges fg}, \text{valid fg \{\#u\#r\#} + c ] \)
  \(\implies ((\{\#u\#r\#}\} + c, LBa a, \{\#v\#r\#}\} + c) \in \text{refpoint fg} \)
- **refpoint-call**: \([ (u,Call p,v) \in \text{edges fg}, \text{valid fg \{\#u\#r\#} + c ] \)

— A base edge transforms the top node of one stack and leaves the other stacks untouched.
— A call edge transforms the top node of a stack and then pushes the entry node of the called procedure onto that stack. It can only be executed if all monitors the called procedure synchronizes on are available. Reentrant monitors are modeled here by checking availability of monitors just against the other stacks, not against the stack of the thread that executes the call. The other stacks are left untouched.
mon fg p ∩ mon-c fg c = {}  
⇒ ([#u#r#]+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) ]]  
⇒ ([#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) ]]  
⇒ ([#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 :: (‘n,’p,’ba,’m,’more) flowgraph-rec-scheme ⇒  
((‘n list * ‘n conf) * (’p,’ba) label * (‘n list * ‘n conf)) set

for fg

where

trss-base: [[(u, Base a, v) ∈ edges fg]]  
⇒  
((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 = {} ]]  
⇒  
((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 ]]  
⇒  
((u#r,c), LSpawn p, (#v,r,#[#entry fg p#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]  
⇒  
valid fg ([#s’#]+c’)

apply (erule trss.cases)

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

lemma (in flowgraph) trss-valid-preserve:
\[
\begin{align*}
((s,c),w,(s',c')) \in \text{trcl} \Rightarrow \text{valid fg} (\{\#s\#\} + c) & \implies \text{valid fg} (\{\#s'\#\} + c') \\
\text{by (induct rule: trcl-pair-induct) (auto intro: trss-valid-preserve-s)}
\end{align*}
\]

lemma (in flowgraph) tr-valid-preserve-s:
\[
\begin{align*}
(c,e,c') \in \text{tr fg}; \text{valid fg} c & \implies \text{valid fg} c' \\
\text{by (rule gtr-preserve-s[where P=valid fg]) (auto dest: trss-valid-preserve-s)}
\end{align*}
\]

lemma (in flowgraph) trp-valid-preserve:
\[
\begin{align*}
((s,c),w,(s',c')) \in \text{trcl} \Rightarrow \text{valid fg} (\{\#s\#\} + c) & \implies \text{valid fg} (\{\#s'\#\} + c') \\
\text{by (rule gtrp-preserve-s[where P=valid fg]) (auto dest: trss-valid-preserve-s)}
\end{align*}
\]

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 \text{refpoint fg}) \iff ((c,e,c') \in \text{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 \text{trcl} (\text{refpoint fg})) \iff ((c,w,c') \in \text{trcl} (\text{tr fg}))
proof
have \((c,w,c') \in \text{trcl} (\text{refpoint fg}) \implies \text{valid fg} c \implies ((c,w,c') \in \text{trcl} (\text{tr fg}))\) by (induct rule: trcl.induct) (auto simp add: refpoint-eq-s trss-valid-preserve-s)
moreover have \((c,w,c') \in \text{trcl} (\text{tr fg}) \implies \text{valid fg} c \implies ((c,w,c') \in \text{trcl} (\text{refpoint fg})\) 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 \text{trcl} (\text{refpoint fg}) = ((c,w,c') \in \text{trcl} (\text{tr fg}))
qed

7.6.3 Case distinctions

lemma trss-c-cases-s[cases set, case-names no-spawn spawn]:
\[
\begin{align*}
((s,c),e,(s',c')) \in \text{trss fg};
\end{align*}
\]

38
\[
\begin{align*}
\text{lemma trss-c-fmt-s: } & \left[\left(\left(s, c, (s', c')\right) \in \text{trss } fg\right) \wedge
\begin{align*}
& c = c' + c \\
& c' = c + c \\
& \exists p u v. u = \text{Entries } p \\
& d = d' \\
& c = c' + c \\
& c' = c + c \\
& \exists p u v. u = \text{Entries } p \\
& d = d' \\
& \end{align*}
\right] \implies P \tag{1}
\end{align*}
\]

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

\text{lemma trss-c-split-s: } \left[\left(\left(s, c, (s', c')\right) \in \text{trss } fg\right) \wedge
\begin{align*}
& c = c' + c \\
& c' = c + c \\
& \exists p u v. u = \text{Entries } p \\
& d = d' \\
& c = c' + c \\
& c' = c + c \\
& \exists p u v. u = \text{Entries } p \\
& d = d' \\
& \end{align*}
\right] \implies P \tag{2}
\]

\text{by \text{force elim!: trss.c-cases-s})}

\text{lemma \text{in flowgraph) trss-c-split-s: } \left[\left(\left(s, c, (s', c')\right) \in \text{trss } fg\right) \wedge
\begin{align*}
& c = c' + c \\
& c' = c + c \\
& \exists p u v. u = \text{Entries } p \\
& d = d' \\
& c = c' + c \\
& c' = c + c \\
& \exists p u v. u = \text{Entries } p \\
& d = d' \\
& \end{align*}
\right] \implies P \tag{3}
\]

\text{apply \text{erule trss-c-cases-s})}

\text{apply \text{subgoal-tac csp))}

\text{apply \text{fastforce))}

\text{apply \text{auto))}

\text{done)

\text{lemma trss-c-cases[cases set, case-names c-case]: } \exists \ s. \left[\left(\left(s, c, (s', c')\right) \in \text{trcl } (\text{trss } fg)\right) \wedge
\begin{align*}
& c = c' + c \\
& c' = c + c \\
& \exists p u v. u = \text{Entries } p \\
& d = d' \\
& c = c' + c \\
& c' = c + c \\
& \exists p u v. u = \text{Entries } p \\
& d = d' \\
& \end{align*}
\right] \implies P \tag{4}
\]

\text{apply \text{induct \text{w))}

\text{case Nil note \text{A=this))}

\text{hence \text{s'=s c'=c by \text{simp-all}})

\text{hence \text{c'=(#)+c by \text{simp})}

\text{from A(2)\text{OF \text{this) show } P by \text{simp})

next)

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

\text{then obtain sh ch where SPLIT1: } (\left(\left(s, c, (sh, ch)\right) \in \text{trss } fg \text{ and SPLIT2: } (\left(\left(sh, ch\right), (s', c')\right) \in \text{trcl } (\text{trss } fg)\right) \text{ by \text{fast dest: trcl-uncons))}

\text{from SPLIT2 show ?case \text{proof (rule \text{IHP\text{(1))}})

\text{fix \text{csp))}

\text{assume \text{C'FMT: c'=csp+c and CSPFMT: } !.s : # csp \implies \exists p u v. s = \text{Entries } fg p \wedge}

\text{(u,Spawn p, v) \in \text{edges } fg \wedge initialproc } fg p \tag{5}
\]

\text{from SPLIT1 show ?thesis)

\text{proof (rule trss-c-cases-s))

\text{assume \text{ch=c with \text{C'FMT CSPFMT IHP\text{(3) show ?case by blast})

next)

\text{fix \text{p))}

\text{assume \text{EFMT: } e = \text{Entries } p \text{ and CHFMT: ch=} (#\text{Entries } fg p)[#]+c}

\text{with \text{C'FMT have } c'=((#\text{Entries } fg p)[#]+csp)+c \text{ by \text{simp add: union-ac})

\text{moreover)

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

39}
trss_cases)

  hence !s. s :@ {#[entry fg p]#} + csp ⇒ ∃ p u v. s = [entry fg p] ∧ (u, Spawn p, v) ∈ edges fg ∧ initialproc fg p using CSPFMT by (unfold initialproc-def, erule_tac mset-un-cases) (auto)

  ultimately show ?case using IHP(3) by blast

  qed

  qed

lemma (in flowgraph) c-of-initial-no-mon:
  assumes A: !s. s:@csp ⇒ ∃ p u v. 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
  qed

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

  qed

corollary (in flowgraph) trss-c-no-mon:
  ((s,c),w,(s',c'))∈trcl (trss fg) ⇒ mon-c fg c' = mon-c fg c
  apply (auto elim!: trss-cases simp add: mon-c-unconc)
  proof
  fix csp x
  assume x ∈ mon-c fg csp
  then obtain s where s:@csp and M: x ∈ mon-s fg s by (unfold mon-c-def, auto)
  moreover assume ∀ s. 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]:
\[(s,c), LSpawn p, (s',c') \in trss \implies mon\ p = \{\}\]
by (auto elim: trss.cases)

lemma trss-no-empty-s[simp]: (([],c),(s',c')) \in trss \implies False
by (auto elim!: trss.cases)

lemma trss-no-empty[simp]:
assumes A: (([],c),w,(s',c')) \in trcl (trss)
shows w=[] \land s'=[] \land c=c'
proof -
note A
moreover {
  fix s
  have ((s,c),w,(s',c')) \in trcl (trss) \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-SPA威尔 SPAWN]:
assumes A: (c,e,c') \in tr f g
assumes A-NO-SPA威尔: !s ce s'.esp. [\[(s,ce),(s',ce') \in trss f g;\]
\[c={\#s\#}+ce; c'={\#s'\#}+ce\]
]\Rightarrow P
assumes A-SPA威尔: !s ce s' p. [\[(s,ce),LSpawn p,(s',{\#entry f g p\#}+ce) \in trss f g;\]
\[c={\#s\#}+ce;\]
\[c'={\#s'\#}+{\#entry f g p\#}+ce;\]
\[e=LSpawn p\]
]\Rightarrow P
shows P
proof -
from A show ?thesis proof (erule-tac gtr-find-thread)
  fix s ce s' ce'
  assume FMT: c = {\#s\#} + ce c' = {\#s'\#} + ce'
  assume B: ((s, ce), e, s', ce') \in trss \thus ?thesis proof (cases rule: trss-cases-s)
    case no-spawn thus ?thesis using FMT B by (-) (rule A-NO-SPA威尔, auto)
    next
    case (spawn p) thus ?thesis using FMT B by (-) (rule A-SPA威尔, auto simp add: union-assoc)
qed
7.7 Advanced properties

7.7.1 Stack composition / decomposition

lemma \textit{trss-stack-comp-s}:
\[ ((s,c),e,(s',c'))\in \text{trss}\ f_g \implies ((s\cdot r,c),e,(s'\cdot r,c'))\in \text{trss}\ f_g \]
by (auto elim!: \textit{trss.cases intro: \textit{trss.intros}})

lemma \textit{trss-stack-comp}:
\[ ((s,c),w,(s',c'))\in \text{trcl} (\text{trss}\ f_g) \implies ((s\cdot r,c),w,(s'\cdot r,c'))\in \text{trcl} (\text{trss}\ f_g) \]

proof (induct rule: trcl-pair-induct)
\begin{itemize}
  \item case empty thus ?case by auto
\end{itemize}

next
\begin{itemize}
  \item case (cons s c e sh ch w s' c') note \textit{IHP}=this
\end{itemize}

from \textit{trss-stack-comp-s}[OF \textit{IHP}(1)] have \[ ((s \cdot r, c), e, sh @ r, ch) \in \text{trss}\ f_g . \]
also note \textit{IHP}(3)
finally show ?case .

qed

lemma \textit{trss-stack-decomp-s}:
\[ \exists sp'. s'=sp'\cdot r \land ((s,c),e,(sp',c'))\in \text{trss}\ f_g \]
by (cases s, simp) (auto intro: \textit{trss.intros elim!: \textit{trss.cases}})

lemma \textit{trss-find-return}:
\[ ((s\cdot r,c),w,(r,c'))\in \text{trcl} (\text{trss}\ f_g); \]
\[ \exists wa wb ch. w=wa@wb; ((s,c),wa,([],ch))\in \text{trcl} (\text{trss}\ f_g); \]
\[ ((r,ch),wb,(r,c'))\in \text{trcl} (\text{trss}\ f_g) \]
\[ \implies P \]
\[ \]
— If \( s=[] \), the proposition follows trivially
\[ \]
apply (cases s=[]) apply fastforce
\[ \]
proof —
\[ \]
— For \( s\neq[] \), we use induction by \( w \)
\[ \]
have IM: \[ \exists wa wb ch. w=wa@wb \land ((s,c),wa,([],ch))\in \text{trcl} (\text{trss}\ f_g) \land ((r,ch),wb,(r,c'))\in \text{trcl} (\text{trss}\ f_g) \]
\[ \]
proof (induct w)
\[ \]
\begin{itemize}
  \item case Nil thus ?case by (auto)
\end{itemize}

next
\begin{itemize}
  \item case (Cons e w) note \textit{IHP}=this
\end{itemize}

then obtain sh ch where \textit{SPLIT1}: \[ ((s\cdot r,c),e,(sh,ch))\in \text{trss}\ f_g \] and \textit{SPLIT2}:
\[ ((sh,ch),w,(r,c'))\in \text{trcl} (\text{trss}\ f_g) \] by (fast dest: \textit{trcl-unscons})
\[ \]
\{ assume CASE: \( e=LRet \)

with \textit{SPLIT1} obtain \( p \) where \textit{EDGE}: \( s\cdot r=\text{return}\ f_g p \neq sh \) \( c=ch \) by (auto elim!: \textit{trss.cases})
\[ \]
with \textit{IHP}(3) obtain \( ss \) where \textit{SHFMT}: \( s=\text{return}\ f_g p \neq ss \) \( sh=ss\cdot r \) by (cases s, auto)
\[ \]
\{ assume CC: \( ss\neq[] \)

with \textit{SHFMT} have \[ \exists ss. ss\neq[] \land sh=ss\cdot r \] by blast
\[ \]
moreover {
\[ \]
assume CC: \( ss=[] \)
\[ \]
42
with CASE SHFMT EDGE have \(((s, c), e, ([], ch)) \in \text{trcl} (\text{trss f}g)\ e \# w = [e] \otimes w\)
by (auto intro: \text{trss-rel})

moreover from SPLIT2 SHFMT CC have \(((r, ch), w, (r, c')) \in \text{trcl} (\text{trss f}g)\)
by simp

ultimately have \(?case by blast\)

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

} moreover {

assume \(e \neq LRet\)

with SPLIT1 IHP(3) have \((\exists ss. ss \neq [] \land sh = ss \otimes r)\ by (force elim!: \text{trss.cases})\)

simp add: append-eq-Cons

} moreover {

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

then obtain ss where CASE: \(ss \neq []\ sh = ss \otimes r\ by blast\)

with SPLIT2 have \((ss \otimes r, ch), w, r, c' \in \text{trcl} (\text{trss f}g)\ by simp\)

from IHP(1)|OF this CASE(1) obtain wa wb ch' where IHAPP: \(w = wa \otimes wb\)
\(((ss, ch), wa, ([], ch')) \in \text{trcl} (\text{trss f}g)\ ((r, ch'), wb, (r, c') \in \text{trcl} (\text{trss f}g)\ by blast\)

moreover from CASE SPLIT1 have \(((s \otimes r, c), e, ss \otimes r, ch) \in \text{trss f}g\ by simp\)

from trss-stack-decomp-s|OF this IHP(3) have \(((s, c), e, ss, ch) \in \text{trss f}g\ by auto\)

with IHAPP have \(((s, c), e \# wa, ([], ch')) \in \text{trcl} (\text{trss f}g)\ by (rule-tac \text{trcl.cons})\)

moreover from IHAPP have \(e \# wa = (e \# wa) \otimes wb\ by auto\)

ultimately have \(?case by blast\)

} ultimately show \(?case by blast\)

qed

lemma trss-return-cases[cases set]: \(!u r c. U r c. [ (u \# r, c), w, (r', c') \in \text{trcl} (\text{trss f}g)\) \]
\(! s' u'. [ r' = s' \otimes u' \# r; (u, c), w, (s' \otimes u', c') \in \text{trcl} (\text{trss f}g)\] \(\Rightarrow P;\)

\(! wa wb ch. [ w = wa \otimes wb; (w, c), w, ([], ch) \in \text{trcl} (\text{trss f}g); [(r, ch), wb, (r', c') \in \text{trcl} (\text{trss f}g)\] \(\Rightarrow P\)

\[\Rightarrow P\]

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: \(((u \# r, c), e, (sh, ch)) \in \text{trss f}g\) and SPLIT2: \(((sh, ch), w, (r', c')) \in \text{trcl} (\text{trss f}g)\ by (fast dest: \text{trcl-cons})\)

\{ fix ba q

assume CASE: \(e = LBa\ ba \lor e = LS'paw\ q\)

with SPLIT1 obtain v where E: \(sh = v \# r\ (((u, c), e, ([], ch)) \in \text{trss f}g\) by (auto elim!: \text{trss.cases intro: trss.intros})
with SPLIT2 have \(((v\#r,c),w,(r',c'))\)\in trcl (trss fg) by simp
hence \(?case proof (cases rule: IHP(1)\{of w, simplified, cases set\})
  case \(1 s\# u\) note CC=\this
  with E(2) have \(((\ast\#u),e\#w,\langle s'\#\ast, u', c'\rangle)\in trcl (trss fg) by simp
from IHP(3)\{OF CC(1)\ this\} show \(?thesis .
next
  case \(2 w_{a\#} w_{b\#} c\) note CC=\this
  with E(2) have \(((\ast\#u),e\#\ast\#w,\langle [\ast],ct\rangle)\in trcl (trss fg) e\#w = (e\#\ast\#w)\@ w_{b\#} c\) by simp-all
  from IHP(4)\{OF this(2,1) CC(3)\} show \(?thesis .
  qed
\}
  moreover \{
  assume CASE: \(e=LRet\)
  with SPLIT1 have sh=r \(((\ast\#u),e,(\ast\#\ast\#w,[\ast],ch))\in trcl fg by (auto elim!: trss.cases intro: trss.intros)
  with IHP(4)\{OF - this(2)\} SPLIT2 have \(?case by auto
\}
  moreover \{
  fix \(q\)
  assume CASE: \(e=LCall q\)
  with SPLIT1 obtain \(a'\) where SHFMT: \(sh=entry fg q \# a' \# r \(((\ast\#u),e,(\ast\#\ast\#w,\langle [\ast],ch\rangle))\in trcl fg by (auto elim!: trss.cases intro: trss.intros)
  with SPLIT2 have \(((\ast\#\ast\#w)\# a' \# r,\ast\#u'),w,(r',c')\in trcl (trss fg) by auto
  hence \(?case proof (cases rule: IHP(1)\{of w, simplified, cases set\})
  case \(1 st\# u\) note CC=\this
  from trss-stack-comp\{OF CC(2), where \(r=[\ast]\) have \(((\ast\#\ast\#w)\# [\ast],w,(st \@ [\ast]) \@ [\ast],c')\in trcl (trss fg) by auto
  with SHFMT(2) have \(((\ast\#u),e\#\ast\#w,(st \@ [\ast]) \@ [\ast],c')\in trcl (trss fg) by auto
  from IHP(3)\{OF - this\} CC(1) show \(?thesis by simp
  next
  case \(2 w_{a\#} w_{b\#} ct\) note CC=\this
  from trss-stack-comp\{OF CC(2), where \(r=[\ast]\) have \(((\ast\#\ast\#w)\# [\ast],w,(st \@ [\ast]) \@ [\ast],c')\in trcl (trss fg) by simp
  with SHFMT have PREPATH: \(((\ast\#u),e\#\ast\#w, [\ast],ct)\in trcl (trss fg) by simp
  from CC have L: \(length w_{b\#} \leq length w\) by simp
  from CC(3) show \(?case proof (cases rule: IHP(1)\{OF L, cases set\})
  case \(1 s'' u'\) note CCC=\this from trcl-concat\{OF PREPATH CCC(2)\}
  CC(1) have \(((\ast\#u),e\#\ast\#w,\langle s''\@[\ast],c'\rangle)\in trcl (trss fg) by (simp)
  from IHP(3)\{OF CCC(1)\ this\} show \(?thesis .
  next
  case \(2 w_{ba\#} w_{ba\#} c'\) note CCC=\this from trcl-concat\{OF PREPATH CCC(2)\} CC(1) CCC(1) have \(e\#w = (e\#\ast\#w)\@ w_{ba\#} [\ast],c')\in trcl (trss fg) by auto
  from IHP(4)\{OF this CCC(3)\} show \(?thesis .
  qed
  qed
  } ultimately show \(?case by (cases e, auto)
  qed
\textbf{lemma (in flowgraph)} \texttt{trss-find-call}:

\begin{verbatim}
!!! [v r c'] \rightarrow (\forall r (\forall v (\forall w ((v \neq r') \rightarrow \in trcl\ (trss\ fg) : r' \neq \emptyset))) \rightarrow \exists rh\ ch p\ wa wb. \\
\quad w = wa @ (LCall\ p) \#\ wb \land \\
\quad \text{proc-of}\ fg\ v = p \land \\
\quad ((\forall sp, c, wa, (rh, ch) \in trcl\ (trss\ fg)) \land \\
\quad \forall (\forall rh, ch, LCall\ p, ((\forall entry\ fg\ p) \#\ r', ch) \in trss\ fg \land \\
\quad ((\forall entry\ fg\ p), ch, wb, (\forall [v, c']) \in trcl\ (trss\ fg))
\end{verbatim}

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

\begin{description}
\item \textbf{case Nil thus \&case by (auto)}
\item \textbf{next case (snoc w e) note IHP\textbf{=}this}
\item \textbf{then obtain rh ch where SPLIT1: (([sp], c, w, (rh, ch)) \in trcl (trss fg) and SPLIT2: ((r, ch), (v \neq r', c)) \in trss fg by (fast dest: trcl-rev-uncons)}
\end{description}

\begin{verbatim}
{ \\
\quad \text{assume } \exists u, rh = u \# r' \\
\quad \text{then obtain } u \text{ where RHFM(T[simp]: rh \# u r'} \text{ by blast} \\
\quad \text{with SPLIT2 have } \text{proc-of}\ fg\ u = \text{proc-of}\ fg\ v \text{ by (auto elim: trss.cases intro: edges-part)} \\
\quad \text{moreover from IHP(1)} [(\forall w\ u\ r'\ ch, OF - SPLIT1[simplified] IHP(\&))] \text{ obtain rt ct p wa wb where} \\
\quad \text{IHAPP: } w = wa @ LCall\ p \#\ wb \text{ proc-of}\ fg\ u = p \ (\forall sp, c, wa, (rt, ct) \in \text{trcl (trss fg)} ((rt, ct), LCall\ p, entry\ fg\ p \# r', ct) \in \text{trss}\ fg \\
\quad \forall ((\forall entry\ fg\ p), ct, wb@\ [e], (\forall [v, c']) \in \text{trcl (trss fg))}} \\
\quad \text{proof -} \\
\quad \text{note IHAPP(\&)} \\
\quad \text{also from SPLIT2 have } (([u], ch), (\forall [v, c']) \in \text{trss}\ fg \text{ by (auto elim!: trss.cases intro!: trssintros)} \\
\quad \text{finally show } \forall\text{thesis}.
\item \textbf{qed}
\item \textbf{moreover from IHAPP have } w@\ [e] = wa @ LCall\ p \# (wb@\ [e]) \text{ by auto} \\
\item \textbf{ultimately have } \forall\text{case by auto}
\} \\
\item \textbf{moreover have } (\forall u, rh = u \# r') \lor \forall\text{case}
\item \textbf{proof (rule trss.cases[OF SPLIT2], simp-all) \textbf{—} Cases for base- and spawn edge are discharged automatically}
\item \textbf{— Case: call-edge}
\item \textbf{case (goal1 ca p r u vv) with SPLIT1 SPLIT2 show } \forall\text{case by fastforce}
\item \textbf{next}
\item \textbf{— Case: return edge}
\item \textbf{case (goal2 q r ca) note CC\textbf{=}this}
\item \textbf{hence } [\text{simp: rh = (return\ fg\ q) \# v \neq r'} \text{ by simp} \\
\item \textbf{with IHP(1)} [(\forall w\ (\forall return\ fg\ q) \# v\ r'\ ch, OF - SPLIT1[simplified]) \text{ obtain rt ct wa wb where} \\
\item \textbf{IHAPP: } w = wa @ LCall\ q \# wb \ (\forall sp, c, wa, (rt, ct) \in \text{trcl (trss fg)} ((rt, ct), LCall\ q, entry\ fg\ q \# v \# r', ct) \in \text{trss}\ fg}
\end{verbatim}
proof -

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

from IHAPP(1) have LENWA: length wa ≤ length w by auto

from IHAPP(1)[OF LENWA IHAPP(2)[simplified] IHP(3)] obtain rh ch h p waa wab where

IHAPP': waa=waa@LCall p # wab proc-of fg u = p (((sp),c),waa,(rh,chw))∈trcl (trss fg) (rh,chw),LCall p, (entry fg p#r',chw))∈trss fg
((entry fg p],chw),wab,([u],ct))∈trcl (trss fg)

by blast

from IHAPP IHAPP' PROC-OF-U have w@e=waa@LCall p#(wab@LCall q#ub@e]) ∧ proc-of fg v = p by auto

moreover have ((entry fg p],chw),wab@((LCall q)#ub@e],([v],c'))∈trcl (trss fg)

proof -

note IHAPP'(5)

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

also from trss-stack-comp[OF IHAPP(4)] have ((entry fg q#v],ct),wab, (return fg q#v],ct))∈trcl (trss fg) by simp

also from CC have ((return fg q#v],ct),wab, ([v],c')∈trss fg by (auto intro: trss-ret)

finally show ?thesis by simp

qed

moreover note IHAPP' CC

ultimately show ?case by auto

qed

ultimately show ?case by blast

qed

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

lemma (in flowgraph) trss-find-call':
assumes A: ((sp),c),w,(return fg p#u',c')) ∈ trcl (trss fg)

and EX: !uh ch wa wb. [ w=wa@LCall p)#ub;
((sp),c),wa,([uh],ch))∈trcl (trss fg);
((uh],ch),LCall p,((entry fg p)#[u'],ch))∈trss fg;
(uh, Call p,u')∈edges fg;
((entry fg p],ch),wab,(return fg p],c')∈trcl (trss fg)
]

⇒ P

shows P

proof -

from trss-find-call[OF A] obtain rh ch wa wb where FC:

w = wa @ LCall p # wb
((sp),c), wa, rh, ch) ∈ trcl (trss fg)
((rh, ch), LCall p, [entry fg p, u'], ch) ∈ trss fg
((entry fg p], ch), wb, [return fg p], c') ∈ trcl (trss fg)

by auto

moreover from FC(3) obtain uh where ADD: rh=[uh] (uh, Call p,u')∈edges

46
fg by (auto elim: trss.cases)
ultimately show ?thesis using EX by auto
qed

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

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

lemma (in flowgraph) trss-bot-path-proc-const:
\(\exists w'\ . \ w=w'@[\text{LRet } sh]\land ((s,c),w',([\text{return } fg\ q],c'))\in trcl (trss fg) \implies p=q\)
using trss-bot-proc-const[of [] entry fg p - - [] return fg q, simplified] .

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

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

7.7.2 Adding threads

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

47
\textbf{7.7.3 Conversion between environment and monitor restrictions}

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

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

\textbf{lemma} (in \texttt{flowgraph}) \texttt{trss-modify-context-s}:  
\[ \text{by (erule \texttt{trss.cases}) (auto intro!: \texttt{trssintros})} \]

\textbf{lemma} (in \texttt{flowgraph}) \texttt{trss-modify-context}[\texttt{rule-format}]:  
\[ \forall \text{cn. }((s,c),w,(s',c')) \in \text{trcl } (\text{trss } fg) \]  
\[ \implies \forall \text{cn. } \text{mon-w } fg \ w \cap \text{mon-c } fg \ cn \ = \{ \} \]
\textbf{proof} (induct rule: \texttt{trcl-pair-induct})  
\textbf{case empty} thus \texttt{?case by simp}

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

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

\textbf{lemma} \texttt{trss-add-context}:  
\[ ((s,c),w,(s',c')) \in \text{trcl } (\text{trss } fg) ; \text{mon-w } fg \ w \cap \text{mon-c } fg \ ce = \{ \} \]
\textbf{by} (auto elim!: \texttt{trss.cases intro!: \texttt{trss.intros simp add: union-assoc mon-c-unconc})

\textbf{48}
proof (induct rule: trcl-pair-induct)
  case empty thus ?case by simp
next
  case (cons s c e sh ch w s’ c’)
  note IHP="this"
  from IHP(4) have MM: mon-e fg e ∩ mon-c fg ce = {} mon-w fg w ∩ mon-c fg ce = {}
  qed

lemma trss-drop-context-s: \[
\begin{array}{l}
\forall s c. ((s,c+ce),(s’,c’+ce))\in trss fg \\
\Rightarrow ((s,c),(s’,c’))\in trss fg \land mon-e fg e \land mon-c fg ce = {} \\
\end{array}
\]
  by (erule trss_cases) (auto intro!: trss.intros simp add: mon-c-unconc union_assoc[of - c ce, symmetric])

lemma trss-drop-context: \[
\begin{array}{l}
\forall s c. ((s,c+ce),w,(s’,c’+ce))\in trcl (trss fg) \\
\Rightarrow ((s,c),w,(s’,c’))\in trcl (trss fg) \land mon-w fg w \land mon-c fg ce = {} \\
\end{array}
\]
proof (induct w)
  case Nil thus ?case by auto
next
  case (Cons e w) note IHP="this"
  then obtain sh ch where SPLIT: ((s,c+ce),e,(sh,ch))\in trss fg ((sh,ch),w,(s’,c’+ce))\in trcl (trss fg) by (fast dest: trcl-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 trss fg mon-e fg e \land mon-c fg ce = {} by blast+
  moreover from CHFMT IHP(1) SPLIT(2) have ((sh,csp+c),w,(s’,c’))\in trcl (trss fg) mon-w fg w \land mon-c fg ce = {} by blast+
  ultimately show ?thesis by auto
qed

lemma trss-xchange-context-s: \[
\begin{array}{l}
\text{assumes } A: ((s,c),e,(s’,csp+c))\in trss fg \\
\text{and } M: mon-c fg cn \subseteq mon-c fg e \\
\text{shows } ((s,cn),e,(s’,csp+cn))\in trss fg \\
\end{array}
\]
proof –
  from trss-drop-context[of - \{#\}, simplified, OF A] have DC: ((s, \{#\}), e, s’, csp) \in trss fg mon-c fg e \land mon-c fg ce = {} by simp-all
  with M have mon-e fg e \land mon-c fg cn = {} by auto
  from trss-add-context-s[of DC(1) this] show ?thesis by auto
qed

lemma trss-xchange-context: \[
\begin{array}{l}
\text{assumes } A: ((s,c),w,(s’,csp+c))\in trcl (trss fg) \\
\text{and } M: mon-c fg cn \subseteq mon-c fg e \\
\text{shows } ((s,cn),w,(s’,csp+cn))\in trcl (trss fg) \\
\end{array}
\]
proof –
  from trss-drop-context[of - \{#\}, simplified, OF A] have DC: ((s, \{#\}), w, s’,
\begin{verbatim}
csp) ∈ trcl (trss fg) mon-w fg w ∩ mon-c fg c = {0} by simp-all with M have mon-w fg w ∩ mon-c fg cn = {0} by auto from trss-add-context[OF DC(1) this] show ?thesis by auto qed

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

lemma trss-drop-all-context[cases set, case-names dropped]:
  assumes A: ((s,c),w,(s',c'))∈trcl (trss fg)
  and C: ‹!csp. [ c'=_csp+c; ((s,#),w,(s',csp))∈trcl (trss fg)] \implies P›
  shows P using A proof (cases rule: trss-c-cases)
  case (c-case csp) with trss-xchange-context[of s c e 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 = {0} ] \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 = {0} ] \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

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

8.1 Semantic properties of restricted flowgraphs

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

— Procedures with isolated return nodes will never return

lemma (in eflowgraph) iso-ret-no-ret: !!u c. 

\[
\text{isolated-ret \ fg \ p;} \\
\text{proc-of \ fg \ u = p;} \\
\text{u \neq \ return \ fg \ p;} \\
\text{(([u],c),w,([return \ fg \ p'],c'))} \in \text{trcl (trss \ fg)}
\]

\[\Rightarrow \text{False}\]

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

case Nil thus ?case by auto

next

case (Cons e w) note IHP=this

then obtain sh ch where SPLIT1: 

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

show ?case proof (cases e)

| case LRet with 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\neq\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\neq\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 wh where A: sh=entry \fg \ q\#[uh] proc-of \fg \ uh = \ p \ uh\neq\return \fg \ p \ by (force elim!: trss.cases simp add: edges-part isolated-ret-def) |
| with SPLIT2 have B: 

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

\by simp

| from trss-return-cases[OF B] obtain w1 w2 ct where C: w=w1@w2 length w2 \leq \text{length } w \ (([entry \fg q],ch),w1,([],[ct])) \in \text{trcl (trss \ fg)} \ (([uh],[ct]),w2,([return \fg p'],c')) \in \text{trcl (trss \ fg)} \text{ by (auto)
from IHP\(1)\) OF C\(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:

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

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

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

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

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

shows False

**proof**

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

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

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

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

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

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

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

qed

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

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

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

shows False

**proof** (cases \(r'\))

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

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

show ?thesis proof (cases \(w'\))

case Nil with \(B\) have \(p=q\) by (auto dest: trcl-empty-cons)

next

case Cons hence \(w' \neq []\) by simp
from no-sl-from-initial[OF this A(2) B(2)] show False .

qed

next

case (Cons u rr) with A(4) have r'=[u] by auto

with no-sl-from-initial[OF A(1,2)] A(3) show False by auto

qed

8.2 Definition of normalized paths

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

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

As for the original semantics, we first define the normalized semantics on a single thread with a context and then use the theory developed in Section 5 to derive interleaving semantics on multisets and configurations with an explicit local thread (loc/env-semantics, cf. Section 5.4).

```plaintext
inductive-set
  ntrs :: (n,p,ba,m,more) flowgraph-rec-scheme ⇒
        ((n list × n conf) × (p,ba) label list × (n list × n conf)) set
for fg
where
— A macrostep transforms one thread by first calling a procedure and then doing a same-level path
  ntrs-step: \(((u#r,ce),LCall p, (entry fg p # u' # r,ce))∈trss fg;
        \(((\text{entry fg p})|ce),w,(\text{|v,ce'})\rangle∈trcl (trss fg)\] ] =⇒
        (((u#r,ce),LCall p#w,(v#u'#r,ce'))∈ntrs fg

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

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

\(^2\)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.
apply auto
done

8.3 Representation property for reachable configurations

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

The first direction is to show that a normalized path is also a path. This
follows from the definitions. Note that we first show that a single macrostep

corresponds to a path and then generalize the result to sequences of macrosteps

```
lemma ntrs-is-trss-s: ((s,c),w,(s',c'))∈ntrs fg ⇒ ((s,c),w,(s',c'))∈trcl (trss fg)
proof (erule ntrs_cases, auto)
```

```
fix p r u u' v w
assume A: ((u # r, c), LCall p, entry fg p # u' # r, c) ∈ trss fg ((entry fg p, c), w, [v], [c']) ∈ trcl (trss fg)
from trss-stack-comp[OF A(2), of u'#r] have ((entry fg p # u' # r, c), w, v # u'#r, c') ∈ trcl (trss fg) by simp
with A(1) show ((u # r, c), LCall p # w, v # u'#r, c') ∈ trcl (trss fg) by auto
```

```
qed
```

```
lemma ntrs-is-trss: ((s,c),w,(s',c'))∈trcl (ntrs fg)
⇒ ((s,c),foldl (op @) [] w,(s',c'))∈trcl (trss fg)
proof (induct rule: trcl_pair_induct)
  case empty thus ?case by simp
next
  case (cons s e c e' ch w s' c') note IHP=this
  from trcl-concat[OF ntrs-is-trss-s[OF IHP(1)] IHP(3)] foldl-conc-empty-eq[of e w] show ?case by simp
```

```
qed
```

```
lemma ntr-is-tr-s: (c,w,c')∈ntr fg ⇒ (c,w,c')∈trcl (tr fg)
by (erule gtrE) (auto dest: ntrs-is-trss intro: gtrI)
```

```
lemma ntr-is-tr: (c,w,c')∈trcl (ntr fg) ⇒ (c,foldl (op @) [] w,c')∈trcl (tr fg)
proof (induct rule: trcl_induct)
  case empty thus ?case by auto
next
  case (cons ee c' w w' c'' ) note IHP=this
  from trcl-concat[OF ntr-is-tr-s[OF IHP(1)] IHP(3)] foldl-conc-empty-eq[of ee w w'] show ?case by (auto)
```

```
qed
```

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

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

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

**Lemma (in efowgraph) ntr-sl-move-left:** !!ce u r w r’ ce’.

\[
\begin{align*}
& ((\#_{\text{entry f g p}}\#), \text{ww}, \#_{u \# r \#} + ce) \in \text{trcl (ntr f g)}; \\
& (([u, ce), w, (r’ ce’)) \in \text{trcl (trss f g)}; \\
& \text{initialproc f g p;}
\end{align*}
\]

\[\text{length } r’ \leq 1; w \neq [] \]

\[\implies \exists \text{ww’}. ((\#_{\text{entry f g p}}\#), \text{ww’}, \#_{r’ @ r \#} + ce) \in \text{trcl (ntr f g)}\]

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

**Case Nil note CC=this hence u=entry f g p by auto**

— If the normalized path is empty, we get a contradiction, because there is no same-level path from the initial configuration of a thread

**With CC(2) no-retsl-from-initial(OF CC(5,3) - CC(4)) have False by blast thus ?case ..**

**Next**

**Case (snoc ee ww) note IHP=this**

— In the induction step, we extract the last macrostep

**Then obtain ch where SPLIT:** (((\#_{\text{entry f g p}}\#), w, w, \text{ch}) \in \text{trcl (ntr f g)} (ch, ce, \# u\#r \#) + ce) \in ntr f g by (fast dest: trcl-rev-uncons)

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

**From SPLIT(2) obtain q wus uh rh ceh uh’ vt cet where**

**STEPFMT:** ee = LCall q\#wus ch = (\# uh\#rh \#) + ceh \# u\#r \# + ce = \# vt\#uh’\#rh \# + cet ((u\#rh, ceh), LCall q, (entry f g q\#uh’\#rh, ceh)) \in trss f g ((\text{entry f g q}, ceh), wus, (vt, cet)) \in trcl (trss f g)

by (blast elim!: gtrE ntrs.cases[simplified])

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

**From STEPFMT(3) show ?case proof (cases rule: mset-single-cases’)**

— If the sl/ret path was executed on the same thread as the last macrostep
case \texttt{loc note} \texttt{CASE=this} \texttt{hence} \texttt{C'}: \texttt{u=vt \ r=uh'\#rh \ ce=cet \ by} \texttt{auto}
— we append it to the last macrostep.
\begin{verbatim}
with \texttt{STEPFMT(5)} \texttt{IHP(3)} \texttt{have NEWPATH}: (((entry fg q), ceh), \texttt{wws@w}, (r', ce')) \in \texttt{trcl (trss fg)} \texttt{by} \texttt{(simp add: trcl-concat)}
— We then distinguish whether we appended a same-level or a returning path
\texttt{show \ ?thesis \ proof (cases r')}
— If we appended a same-level path
\begin{verbatim}
case \texttt{(Cons \ v')} — Same-level path \texttt{with IHP(5)} \texttt{have CC: r'=[v'] \ by} \texttt{auto}
— The macrostep still ends with a same-level path
\end{verbatim}
\begin{verbatim}
with \texttt{NEWPATH} \texttt{have} (((entry fg q), ceh), \texttt{wws@w}, ([v'], ce')) \in \texttt{trcl (trss fg)} \texttt{by}
\texttt{simp}
— and thus remains a valid macrostep
\end{verbatim}
\begin{verbatim}
from \texttt{gtrI-s \ OF ntrs-step[OF STEPFMT(4), simplified, \ OF this]} \texttt{have} \texttt{([#uh \ \# rh#] + ceh, LCall q \ \# wws@w, \ [#v' \ \# uh' \ \# rh#] + ce')} \texttt{in ntr fg} .
— that we can append to the prefix of the normalized path to get our proposition
\end{verbatim}
\begin{verbatim}
with \texttt{STEPFMT(2)} \texttt{SPLIT(1) CC C'(2)} \texttt{have} (((entry fg q), wws@w), LCall q \ # wws@w, \ [# r@r #] + ce') \in \texttt{trcl (ntr fg)} \texttt{by}
\texttt{(auto simp add: trcl-rev-cons)}
\texttt{thus \ ?thesis \ by} \texttt{blast}
\end{verbatim}
\begin{verbatim}
next
— If we appended a returning path
\texttt{case \ Nil note CC=\texttt{this}}
— The macrostep now ends with a returning path, and thus gets a same-level path
\begin{verbatim}
\texttt{have NEWSL:} (((uh], ceh), \texttt{LCall q \ # wws @ w, [uh'], ceh')} \in \texttt{trcl (trss fg)}
\texttt{proof —}
\texttt{from \texttt{STEPFMT(4)} \texttt{have} (((uh], ceh), \texttt{LCall q, (entry fg q#[uh'], ceh})) \in \texttt{trss fg} \texttt{by}
\texttt{(auto elim!: trss.cases intro: trss.intros)}
\texttt{also from \texttt{trss-stack-comp[OF NEWPATH] CC have} (((entry fg q#[uh'], ceh), wws@w, ([uh'], ce')) \in \texttt{trcl (trss fg)} \texttt{by} \texttt{auto}
\texttt{finally show \ ?thesis \ .}}
\texttt{qed}
— Hence we can apply the induction hypothesis and get the proposition
\end{verbatim}
\begin{verbatim}
from \texttt{IHP(1)[OF - NEWSL] SPLIT STEPFMT(2) IHP(4) CC C'(2)} \texttt{show}
\texttt{?thesis \ by} \texttt{auto}
\texttt{qed}
\texttt{next}
— If the sl/ret path was executed on a different thread than the last macrostep
\texttt{case \ (env cc) note \texttt{CASE=\texttt{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
\begin{verbatim}
from \texttt{STEPFMT(5) obtain cspt where CETFMT: cet=cspt+ceh \ !s. s:#cspt \iff \exists p. s=[entry fg p] \land \texttt{initialproc fg p}}
\texttt{by} \texttt{(unfold \ initialproc-def) \ (erule \ trss-cases, blast)}
— The spawned threads do not hold any monitors yet
\texttt{hence \texttt{CSPT-NO-MON: \mon-c fg cspt = \{\} \ by} \texttt{(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
\end{verbatim}
\end{verbatim}
from CASE(1) CETFMT(1) have u#r :# cspt+ceh by auto
thus ?thesis proof (cases rule: mset-un-cases[cases set])
— The sl/ret path cannot have been executed on a freshly spawned thread
due to the restrictions we made on the flowgraph

case left — Thread was spawned with CETFMT obtain q where u=entry
fg q r=[] initialproc fg q by auto
with IHP(3,5,6) no-rets-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 CEHFM T: ceh={# u#r #}+(ceh-{# u#r #}) by auto
have CHFM T: ch ={# u#r #}+({# uh#rh #}+(ceh-{# u#r #}))
proof —
from CEHFM T STEPFM T(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
c proof —
have mon-n fg uh ⊆ mon-n fg uh' using STEPFM T(4) by (auto elim!: trss.cases dest: mon-n-same-proc edges-part)
moreover have mon-c fg (ceh-{# u#r #}) ⊆ mon-c fg cc proof —
from CASE(3) CETFMT have cc=(cspt+ceh)−{#u#r#} by simp
with CC have cc = cspt+(ceh−{#u#r#}) by (auto 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 cs'p where CE'FM T: ce'=csp'+cc mon-c fg cs'p = {}
by (−) (erule 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) CETFMT have NSL:
(((u), {#uh # rh#} + (ceh-{# u# rh#})), w, r', cs'p + ({#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) CHFM T obtain ww' where
NNPATH: ((#entry fg p[#]), ww', {#r' @ r#} + (cs'p + ((#uh # rh#) + (ceh
− {#u # r#}))) ∈ trcl (nttr 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#} + (cs'p + ({#uh # rh#} + (ceh − {#u # r#}))), ee,
\{#vt \not= uh' \not= rh\} + (csp + (#r'@r\} + (csp' + (ceh - (#u \not= r\}))))
\in \text{ntr fg }

\text{proof —}

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

\text{have MON-CEH: mon-c fg \{#r'@r\} + (csp' + (ceh - (#u \not= r\)}) \subseteq mon-c fg \text{ ceh proof —}}

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

moreover from CEHFMT have mon-c fg ceh = mon-c fg \{#u \not= r\} + (ceh - (#u \not= r\}) by simp — Need to state this explicitly because of recursive simp rule ceh = \{#u \not= r\} + (ceh - (#u \not= r\})

ultimately show \text{thesis using CE'FMT(2) by (auto simp add: mon-c-unconc mon-s-unconc)}

qed

— And we can reassemble the macrostep within the new context

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

moreover from trss-xchange-context[\text{OF - MON-CEH, of \text{entry fg q}} u ws [vt] csp]\text{ STEPFMT(5) CETFMT(1) have

((\text{entry fg q}, \{#r'@r\} + (csp' + (ceh - (#u \not= r\}))), uws, [vt],
csp + (\{#r'@r\} + (csp' + (ceh - (#u \not= r\})))) \in \text{trcl (trss fg) by blast

moreover note STEPFMT(1)

ultimately have ((uh#rh,(\{#r'@r\} + (csp' + (ceh - (#u \not= r\}))),\text{vt,ah',rh,csp}+(\{#r'@r\} + (csp' + (ceh - (#u \not= r\}))))\in\text{ntrs fg by (blast intro: ntrs.intros[simplified])

from \text{gtrI-s[\text{OF this} show \text{thesis by (simp add: union-ac)}

qed

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

from \text{trcl-rev-cons[\text{OF NNPATH this} have (\{#|entry fg p|\}, wu'@\text{[ce],
\{#vt \not= uh' \not= rh\} + (csp + (\{#r'@r\} + (csp' + (ceh - (#u \not= r\}))))
\in \text{trcl (ntr fg) .

— And show that we got the right configuration

moreover from CC CETFMT CASE(3)[symmetric] CASE(2) CE'FMT(1) have \{#vt \not= uh' \not= rh\} + (csp + (\{#r'@r\} + (csp' + (ceh - (#u \not= r\})))) = \{# r'@r\} + ce' by (simp add: union-ac diff-union-single-convs

ultimately show \text{thesis by auto

qed

qed

qed

Finally we can prove: \text{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') ∈ trcl (tr fg);
  cstart = [# [entry fg p] #];
  initialproc fg p []
  ⇒ ∃ w'. ( [# [entry fg p] #], w', c') ∈ trcl (ntr fg)
  — The lemma is shown by induction on the reaching path

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

next 

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

  then obtain w' where IHP': ([# [entry fg p] #], w', c') ∈ trcl (ntr fg) (c, e, c') ∈ tr
  fg by blast
  — We make explicit the thread on that this last step was executed

  from gtr-find-thread [OF IHP'(2)] obtain s ce s' ce' where TSTEP: c = {#s#}
  + ce c' = {#s'##} + ce' ((s, ce), e, (s', ce')) ∈ trss fg by blast
  — The proof is done by a case distinction whether the last step was a call or not

  { 
    — Last step was a procedure call
    fix q
    assume CASE: e=LCall q
    — As it is the last step, the procedure call will not return and thus is a valid macrostep
    have (c, LCall q # [], e') ∈ ntr fg using TSTEP CASE by (auto elim!: trss.cases intro!: ntrs.intros gtr-s trss.intros)
    — That can be appended to the initial normalized path
    from trcl-rev-cons [OF IHP'(1) this] have case by blast
  }
  moreover { 
    — Last step was no procedure call
    fix q a
    assume CASE: e=LBase a ∨ e=LSpawn q ∨ e=LRet
    — Then it is a same-level or returning path
    with TSTEP(3) obtain a r r' where SLR: s = u#r s' = r@r length r' ≤ 1
    (((u), ce), (r', ce')) ∈ trcl (trss fg) by (force elim!: trss.cases intro!: trss.intros)
    — That can be appended to the normalized path using the
    { [# [entry fg p] #], ?ww, { # ?u # ?r# } + ?ce} ∈ trcl (ntr fg): 
    (((?u), ?ce), ?w, ?r', ?ce') ∈ trcl (trss fg); initialproc fg ?p; length ?r' ≤ 1; ?w ≠ []} ⇒ ∃ ww'. 
    { [# [entry fg p] #], ww', { # ?r' @ ?r# } + ?ce'} ∈ trcl (ntr fg) - lemma
    from ntr-sl-move-left [OF - SLR(4) IHP(5) SLR(3)] IHP'(1) TSTEP(1)
    SLR(1) obtain ww' where ( [# [entry fg p] #], ww', { # ?r' @ ?r# } + ce') ∈ trcl (ntr fg) by auto
    with SLR(2) TSTEP(2) have case by auto
  }
  ultimately show case by (cases e, auto)

qed

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

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

\[
\begin{align*}
\vdash \Rightarrow P \\
\Rightarrow \Rightarrow P \\
\text{by (auto dest!: ntrs-is-trss-s elim!: trss-c-cases)}
\end{align*}
\]

**Lemma** `ntrs-c-cases[cases set]`: [ ]

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

\[
\begin{align*}
\vdash \Rightarrow P \\
\Rightarrow \Rightarrow P \\
\text{by (auto dest!: ntrs-is-trss elim!: trss-c-cases)}
\end{align*}
\]

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`:

\[
\begin{align*}
\text{assumes A: } & \quad ((s,c),e,(s',c'))\in \text{ntrp fg} \\
& \quad \text{and V: } \text{valid fg } (\{\#s\#\}+c) \\
\text{shows } & \quad \text{valid fg } (\{\#s'\#\}+c) \\
\text{using } & \quad \text{ntr-valid-preserve-s[OF gtrp2gtr-s[OF A] V]} \text{ by assumption}
\end{align*}
\]

**Lemma (in flowgraph)** `ntrp-valid-preserve`:

\[
\begin{align*}
\text{assumes A: } & \quad ((s,c),e,(s',c'))\in \text{trcl (ntrp fg)} \\
& \quad \text{and V: } \text{valid fg } (\{\#s\#\}+c) \\
\text{shows } & \quad \text{valid fg } (\{\#s'\#\}+c) \\
\text{using } & \quad \text{ntr-valid-preserve[OF gtrp2gtr[OF A] V]} \text{ by assumption}
\end{align*}
\]
8.4.2 Monitors

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

**definition**

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

**definition**

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

**definition**

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

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

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

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

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

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

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

\[ \text{mon-ww } fg \text{ } (\text{w1 }\&\text{w2}) = \text{mon-ww } fg \text{ w1 } \lor \text{mon-ww } fg \text{ w} \]

by (induct \text{w1}) auto

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

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

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

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

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

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

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

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

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

\[ \text{mon-env } fg \text{ } (\text{w1 }\&\text{w2}) = \text{mon-env } fg \text{ w1 } \lor \text{mon-env } fg \text{ w2} \]

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

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

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

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

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

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

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

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

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

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

\[ \text{mon-loc } fg \text{ } (\text{w1 }\&\text{w2}) = \text{mon-loc } fg \text{ w1 } \lor \text{mon-loc } fg \text{ w2} \]

by (unfold \text{mon-loc-def}) (auto simp add: mon-ww-unconc)
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: \( w \preceq w' \implies \operatorname{mon-ww} f g w \subseteq \operatorname{mon-ww} f g w' \)
by (induct rule: less-eq-list-induct) auto

lemma mon-ww-cil:
  \( w \in w_1 \odot_{\alpha} w_2 \implies \operatorname{mon-ww} f g w = \operatorname{mon-ww} f g w_1 \cup \operatorname{mon-ww} f g w_2 \)
by (induct rule: cil-set-induct-fix \alpha) auto

lemma mon-loc-cil:
  \( w \in w_1 \odot_{\alpha} w_2 \implies \operatorname{mon-loc} f g w = \operatorname{mon-loc} f g w_1 \cup \operatorname{mon-loc} f g w_2 \)
by (induct rule: cil-set-induct-fix \alpha) auto

lemma mon-env-cil:
  \( w \in w_1 \odot_{\alpha} w_2 \implies \operatorname{mon-env} f g w = \operatorname{mon-env} f g w_1 \cup \operatorname{mon-env} f g w_2 \)
by (induct rule: cil-set-induct-fix \alpha) auto

lemma mon-ww-of-le-rem:
  \( \operatorname{mon-ww} f g (\text{map le-rem-s} w) = \operatorname{mon-loc} f g w \cup \operatorname{mon-env} f g w \)
by (induct w) (auto split: el-step.split)

lemma mon-env-ileq: \( w \preceq w' \implies \operatorname{mon-env} f g w \subseteq \operatorname{mon-env} f g w' \)
by (induct rule: less-eq-list-induct) auto

lemma mon-loc-ileq: \( w \preceq w' \implies \operatorname{mon-loc} f g w \subseteq \operatorname{mon-loc} f g w' \)
by (induct rule: less-eq-list-induct) auto

lemma mon-loc-map-loc [simp]: \( \operatorname{mon-loc} f g (\text{map LOC} w) = \operatorname{mon-ww} f g w \)
by (unfold mon-loc-def) simp

lemma mon-env-map-env [simp]: \( \operatorname{mon-env} f g (\text{map ENV} w) = \operatorname{mon-ww} f g w \)
by (unfold mon-env-def) simp

lemma mon-loc-map-env [simp]: \( \operatorname{mon-loc} f g (\text{map ENV} w) = \{} \)
by (induct w) auto

lemma mon-env-map-loc [simp]: \( \operatorname{mon-env} f g (\text{map LOC} w) = \{} \)
by (induct w) auto

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

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

lemma (in flowgraph) ntrp-mon-increasing-s:
  \(((s,c),e,(s',c'))\in ntrp fg \implies mon-s fg s \subseteq mon-s fg s' \wedge mon-c fg c \subseteq mon-c fg c'\)
apply (erule gtrp_cases)
apply (auto dest: ntrs-mon-increasing-s simp add: mon-c-unconc)
apply (erule ntrs-c-cases-s)
apply auto

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

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

8.4.3 Modifying the context

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

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

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

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

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

lemma (in flowgraph) ntrp-mon-loc-e-no-ctx:
  \(((s,c),LOC e,(s',c'))\in ntrp fg \implies mon-w fg e \cap mon-c fg c = \{\}\)
by (auto elim!: gtrp_cases dest!: ntrs-mon-e-no-ctx)
The next lemmas are rules how to add or remove threads while preserving the executability of a path

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

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


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), e, (s', c')) \in \text{ntrs fg} \] 

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

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

proof –

from A obtain \( p \ 4 \ 5 \ w \ w \ w \ w \ w \ w \ w \ w \ \text{where} \) \( S: s = u \# r \ ee = \text{LCall p } s' = v \# u' \# r \) 

\( (u \# r, c), \text{LCall p, (entry fg p # u' # r, c))} \in \text{trss fg} \) 

\( ((\text{entry fg p, c, w, (v, c'})) \in \text{trcl (trss fg)} \) by (blast elim!: ntrs_cases[simplified])

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

\( \text{trss fg by auto} \)

moreover from S trss-modify-context[OF S(5)] B obtain \( csp \) where \( c' = csp + c \) \( \text{mon-c fg } csp = \{ \) \( \land ((s, cn), ee, (s', csp + cn)) \in \text{ntrs fg} \) by auto

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

qed

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

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

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

\[ \exists \ csp. \ c' = csp + c \land \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 \((\text{cons } s \ c \ e \ \text{sh } w \ s' \ c')\) note IHP=\(\text{this show } ?\text{case}\)

proof (intro allI simp)

\( \text{fix } cn \)

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

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

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

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

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

\( \in \text{trcl (ntrp fg)} \) by blast

64
proof

qed

lemma ntrs-xchange-context-s:
  assumes A: \((s,c,ee,(s',csp+c))\)\in ntrs fg
  and B: mon-c fg cn \subseteq mon-c fg c
  shows ((s,cn),ee,(s',csp+cn))\in ntrs fg

proof –
  obtain p r u u' v w where S: s=u#r ee=LCall p#w s'=v#u'#r ((u#r,c),LCall p,(entry fg p#u'#r,c))\in trss fg ((\{entry fg p\},c),w,([v],csp+c))\in trcl (trss fg)

proof –
  case goal1 moreover
    from ntrs_cases[OF A, simplified] obtain ce e' p r u u' v w where s = u # r c = ce ee = LCall p # w s' = v # u' # r csp + ce = ce' ((u # r, ce), LCall p, entry_fg p # u' # r, ce) \in trss fg
    (\{entry_fg p\}, ce), w, [v], ce' \in trcl (trss fg).
  hence s = u # r ee = LCall p # w s' = v # u' # r ((u # r, c), LCall p, (entry fg p # u' # r, c)) \in trss fg ((\{entry fg p\}, c), w, ([v], csp+c)) \in trcl (trss fg) by auto

ultimately show \(?thesis\).

qed


qed

lemma ntrs-replace-context-s:
  assumes A: \((s,c+c+cr),ee,(s',c'+cr)\)\in ntrs fg
  and B: mon-c fg crn \subseteq mon-c fg cr
  shows ((s,c+c+cr),ee,(s',c'+cr))\in ntrs fg

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

qed

lemma (in flowgraph) ntrs-xchange-context: \(!!s c c' cn. [\]
  ((s,c),ww,(s',c'))\in trcl (ntrs fg);
  mon-c fg cn \subseteq mon-c fg c
  \]
  \(\Rightarrow\) \exists csp.
  c' = csp + c \wedge ((s,cn),ww,(s',csp+cn))\in trcl (ntrs fg)

proof (induct \(ww\))
  case Nil note CASE=this
  thus \(?case\) by (auto intro!: exI[of - \{\#\}])

next
  case (Cons ee ww) note IHP=this
  then obtain sh ch where SPLIT: ((s,c),ee,(sh,ch))\in ntrs fg ((sh,ch),ww,(s',c'))\in trcl

65
(ntrs fg) by (fast dest; trcl-uncons)
from ntrs-cases-s[OF SPLIT(1)] obtain csp where CHFMT: ch=csp+c
!!s. s:#csp\rightarrow\exists p u v. s=[entry fg p] ∧ (u, Spawn p, v) ∈ edges fg ∧ initialproc fg p by blast
with ntrs-xchange-context-s SPLIT(1) IHP(3) have ((s,cn),ce,(sh,cspch+cn))∈ntrs fg by blast
also
from c-of-initial-no-mon CHFMT(2) have CSPH-NO-MON: mon-c fg csph = {} by auto
with IHP(3) CHFMT have 1: mon-c fg (cspch+cn) ⊆ mon-c fg ch by (auto simp add: mon-c-unconc)
from IHP(1)[OF SPLIT(2) this] obtain csp where C’FMT: c’=csp+ch and SND: ((sh,cspch+cn),ww,(s’,csp+(csph+cn)))∈trcl (ntrs fg) by blast
note SND
finally have ((s, cn), ce ≠ ww, s’, (csp + csph) + cn) ∈ trcl (ntrs fg) by (simp add: union-assoc)
moreover from CHFMT(1) C’FMT have c’=(csp+ch)+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))∈trcl (ntrs fg)
  and B: mon-c fg crn ⊆ mon-c fg cr
  shows ((s,c+crn),ww,(s’,c’+crn))∈trcl (ntrs fg)
proof –
  from ntrs-cases[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 (e+crn) ⊆ mon-c fg (e+cr) by (auto simp add: mon-c-unconc)
  from ntrs-xchange-context[OF A MON] G have ((s,c+crn),ww,(s’,csp+(c+crn)))∈trcl (ntrs fg) by auto
  with F show ?thesis by (simp add: union-assoc)
qed

lemma (in flowgraph) ntr-add-context-s:
  assumes A: (e,e,c’)∈ntr fg
  and B: mon-w fg e ∩ mon-c fg cn = {}
  shows (c+cn,e,c’+cn)∈ntr fg
proof –
  from gtrE[OF A] obtain s ce s’ ce’ where NTRS: c = {#s#} + ce c’ = {#s’#} + ce’ ((s, ce), e, s’, ce’) ∈ ntrs fg .
  from ntrs-mon-e-no-ctx[OF NTRS(3)] B have M: mon-w fg e ∩ (mon-c fg (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))∈ntrs fg by (auto simp add: union-assoc)
  with NTRS show ?thesis by (auto simp add: union-assoc intro: gtr1-s)
qed
lemma (in flowgraph) ntr-add-context:
\[(c,w,c') \in \text{trcl} (\text{ntr} fg) \land \text{mon-ww} fg w \cap \text{mon-c} fg cn = \{\}] \Rightarrow (c+cn, w, c'+cn) \in \text{trcl} (\text{ntr} fg)
by (induct rule: trcl.induct) (simp, force dest: ntr-add-context-s)

lemma (in flowgraph) ntr-add-context-s:
assumes A: \((s,c,e,(s',c')) \in \text{ntr fg}\)
and B: \(\text{mon-w} fg e \cap \text{mon-c} fg cn = \{\}\)
shows \((s,c+cn),e,(s',c'+cn)\) \in \text{ntr fg}
(force simp add: mon-c-unc onc union-ac)

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

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

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

lemma ntrs-stack-top-decomp-s:
  assumes A: \( (u \# r, c), e, (s', c') \in \text{ntrs fg} \)
  and EX: \( \forall u' p. \) \[
        s' = v\#u'\#r; \\
        ((u, c), e, (v, u', c')) \in \text{ntrs fg}; \\
        (u, \text{Call p, u'}) \in \text{edges fg} \]
  \[ \Longrightarrow P \] 
  shows P
  using A
  proof (cases rule: ntrs.cases)
  case ntrs-step then obtain u' v p w where CASE: \( e = \text{LCall p \# w s' = v\#u'\#r} \)
  \( (u \# r, c), \text{LCall p, (entry fg p \# u' \# r, c)} \in \text{trss fg} \) \( ((\text{entry fg p}, c), w, (v, c')) \in \text{trcl (trss fg)} \) by (simp)
  from trss-stack-decomp-s[where \( s = [u] \), simplified, OF CASE(3)] have SDC: \( \left( (\text{[u], c}), \text{LCall p, (entry fg p, u'), c} \right) \in \text{trss fg} \) by auto
  with CASE(1,4) have \( (\text{[u], c}, e, (v, u', c')) \in \text{ntrs fg} \) by (auto intro!: ntrs.ntrs-step)
  moreover from SDC have \( (u, \text{Call p, u'}) \in \text{edges fg} \) by (auto elim!: trss.cases)
  ultimately show \( ?thesis \) using CASE(2) by (blast intro!: EX)
  qed

lemma ntrs-stack-decomp-s:
  assumes A: \( (u \# s \circ r, c), e, (s', c') \in \text{ntrs fg} \)
  and EX: \( \forall u u' p. \) \[
        s' = v\#u'\#r; \\
        ((u, c), e, (v, u', c')) \in \text{ntrs fg}; \\
        (u, \text{Call p, u'}) \in \text{edges fg} \]
  \[ \Longrightarrow P \] 
  shows P
  using A
  proof (cases rule: ntrs.cases)
  case ntrs-step then obtain u' v p w where CASE: \( e = \text{LCall p \# w s' = v\#u'\#r} \)
  \( (u \# r, c), \text{LCall p, (entry fg p \# u' \# r, c)} \in \text{trss fg} \) \( ((\text{entry fg p}, c), w, (v, c')) \in \text{trcl (trss fg)} \) by (simp)
  from trss-stack-decomp-s[where \( s = [u] \), simplified, OF CASE(3)] have SDC: \( \left( (\text{[u], c}), \text{LCall p, (entry fg p, u', c)} \right) \in \text{trss fg} \) by auto
  with CASE(1,4) have \( (\text{[u], c}, e, (v, u', c')) \in \text{ntrs fg} \) by (auto intro!: ntrs.ntrs-step)
  moreover from SDC have \( (u, \text{Call p, u'}) \in \text{edges fg} \) by (auto elim!: trss.cases)
  ultimately show \( ?thesis \) using CASE(2) by (blast intro!: EX)
  qed

68
s'=v#u'#s@r;
((u#s,c),ee,(v#u'#s,c'))∈ntrs fg;
(u,Call p,u')∈edges fg
] ⟹ P
shows P
apply (rule ntrs-stack-top-decomp-s[OF A])
apply (rule EX)
apply (auto dest: ntrs-stack-comp-s)
done

lemma ntrs-stack-decomp: !!u s r c. [(u#s@r,c),ee,(s',c')]}∈trcl (ntrs fg);
!!v rr. [s'=v#rr@r; ((u#s,c),ww,(v#rr,c'))∈trcl (ntrs fg)] ⟹ P
] ⟹ P
proof (induct ww)
case Nil thus ?case by fastforce
next
case (Cons e v) from Cons.prems show ?case proof (cases rule: trcl-pair-unconsE)
case (split sh ch)
from ntrs-stack-decomp-s[OF split(1)] obtain vh uh p where F: sh = vh#uh#s@r
((u#s, c), e, vh#uh#s, ch) ∈ ntrs fg (u, Call p, uh) ∈ edges fg by blast
from F(1) split(2) Cons.hyps[of vh uh#s r ch] obtain v' rr where S:
s'=v'#rr@r ((vh#uh#s,ch),v,(v'#rr,c'))∈trcl (ntrs fg) by auto
don't prove thesis by blast
qed
lemma ntrp-stack-decomp-s:
assumes A: ((u#s@r,c),ee,(s',c'))∈ntrp fg
and EX: !!v rr. [ s'=v#rr@r; ((u#s,c),ee,(v#rr,c'))∈trcl fg ] ⟹ P
shows P
using A
proof (cases rule: gtrp.cases)
case gtrp-loc thus ?thesis using EX by (force elim!: ntrp-stack-decomp-s intr!: gtrp.intros)
next
case gtrp-env then obtain e s s' ce ce' where S: ce=ENV e s'=u#s@r
c={#ss#}+ce c'={#ss'#}+ce'((ss,ce+#u#s@r#),e,(ss',ce'#+#u#s@r#)}∈ntrs fg by (auto simp add: union-ac)
from ntrs-replace-context-s[OF S(5)], where crn=#u#s# have ((ss, #u # s#) + ce, s', #u # s#) ∈ ntrs fg by (auto simp add: mon-s-unconc union-ac)
with S show P by (rule-tac EX) (auto intro: gtrp.gtrp-env)
qed
lemma ntrp-stack-decomp: !!u s r c. [(u#s@r,c),ww,(s',c')]}∈trcl (ntrs fg);
!!v rr. [s'=v#rr@r; ((u#s,c),ww,(v#rr,c'))∈trcl (ntrs fg)] ⟹ P
] ⟹ P

proof \((\text{induct } \mathbf{ww})\)
  case \(\mathbf{Nil}\) thus \(\text{case by fastforce}\) next
  case \((\text{Cons } e \; w)\) from \(\text{Cons.prems}\) show \(\text{case proof (cases rule: trcl-pair-unconsE)}\)
    case \((\text{split } s h \; c h)\)
      from \(\text{ntrp-stack-decomp-s[OF split(1)]}\) obtain \(v h \; r h\) where \(F: s h = v h \# r h @ r ((u # s, e), v h \# r h , c h) \in \text{ntrp fg}\) by blast
      from \(F(1) \text{ split(2)}\) Cons.hyps[of \(v h \; r h \; r \; c h]\) obtain \(v' r r\) where \(S: s' = v' \# r r @ r ((v h \# r h , c h), v (v' # r r , c')) \in \text{trcl (ntrp fg)}\) by auto
      from \(\text{trcl.cons[OF F(2) S(2)]}\) 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 \(\otimes_\alpha\)-operator:
A step on a normalized paths enters the monitors of the first called procedure and passes the monitors that occur in the following same-level path.

\begin{definition}
\(\alpha_n\; fg \; e =\)
\begin{cases}
\{(\{}\},\{\}) & \text{if } e = [\] \\
\text{(mon-e } fg \; (\text{hd } e), \text{mon-w } fg \; (\text{tl } e)) & \text{else}
\end{cases}
\end{definition}

\begin{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)
\end{lemma}

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

\begin{definition}
\(\alpha_nl\; fg \; e =\) \(\alpha_n\; fg \; (\text{le-rem-s } e)\)
\end{definition}

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

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

\begin{lemma}
\(\alpha_nl-simps\) (simp):
\(\alpha_nl\; fg \; (\text{ENV } x) = \alpha_n\; fg \; x\)
\(\alpha_nl\; fg \; (\text{LOC } x) = \alpha_n\; fg \; x\)
by (unfold \(\alpha_nl\)-def, auto)
\end{lemma}

\begin{lemma}
\(\alpha_nl-simpsI\) (simp):
\((\alpha_nl\; fg) \circ \text{ENV} = \alpha_n\; fg\)
\end{lemma}
\[(\alpha n fg) \circ \text{LOC} = \alpha n fg\]
by (unfold \(\alpha n\)-def' \(\text{comp-def}\) \(\text{simp-all}\))

**lemma** \(\alpha n\)-\(\alpha n\): \((\alpha n fg) \circ \text{le-rem-s} = \alpha n fg\)
\[\text{unfolding } \alpha n\)-def\[\text{[symmetric]} .. \]

**lemma** \(\alpha n\)-\(\text{fst-snd}\): \(\text{fst} (\alpha n fg w) \cup \text{snd} (\alpha n fg w) = \text{mon-w fg w}\)
by (induct \(w\)) \(\text{auto}\)

**lemma** mon-pl-of-\(\alpha n\):
\(\text{mon-pl (map (\alpha n fg) w)} = \text{mon-loc fg w} \cup \text{mon-env fg w}\)
by (induct \(w\)) \(\text{auto}\)

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

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

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

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

**lemma** cil-\(\alpha n\)-cons2:
\([w \in wa \otimes_{\alpha n} fg \(wb\); \text{fst} (\alpha n fg e) \cap \text{mon-ww fg \(wa\)} = \{\}]\)
\(\Rightarrow e \# w \in wa \otimes_{\alpha n} fg e \# \(wb\)\)
apply (rule cil-\(\alpha n\)-cons2)
apply assumption
apply (subst cil-\(\alpha n\)-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 \(\text{mon-s fg s'} = \text{mon-s fg s} \cup \text{fst} (\alpha n fg e)\)
proof –
from \(A\) obtain \(u\ r\ p\ u'\ w\ v\) where \(\text{DET: } s=\# u\ e=\text{LCall p\#w (}(\#\# r,c),\text{LCall p, (entry fg p\#u\#r,c)}) \in \text{trss fg (}(\text{entry fg p}\[c,c'),w,((v,c')) \in \text{trcl (trss fg s') = v\#u\#r}\)
by (blast elim!: ntrs_cases[simplified])

hence mon-n fg u = mon-n fg u’ by (auto elim!: trss_cases dest: mon-n-same-proc
edges-part)

with trss-bot-proc-const[where s=[] and s’=[], simplified, OF DET(4)] DET(1,2,5)

show thesis by (auto simp add: mon-n-def αn-def)

qed

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

lemma (in flowgraph) ntr-mon-s:
assumes A: ((s,c),e,(s’,c’))∈ntr fg
shows mon-c fg ({#s#}+c) = mon-c fg (αn fg e)
using ntr-mon-s[OF A] by (unfold αn-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 inter-
leavable and the monitors of the initial configurations are compatible

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

lemma (in flowgraph) ntr-split:
¬[ca cb. [(ca+cb,w,c’)∈trcl (ntr fg); valid fg (ca+cb)] ⇒
\exists ca’ cb’ wa wb.
\c’=ca’+cb’ ∧
\w∈(wa⊗an fgw) ∧
\mon-c fg ca ∩ (mon-c fg cb ∪ mon-ww fg wb) = {} ∧
\mon-c fg cb ∩ (mon-c fg ca ∪ mon-ww fg wa) = {} ∧
\ca,wb,cb’∈trcl (ntr fg)] ∧ (cb,wb,ca’+cb’)∈trcl (ntr fg)

proof (induct w) — The proof is done by induction on the path
— If the path is empty, the lemma is trivial

next

case (Cons e w) note IHP=this
— We split a non-empty paths after the first (macro) step
then obtain \( ch \) where \( \text{SPLIT} \): \( (ca+cb,e,ch) \in \text{ntr} \) \( (ch,w,c') \in \text{trcl} \) \( (ntr fg) \) by \( \text{(fast dest; trcl-uncons)} \)
— Pick the stack that made the first step

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

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

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

from \( \text{ntrs-mon-c-no-cx}[OF \text{NTRS(3)}] \) have \( \text{MONED} \): \( \text{mon-w} \) \( fg \) \( e \cap \text{mon-c} \) \( fg \) \( ce = \{\} \) by \( \text{(auto simp add: mon-c-unconc)} \)
— Needed later: The intermediate configuration is valid

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

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

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

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

case \( \text{left note CASE=this} \)
— We can write the intermediate configuration so that it is suited for the induction hypothesis

with \( \text{CEHFMT NTRS} \) have \( \text{CHFMT} \): \( ch=(\{\#sh\}+csp+(ca-\{\#s\}))+cb \)
by \( \text{(simp add: union-ac)} \)
— and by the induction hypothesis, we split the path from the intermediate configuration

with \( \text{IHP(1)} \) \( \text{SPLIT(2)} \) \( \text{CHVALID} \) obtain \( ca' \) \( cb' \) \( wa \) \( wb \) where \( \text{IHAPP} \):
\( c'=ca'+cb' \)
\( w \in wa \ominus an \) \( fn \) \( wb \)
\( \text{mon-c} \) \( fg \) \( ((\#sh\})+csp+(ca-\{\#s\})) \cap (\text{mon-c} \) \( cb \cup \text{mon-ww} \) \( fg \) \( wb)=\{\} \)
\( \text{mon-c} \) \( fg \) \( cb \cup (\text{mon-c} \) \( ((\#sh\})+csp+(ca-\{\#s\})) \cup \text{mon-ww} \) \( fg \) \( wa)=\{\} \)
\( ((\#sh\})+csp+(ca-\{\#s\}),wa,ca') \in \text{trcl} \) \( (ntr fg) \)
\( (cb,wb,cb') \in \text{trcl} \) \( (ntr fg) \)
by \( \text{blast} \)
moreover
— It remains to show that we can execute the first step with the right part of the configuration removed

have \( \text{FIRSTSTEP} \): \( (ca,e,\{\#sh\}+csp+(ca-\{\#s\})) \in \text{ntr} \) \( fg \)
proof —
from \( \text{CASE(2)} \) have \( \text{mon-c} \) \( fg \) \( (ca-\{\#s\}) \subseteq \text{mon-c} \) \( fg \) \( ce \) by \( \text{(auto simp add: mon-c-unconc)} \)
with \( \text{ntrs-xchange-context-s NTRS(3)} \) \( \text{CEHFMT CASE(2)} \) have \( ((s,ca-\{\#s\}),e,(sh,csp+(ca-\{\#s\}))) \)
\( fg \) by \( \text{blast} \)
from \( \text{gtrI-s[OF this]} \) \( \text{CASE(1)} \) show \( ?\text{thesis by} \) \( \text{(auto simp add: union-assoc)} \)
qed
with \textit{IHAPP}(5) have \((\textit{ca},\textit{e}\#\textit{wa},\textit{ca'})\in\textit{trcl}(\textit{ntr}\ \textit{fg})\) by \textit{simp}
moreover
\quad and that we can prepend the first step to the interleaving
have \(\textit{e}\#\textit{w} \in \textit{e}\#\textit{wu} \otimes_{\textit{an}} \textit{fg} \ \textit{wb}\)
proof
\quad from \textit{ntrs-called-mon}[OF \textit{NTRS}(3)] have \(\textit{fst}(\textit{an} \ \textit{fg} \ \textit{c}) \subseteq \textit{mon-s} \ \textit{fg} \ \textit{sh}\).
\quad with \textit{IHAPP}(3) have \(\textit{fst}(\textit{an} \ \textit{fg} \ \textit{c}) \cap \textit{mon-fw} \ \textit{fg} \ \textit{wb} = \{\} \) by (auto simp add: \textit{mon-c-unconc})
\quad from \textit{cil-an-cons}[OF \textit{IHAPP}(2) \ this] \textit{show} \ ?\textit{thesis} .
qed
moreover
\quad and that the monitors of the initial context does not interfere
have \(\textit{mon-c} \ \textit{fg} \ \textit{ca} \cap (\textit{mon-c} \ \textit{fg} \ \textit{cb} \cup \textit{mon-fw} \ \textit{fg} \ \textit{wb}) = \{\} \ \textit{mon-c} \ \textit{fg} \ \textit{cb} \cap (\textit{mon-c} \ \textit{fg} \ \textit{ca} \cup \textit{mon-fw} \ \textit{fg} \ (e\#\textit{wa})) = \{\}
\quad \textit{proof}
\quad \quad \textit{from} \textit{ntr-mon-increasing-s}[OF \textit{FIRSTSTEP}] \textit{IHAPP}(3) \ \textit{show} \ \textit{mon-c} \ \textit{fg} \ \textit{ca}
\quad \quad \cap (\textit{mon-c} \ \textit{fg} \ \textit{cb} \cup \textit{mon-fw} \ \textit{fg} \ \textit{wb}) = \{\} \ \textit{by} \ \textit{auto}
\quad \quad \textit{from} \textit{MONED} \ \textit{CASE} \ \textit{have} \ \textit{mon-c} \ \textit{fg} \ \textit{cb} \cap \textit{mon-w} \ \textit{fg} \ \textit{e} = \{\} \ \textit{by} \ \textit{(auto simp add: mon-c-unconc)}
\quad \quad \textit{with} \textit{ntr-mon-increasing-s}[OF \textit{FIRSTSTEP}] \textit{IHAPP}(4) \ \textit{show} \ \textit{mon-c} \ \textit{fg} \ \textit{cb}
\quad \quad \cap (\textit{mon-c} \ \textit{fg} \ \textit{ca} \cup \textit{mon-fw} \ \textit{fg} \ (e\#\textit{wa})) = \{\} \ \textit{by} \ \textit{auto}
qed
ultimately \textit{show} \ ?\textit{thesis} \ \textit{by} \ \textit{blast}
next
\quad 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
\textbf{case} right \ textbf{note} \ \textit{CASE}=\textit{this}
\quad \textit{with} \ \textit{CEHFMT} \ \textit{NTRS} \ \textit{have} \ \textit{CHFMT}: \textit{ch}=\textit{ca}+\{(\#\textit{sh}\#)+\textit{csp}+(\textit{cb}-(\#\textit{s}\#))\}
\quad by (simp add: \textit{union-ac})
\quad \textit{with} \ \textit{IHAPP}(1) \ \textit{SPLIT}(2) \ \textit{CHVALID} \ \textit{obtain} \ \textit{ca'} \ \textit{cb}' \ \textit{wa} \ \textit{wb} \ \textit{where} \ \textit{IHAPP}: \textit{c}'=\textit{ca}'+\textit{cb}' \ \textit{w}\otimes_{\textit{an}} \textit{fg} \ \textit{wb} \ \textit{mon-c} \ \textit{fg} \ \textit{ca} \cap (\textit{mon-c} \ \textit{fg} \ (\{(\#\textit{sh}\#)+\textit{csp}+(\textit{cb}-(\#\textit{s}\#))\})
\quad \cup \textit{mon-fw} \ \textit{fg} \ \textit{wb})=\{\}
\quad \ \textit{mon-c} \ \textit{fg} \ ((\{(\#\textit{sh}\#)+\textit{csp}+(\textit{cb}-(\#\textit{s}\#))\}) \cap (\textit{mon-c} \ \textit{fg} \ \textit{ca} \cup \textit{mon-fw} \ \textit{fg} \ \textit{wa})=\{\}
\quad \ \textit{(ca,wa,ca')\in trcl (nf}\ \textit{fg}) ((\{(\#\textit{sh}\#)+\textit{csp}+(\textit{cb}-(\#\textit{s}\#))\}),\textit{wb},\textit{cb}')\in\textit{trcl (ntr}\ \textit{fg})
\quad \quad by \ \textit{blast}
\quad \textit{moreover}
\quad \quad \textit{have} \ \textit{FIRSTSTEP}: \textit{(cb},\textit{e},\{(\#\textit{sh}\#)+\textit{csp}+(\textit{cb}-(\#\textit{s}\#))\})\in\textit{ntr}\ \textit{fg} \ \textit{proof}
\quad \quad \quad \textit{from} \ \textit{CASE}(2) \ \textit{have} \ \textit{mon-c} \ \textit{fg} \ (\textit{cb}-(\#\textit{s}\#)) \subseteq \textit{mon-c} \ \textit{fg} \ \textit{ce} \ \textit{by} \ \textit{(auto simp add: mon-c-unconc)}
\quad \quad \textit{with} \ \textit{ntrs-xchange-context-s} \ \textit{NTRS}(3) \ \textit{CEHFMT} \ \textit{CASE}(2) \ \textit{have} (\textit{s},\textit{cb}-(\#\textit{s}\#),\textit{e},(\textit{sh},\textit{csp}+(\textit{cb}-(\#\textit{s}\#))))\in\textit{fg} \ \textit{by} \ \textit{blast}
\quad \quad \textit{from} \ \textit{gtr1-s}[OF \ this] \ \textit{CASE}(1) \ \textit{show} \ ?\textit{thesis} \ \textit{by} \ \textit{(auto simp add: union-assoc)}
\quad \textit{qed}
\quad \textit{with} \ \textit{IHAPP}(6) \ \textit{have} \ \textit{PA}: (\textit{cb},\textit{e}\#\textit{wb},\textit{cb'})\in\textit{trcl (ntr}\ \textit{fg}) \ \textit{by} \ \textit{simp}
\quad \textit{moreover}
\quad \quad \textit{have} \ \textit{e}\#\textit{w} \in \textit{wa} \otimes_{\textit{an}} \textit{fg} \ \textit{e}\#\textit{wb}
\quad \quad \textit{proof}
\quad \quad \quad \textit{from} \ \textit{ntrs-called-mon}[OF \ \textit{NTRS}(3)] \ \textit{have} \ \textit{fst}(\textit{an} \ \textit{fg} \ \textit{c}) \subseteq \textit{mon-s} \ \textit{fg} \ \textit{sh} .
74
with IHAPP(4) have \( \text{fst}(\alpha_n \text{fg} e) \cap \text{mon-wf \ fg \ wa} = \{\} \) by (auto simp add: mon-c-unconc)
from cil-an-cons2[of IHAPP(2) this] show \( \text{thesis} \).
qed

moreover
have \( \text{mon-c \ fg \ cb} \cap (\text{mon-c \ fg \ ca} \cup \text{mon-wf \ fg \ wa}) = \{\} \) by auto
proof -
  from ntr-mon-increasing-s[of FIRSTSTEP] IHAPP(4) show \( \text{mon-c \ fg \ cb} \cap (\text{mon-c \ fg \ ca} \cup \text{mon-wf \ fg \ wa}) = \{\} \) by auto
  from MONED CASE have \( \text{mon-c \ fg \ ca} \cap \text{mon-wf \ fg \ e} = \{\} \) by (auto simp add: mon-c-unconc)
  with ntr-mon-increasing-s[of FIRSTSTEP] IHAPP(3) show \( \text{mon-c \ fg \ ca} \cap (\text{mon-c \ fg \ cb} \cup \text{mon-wf \ fg \ (e\#w)})) = \{\} \) by auto
qed
ultimately show \( \text{thesis} \) by blast
qed

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

**lemma** (in flowgraph) ntrp-split:

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

\[ \Rightarrow \exists w_1 w_2 c_1' c_2'. \]
\[ w \in w_1 \otimes \text{and \ fg} (\text{map \ ENV \ w2}) \land \
  c' = c_1' + c_2' \land \
  ((s,c_1),w_1,(s',c_1'))\in\text{trcl}(\text{ntrp \ fg}) \land \
  (c_2,w_2,c_2')\in\text{trcl}(\text{ntrf \ fg}) \land \
  \text{mon-wf \ fg} (\text{map \ le-rem-s \ w1}) \cap \text{mon-c \ fg} c_2 = \{\} \land \
  \text{mon-wf \ fg} w_2 \cap \text{mon-c \ fg} (\{\#s\#\}+c1) = \{\}
\]

**proof** (induct w)
\begin{itemize}
  \item **case** Nil thus \( \exists \text{case} \) by (auto intro: ezI[of - []] ezI[of - {#}])
  \item **next**
  \begin{itemize}
    \item **case** (Cons ee w) then obtain sh ch where SPLIT: \((s,c_1+c_2),ee,(sh,ch))\in\text{ntrp \ fg}((sh,ch),w,(s',c'))\in\text{trcl}(\text{ntrp \ fg}) \text{ by (fast dest: trcl-uncons)}
    \item from SPLIT(1) show \( \exists \text{case} \) proof (cases rule: gtrp-cases)
    \begin{itemize}
      \item **case** gtrp-loc then obtain e where CASE: \( ee = \text{LOC} e ((s,c_1+c_2),e,(sh,ch))\in\text{ntrp \ fg} \text{ by auto}
      \item from ntr-loc-cases-s[of CASE(2)] obtain csp where CHFMT: \( ch = (csp+c_1)+c_2 \land s : # \ csp \Rightarrow \exists p u v. (s = \text{entry} \ p) \land (u, \text{Spawn} p, v) \in \text{edges} \ fg \land \text{initialproc} \ fg \ p \text{ by (simp add: union-assoc, blast)}
      \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}
(c2, w2, c2') ∈ trcl (ntrp fg) \( \text{mon-ww fg } (\text{map le-rem-s w1}) \cap \text{mon-c fg c2} = \{ \} \) \( \text{mon-ww fg w2 } \cap \text{mon-c fg } \{ \#s\# \} + (\text{csp } + c1) = \{ \} \) by blast

\text{have ee#w } \in \text{ee#w1 } \circ \text{antl fg } (\text{map ENV w2}) \text{ proof (rule cil-cons1)}

\text{from ntrp-mon-env-w-no-ctx[OF SPLIT(2), unfolded mon-env-def] have mon-ww fg } (\text{map le-rem-s } (\text{env w})) \cap \text{mon-s fg sh } = \{ \} .

\text{moreover have mon-ww fg w2 } \subseteq \text{mon-ww fg } (\text{map le-rem-s } (\text{env w})) \text{ proof }

\text{from cil-subset-il IHAPP(1) uleq-interleave have map ENV w2 } \preceq w \text{ by blast}

\text{from le-list-filter[OF this] have env } (\text{map ENV w2}) \preceq w \text{ by (unfold env-def) 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 } (\text{env w}) \text{ by simp}

\text{thus ?thesis by (rule mon-ww-ileq)}

\text{qed}

\text{ultimately have mon-ww fg w2 } \cap \text{mon-s fg sh } = \{ \} \text{ by blast}

\text{with ntrp-mon-s[OF CASE(2)] CASE(1) show \text{fst } (\text{antl fg ee }) \cap \text{mon-pl (map (\text{antl fg }) (\text{map ENV w2})) } = \{ \} \text{ by (auto simp add: cil-cn-cns-helper)}

\text{qed (rule IHAPP(1))}

\text{moreover have } ((s,c1), ee#w1, (s',c1')) \in trcl (\text{ntrp fg}) \text{ proof –}

\text{from ntrp-exchange-context-s[\text{of s c1+c2 e sh csp } f] c1] CASE(2) CHFMT(1) have } ((s, c1), e, sh, csp + c1) \in ntrp fg \text{ by (auto simp add: mon-c-unconc union-ac)}

\text{with CASE(1) have } ((s, c1), ee, sh, csp + c1) \in ntrp fg \text{ by (auto intro: gtrp,grtp-loc)}

\text{also note IHAPP(3)}

\text{finally show ?thesis .}

\text{qed}

\text{moreover from CASE(1) ntrp-mon-c-no-ctx[OF CASE(2)] IHAPP(5) have mon-ww fg } (\text{map le-rem-s } (ee#w1)) \cap \text{mon-c fg c2 } = \{ \} \text{ by (auto simp add: mon-c-unconc)}

\text{moreover from ntrp-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: e=ENV e c1+c2=\{\#ss\#\}+ce sh=s ch=\{\#ssh\#\}+ceh ((ss,\{\#s\#\}+ce),e,(ssh,\{\#s\#\}+ceh))}\in ntrp fg by auto}

\text{from ntrp-c-cases-s[OF CASE(3)] obtain csp where HFMT: \{\#s\#\}+ceh = csp + ((\{\#s\#\}+ceh \wedge s :\# csp \implies \exists p u v. s = [\text{entry fg p } ] \wedge (u, \text{Spawn p, v }) \in \text{edges fg } \wedge \text{initialproc fg p p by (blast)}}

\text{from union-left-cancel[\text{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 = \{ \} by (blast intro!: c-of-initial-no-mon)}

\text{from CASE(2)[symmetric] show ?thesis proof (cases rule: mset-unplsm-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 HFMT(1) CASE(4) CEHFMT have CHFMT': ch = (csp + {#ssh#} + (c1 - {#ss#})) + c2 by (simp add: union-ac)
  have VALID: valid fg (\{#ss#\} + (csp + {#ssh#} + (c1 - {#ss#})) + c2)
proof -
  from ntr-valid-preserve-s[OF gtrI-s, OF CASE(5)] Cons.prems(2) CASE(2)
  have valid fg (\{#ssh#\} + (\{#ss#\} + ceh)) by (simp add: union-assoc) (auto simp add: union-ac)
  with left CEHFMT show \(\text{?thesis}\) by (auto simp add: union-ac)

qed

from Cons.hyps[OF - VALID.of s' \(\cap\) CHFMT' SPLIT(2) CASE(3)] obtain
w1 w2 c1' c2' where IHAPP: \(\forall w \in w1 \circ_{\circ w2} \circ\text{ map ENV w2} = c1' + c2'
\((s, csp + \{#ssh#\} + (c1 - \{#ss#\}))\), \(\forall w \in w1, s', c1') \in \text{ trcl (ntrp fg)} (c2, w2, c2') \in \text{ trcl (ntr fg)}

\text{mon-wv \(\forall fg (\text{map le-ren-s w1}) \cap \text{ mon-c fg c2} = \{\} \)} \text{ mon-wv \(\forall fg w2 \cap \text{ mon-c fg (\{#ss#\} + (csp + \{#ssh#\} + (c1 - \{#ss#\}))) = \{\} \)} by blast

have ee \(\forall w \in (ee \# w1) \circ_{\circ w2} \circ\text{ map ENV w2} \text{ proof (rule cil-cons1)}

from IHAPP(6) have mon-wv \(\forall fg w2 \cap \text{ mon-s fg ssh} = \{\} \) by (auto simp add: mon-c-unconc)

moreover from ntrs-mon-s[of CASE(5)] CASE(1) have \(\forall fg ee \subseteq \text{ mon-s fg ssh by auto}

ultimately have \(\forall\) \(\forall fg ee \cap \text{ mon-wv fg w2 = \{\}} \) by auto

moreover have \(\forall\) \(\forall mon-pl (\text{map (\forall fg) (\text{map ENV w2}))) = \text{ mon-wv fg w2}

by (simp add: cil-\alpha-cons-helper)

ultimately show \(\forall\) \(\forall fg ee \cap \text{ mon-pl (\text{map (\forall fg) (\text{map ENV w2}) = \{\}) by auto}

qed (rule IHAPP(1))

moreover

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

from left HFMT(1) have \(\forall\) \(\forall\) \(\forall\) \(\forall ceh = (csp + \{#ssh#\} + (c1 - \{#ss#\}) + c2 \{#ss#\} + ceh

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

with CASE(5) ntrs-exponent-context-s[of ss \(\{#ss#\} + (c1 - \{#ss#\}) + c2 \text{ e ssh csp fg ((c1 - \{#ss#\}) + c2 \text{ e ssh}}

\text{cee = (s, ss, \{#ss#\}) + ceh, ssh, \{#ss#\} + (csp + (c1 - \{#ss#\})) \in ntrs fg by (auto simp add: mon-c-unconc union-ac)}

from gtrp.gtrp-env[of this] left(1)[symmetric] CASE(1) show \(\text{?thesis}\) by (simp add: union-ac)

qed

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

moreover

from ntrs-mon-c-no-context[of CASE(5)] left CASE(1) IHAPP(5) have \(\forall\) \(\forall\) \(\forall\) \(\forall mon-wv fg (\text{map le-ren-s (ee \# w1)}) \cap \text{ mon-c fg c2 = \{\}} by (auto simp add: mon-c-unconc)

moreover

from ntrp-mon-increasing-s[of SS] IHAPP(6) have \(\forall\) \(\forall\) \(\forall\) \(\forall mon-wv fg w2 \cap \text{ mon-c fg ((c1 - \{#ss#\}) + c1 = \{\}} by (auto simp add: mon-c-unconc)

moreover note 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

\textbf{case right}

\textbf{with HFMT(1) CASE(4) CEHFMT have CHFMT':} \( cf = c1 + (csp + \{ssh\} + (c2 - \{ss\})) \)

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

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

\textbf{proof –}

from ntr-valid-preserve-s[OF gtrI-s, OF CASE(5)] Cons.prems(2) CASE(2)

\textbf{have valid fg \((\{ssh\} + ((\{ss\} + ceh))\) by (simp add: union-assoc) (auto simp add: union-ac)}

\textbf{with right CEHFMT show \(?thesis\) by (auto simp add: union-ac)}

\textbf{qed}

from Cons.hyps[OF - VALID, OF s' c'] CHFMT' SPLIT(2) CASE(3) obtain

\textbf{w1 w2 c1' c2'} where IHAPP: \(w \in w1 \otimes_{all fg} map\ ENV\ w2\ c' = c1' + c2'\)

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

\(w2, c2') \in \text{trcl (ntr fg)}\)

\textbf{mon-ww fg (map le-rem-s w1) \cap mon-c fg (csp + \{ssh\} + (c2 - \{ss\})) = \{\}}

\textbf{have \(ee \# w \in w1 \otimes_{all fg} map\ ENV\ (c\# w)\) proof (simp add: CASE(1), rule c1-cons2)}

\textbf{from IHAPP(5) have mon-ww fg (map le-rem-s w1) \cap mon-s fg ssh = \{\}}

\textbf{by (auto simp add: mon-c-unconc)}

\textbf{moreover from ntrs-mon-s[OF CASE(5)] CASE(1) have \(fst (anl fg ee) \subseteq mon-s fg ssh\) by auto}

\textbf{ultimately have \(fst (anl fg ee) \cap mon-ww fg (map le-rem-s w1) = \{\}}\ by auto

\textbf{moreover have mon-pl (map (anl fg) w1) = mon-ww fg (map le-rem-s w1) by (unfold anl-def') (simp add: c1-an-cons-helper[symmetric])}

\textbf{ultimately show \(fst (anl fg (ENV e)) \cap mon-pl (map (anl fg) w1) = \{\)}\ by auto

\textbf{using CASE(1) by auto}

\textbf{qed (rule IHAPP(1))}

\textbf{moreover}

\textbf{have \(SS: (c2, c, csp + \{ssh\} + (c2 - \{ss\}))\in ntrp fg\) proof –}

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

\textbf{with CASE(5) ntrs-xchange-context-s[\{SSF\, \{s\} = \{s\} + c1 + (c2 - \{ss\})\} e ssh csp fg c2 - \{ss\}\) have}

\((ss, (c2 - \{ss\}), e, ssh, csp + (c2 - \{ss\}))\) \in ntrs fg by (auto simp add: mon-c-unconc union-ac)

\textbf{from gtrI-s[OF this] right(1)[symmetric] show \(?thesis\) by (simp add: union-ac)}

\textbf{qed}

\textbf{from trcl.cons[OF this IHAPP(4)] have \((c2, e \# w2, c2') \in \text{trcl (ntr fg)}\)}

\textbf{moreover}

\textbf{from ntr-mon-increasing-s[OF SS] IHAPP(5) have mon-ww fg (map le-rem-s w1) \cap mon-c fg c2 = \{\)}\ by (auto simp add: mon-c-unconc)

\textbf{moreover}
from ntrs-mon-e-no-cxts[OF CASE(5)] right IHAPP(6) have mon-ww fg (e#w2) ∩ mon-c fg ((#s#) + c1) = {} by (auto simp add: mon-c-unconc)
moreover note IHAPP(2,3)
ultimately show ?thesis by blast
qed
qed

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

lemma (in flowgraph) ntr-split':
assumes A: (ca+cb,w,c')∈trcl (ntr fg)
and VALID: valid fg (ca+cb)
shows ∃ ca' cb' wa wb.
c' = ca' + cb' ∧
w ∈ (wa ⊗ anl fg wb) ∧
mon-c fg ca ∩ (mon-c fg cb ∪ mon-ww fg wb) = {} ∧
mon-c fg cb ∩ (mon-c fg ca ∪ mon-ww fg wa) = {} ∧
(ca,wa,cb)∈trcl (ntr fg) ∧
(cb,wb,c')∈trcl (ntr fg)
proof (cases ca)
case empty with A show ?thesis by (fastforce intro: exI[of - {}] exI[of - []])
next
case (add s cae) hence CASE: ca+cb={#s#}+(cae+cb) by (simp add: union-ac)

with ntrs.gtr2gtrp A obtain s' ce' ww where NTRP: c'={#s'#}+ce' w=map le-rem-s w1 ((s,cae+cb),ww,(s',ce'))∈trcl (ntrp fg) by (simp, blast)
from ntrp-split[OF NTRP(3)] VALID CASE obtain w1 w2 c1' c2' where LEM:

ww ∈ w1 ⊗ anl fg map ENV w2 ce' = c1' + c2' ((s, cae), w1, s', c1') ∈ trcl (ntrp fg) (cb, w2, c2') ∈ trcl (ntr fg) mon-ww fg (map le-rem-s w1) ∩ mon-c fg cb = {}
mon-ww fg w2 ∩ mon-c fg (#s#) + cae = {}
by (auto simp add: union-ac)
from cil-map[OF LEM(1)[unfolded anl-def''] NTRP(2) have w ∈ map le-rem-s w1 ⊗ anl fg w2 by auto

moreover from gtrp2gtrp[OF LEM(3)] add NTRP(1) LEM(2) have (ca,map le-rem-s w1,#s'#)+c1'∈trcl (ntr fg) c'=((#s'#)+c1') + c2' by (simp-all add: union-ac)
moreover from add LEM(6) have mon-c fg ca ∩ mon-ww fg w2 = {} by (auto simp add: union-ac)
with VALID have mon-c fg ca ∩ (mon-c fg cb ∪ mon-ww fg w2) = {} by (auto simp add: valid-unconc)
moreover from VALID LEM(5) have mon-c fg cb ∩ (mon-c fg ca ∪ mon-ww fg (map le-rem-s w1))={} by (auto simp add: valid-unconc)
moreover note LEM(4)
ultimately show ?thesis by blast
qed

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:** $A : w \in \text{wa} \otimes \alpha \ntr fg \text{wb}$ and

$B : (ca, wa, ca') \in \text{trcl} (\ntr fg)$

$(cb, wb, cb') \in \text{trcl} (\ntr fg)$

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

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

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

**Proof:**

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

**From A have** $\forall \text{ ca cb } (ca, wa, ca') \in \text{trcl} (\ntr fg)$ \& $(cb, wb, cb') \in \text{trcl} (\ntr fg)$ \&

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

**Proof (induct rule: cil-set-induct-fixa)**

— If both words are empty, the proposition is trivial

**Case empty thus ?case by simp**

**Next**

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

**Case (left e w' w1' w2) thus ?case proof (intro allI impI)**

**Case (goal1 ca cb) hence I : w' ∈ w1' ⊗ α n tr w2 fst (α fg e) ∩ mon-pl (map (α fg) w2) = {}**

!!ca cb.

$[(ca, w1', ca') \in \text{trcl} (\ntr fg);$

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

$\text{mon-c fg ca } \cap (\text{mon-c fg cb } \cup \text{mon-ww fg w2}) = \{\};$$\text{mon-c fg cb } \cap (\text{mon-c fg ca } \cup \text{mon-ww fg w1'}) = \{\}] \implies$

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

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

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

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

| — Split the left path after the first step

| **Then obtain** cah where **SPLIT**: $(ca, e, cah) \in \text{trcl} (\ntr fg) (ca, w1', ca') \in \text{trcl} (\ntr fg)$ **by** (fast dest: trcl-uncons)

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

| **From ntr-add-context-s[OF SPLIT(1)], where cn=cb** I(7) have $(ca + cb, e, cah + cb) \in \text{trcl (ntr fg)}$ **by** auto

| **Also**

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

| **Have** $(ca + cb, w', ca' + cb') \in \text{trcl} (\ntr fg)$ **Proof —**

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

| **With I(7) have** MON-CB: $\text{mon-c fg cb } \cap (\text{mon-c fg cah } \cup \text{mon-ww fg w1'})$
this case is done completely analogous

\[ \text{case (right e w' w2' w1)} \quad \text{thus ?case proof (intro allI impI)} \]

\[ \text{case (goal1 ca cb) hence I: w' \in w1 \otimes_{\alpha_f} f_g w2' \text{fst (an fg e) \cap mon-pl (map (an fg) w1) = \{}} \]

!! ca cb.

\[ \[(ca, w1, ca') \in \text{trcl (ntr fg)}; \]
\[ (cb, w2', cb') \in \text{trcl (ntr fg)}; \]
\[ \text{mon-c fg \cap (mon-c fg cb \cup mon-ww fg w2') = \{}}; \]
\[ \text{mon-c fg cb \cap (mon-c fg ca \cup mon-ww fg w1) = \{}} \]
\[ (ca + cb, w', ca' + cb') \in \text{trcl (ntr fg)} \]
\[ (ca, w1, ca') \in \text{trcl (ntr fg)} (cb, e#w2', cb') \in \text{trcl (ntr fg)} \]
\[ \text{mon-c fg ca \cap (mon-c fg cb \cup mon-ww fg (e#w2')) = \{}} \]
\[ \text{mon-c fg cb \cap (mon-c fg ca \cup mon-ww fg w1) = \{}} \quad \text{by blast +} \]

\[ \text{then obtain cbh where SPLIT: (cb,e,cbh)\in ntr fg (cbh,w2',cb')\in trcl (ntr fg) by (fast dest: trcl-uncons)} \]

\[ \text{from ntr-add-context-s[OF SPLIT(1), where cn=ca] I(6) have (ca + cb, e, ca + cbh) \in ntr fg by (auto simp add: union-commute)} \]

\[ \text{also have (ca + cbh, w', ca' + cb') \in trcl (ntr fg) proof --} \]

\[ \text{from I(2,6,7) ntr-mon-s[OF SPLIT(1)] have MON-CBH: mon-c fg cbh \cap (mon-c fg ca \cup mon-ww fg w1) = \{}} \text{ by (cases e) (auto simp add: cil-\alpha-cons-helper) with I(6) have MON-CA: mon-c fg ca \cap (mon-c fg cbh \cup mon-ww fg w2') = \{}} \text{ by auto} \]

\[ \text{from I(3)|OF I(4) SPLIT(2) MON-CA MON-CBH] show ?thesis .} \]

\text{qed}

\text{finally show ?case .} \]

\text{qed}

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

\text{qed}

\text{lemma (in flowgraph) ntrp-unsplit:} \]

\text{assumes A: w} \in \text{wa\otimes_{\alpha_f} f_g (map ENV wb) and} \]

\text{B: ((s,ca),wa,(s',ca'))\in trcl (ntrp fg)} \]

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

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

\text{mon-c fg cb \cap (mon-c fg ((s#)\cap (mon-c fg (map le-rem-s wa))\cap mon-ww fg (map le-rem-s wa))))=\{}} \]

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

\text{proof --} \]

\text{fix wb'} \]

\text{have w} \in \text{wa\otimes_{\alpha_f} f_g wb'} \quad \text{\Rightarrow}
\[
\forall s\ ca\ cb\ wb.\ wb' = \map\ ENV\ wb \land \\
(s, ca, wa, (s', ca')) \in \trcl\ (ntrp\ fg) \land (cb, wb, cb') \in \trcl\ (ntr\ fg) \land \mon{c}\ fg \\
\(((\#\#) + ca) \cap (\mon{c}\ fg\ cb \cup \mon{ww}\ fg\ wb) = \{} \land \mon{c}\ fg\ cb \cap (\mon{c}\ fg \\
\(((\#\#) + ca) \cup \mon{ww}\ fg\ (\map\ le-rem-s\ wa)) = \{} \rightarrow \\
(s, ca + cb, w, (s', ca' + cb')) \in \trcl\ (ntrp\ fg)\] \\
\textbf{proof (induct rule: cil-set-induct-fixa)} \\
\textbf{case empty thus \ ?case by simp} \\
\textbf{next} \\
\textbf{case (left e w' w1' w2) thus \ ?case proof (intro allI impl1)} \\
\textbf{case (goal1 s ca cb wb) hence I: w' \in w1' \otimes_{\text{allI}} w2\ fst (\text{allI} f g e) \cap \\
\mon{pl}\ (\map\ (\text{allI} f g) w2) = \{\} \\
!!s\ ca\ cb\ wb. [ \\
w2 = \map\ ENV\ wb; \\
((s, ca), w1', s', ca') \in \trcl\ (ntrp\ fg); \\
(cb, wb, cb') \in \trcl\ (ntr\ fg); \\
\mon{c}\ fg\ (((\#\#) + ca) \cap (\mon{c}\ fg\ cb \cup \mon{ww}\ fg\ wb) = \{\}; \\
\mon{c}\ fg\ cb \cap (\mon{c}\ fg\ (((\#\#) + ca) \cup \mon{ww}\ fg\ (\map\ le-rem-s \\
w1'))) = \{\} \\
] \rightarrow (s, ca + cb), w', s', ca' + cb') \in \trcl\ (ntrp\ fg)\] \\
w2 = \map\ ENV\ wb \\
((s, ca), e \neq w1', s', ca') \in \trcl\ (ntrp\ fg) \\
(cb, wb, cb') \in \trcl\ (ntr\ fg) \\
\mon{c}\ fg\ (((\#\#) + ca) \cap (\mon{c}\ fg\ cb \cup \mon{ww}\ fg\ wb) = \{\}; \\
\mon{c}\ fg\ cb \cap (\mon{c}\ fg\ (((\#\#) + ca) \cup \mon{ww}\ fg\ (\map\ le-rem-s (e \neq \\
w1')))) = \{\} \\
by\ blast+ \\
\textbf{then obtain sh\ cah\ where SPLIT: ((s, ca), e, (sh, cah)) \in ntrp\ fg\ ((sh, cah), w1', s', ca')) \in \trcl \\
(ntrp\ fg)\ by\ (fast\ dest;\ trcl-uncos) \\
\textbf{from\ ntrp-add-context-s\ (OF\ SPLIT(1), of\ cb)\ I(8)\ have\ ((s, ca + cb), e, \\
sh, cah + cb) \in ntrp\ fg\ by\ auto} \\
\textbf{also have\ ((sh, cah + cb), w', s', ca'+ cb') \in \trcl\ (ntrp\ fg)\ proof (rule I(3))} \\
\textbf{from\ ntrp-mon-s\ (OF\ SPLIT(1))\ I(2, 4, 7, 8)\ show\ 1: \mon{c}\ fg\ (((\#\#) + cah) \cap (\mon{c}\ fg\ cb \cup \mon{ww}\ fg\ wb) = \{\} \\
by\ (cases e)\ (rename-tac\ a, case-tac\ a, simp\ add:\ cil-\text{allI}\ cons-helper, \\
fastforce\ simp\ add: cil-\text{allI}\ cons-helper)+ \\
\textbf{from\ I(8)\ I\ show\ \mon{c}\ fg\ cb \cap (\mon{c}\ fg\ (((\#\#) + cah) \cup \mon{ww}\ fg\ (\map\ le-rem-s\ w1')) = \{\} \by auto \\
\textbf{qed\ (auto\ simp\ add: I(4, 6)\ SPLIT(2))} \\
\textbf{finally show \ ?case .} \\
\textbf{qed} \\
\textbf{next} \\
\textbf{case (right ee w' w2' w1) thus \ ?case proof (intro allI impl1)} \\
\textbf{case (goal1 s ca cb wb) hence I: w' \in w1 \otimes_{\text{allI}} w2\ fst (\text{allI} f g e) \cap \\
\mon{pl}\ (\map\ (\text{allI} f g) w1) = \{\} \\
!!s\ ca\ cb\ wb. [ \\
w2' = \map\ ENV\ wb; \\
((s, ca), w1, s', ca') \in \trcl\ (ntrp\ fg); \\
(cb, wb, cb') \in \trcl\ (ntr\ fg); \\
\mon{c}\ fg\ (((\#\#) + ca) \cap (\mon{c}\ fg\ cb \cup \mon{ww}\ fg\ wb) = \{\}; \\
\textbf{next page}
\[ mon\_c\ fg\ cb \cap (mon\_c\ fg\ (\#s\#) + ca) \cup mon\_ww\ fg\ (map\ le\_rem\_s\ w1) = \{ \} \]

\[ \exists (s, ca + cb), w', s', ca' + cb') \in trcl (ntrp\ fg) \]

\[ ee\#w2' = map\ ENV\ wb \]
\[ (ee, ca), w1, s', ca') \in trcl (ntrp\ fg) \]
\[ (cb, wb, cb') \in trcl (ntrp\ fg) \]
\[ mon\_c\ fg\ ((\#s\#) + ca) \cap (mon\_c\ fg\ cb \cup mon\_ww\ fg\ wb) = \{ \} \]
\[ mon\_c\ fg\ cb \cap (mon\_c\ fg\ ((\#s\#) + ca) \cup mon\_ww\ fg\ (map\ le\_rem\_s\ w1)) = \{ \} \]

\[ by \ fastforce+ \]
\[ from\ I(4)\ obtain\ e\ wb'\ where\ EE: wb=e#wb'\ ee=ENV\ e\ w2'=map\ ENV\ wb'\ by \ (cases\ wb,\ auto) \]
\[ with\ I(6)\ obtain\ cbh\ where\ SPLIT: (cb,e,cbh)\in ntrp\ fg\ (cbh,wb',cb')\in trcl\ (ntrp\ fg)\ by \ (fast\ dest: trcl-unscons) \]
\[ have\ ((s, ca + cb), ee, (s, ca + cbh)) \in ntrp\ fg\ proof \]
\[ from\ gtrE[OF\ SPLIT(1)]\ obtain\ sb\ cbh\ sbh\ cebh\ where\ NTRS: \ cb = \{\#s\#\} + ceb \ cbh = {\#s\#} + cebh\ ((sb, ceb), e, sbh, cbh) \in ntrp\ fg. \]
\[ from\ ntrs-put-context-s[OF\ NTRS(3)],\ of\ \{\#s\#\} + ca\ EE(1)\ I(7)\ have \]
\[ ((sb, \{\#s\#\} + (ca+ceb)), e, sbh, \{\#s\#\} + (ca+ceb)) \in ntrp\ fg\ by \ (auto\ simp\ add: union-ac) \]
\[ from\ gtrp-env[OF\ this]\ NTRS(1, 2)\ EE(2)\ show \ ?thesis\ by \ (simp\ add: union-ac) \]
\[ qed \]
\[ also\ have\ ((s,ca+cbh),w',(s',ca'+cb')\in trcl (ntrp\ fg)\ proof \ (rule\ I(3)) \]
\[ from\ ntrs-mon-s[OF\ SPLIT(1)]\ I(2, 4, 7, 8)\ EE(2)\ show\ 1: mon\_c\ fg\ cbh \]
\[ \cap (mon\_c\ fg\ ((\#s\#) + ca) \cup mon\_ww\ fg\ (map\ le\_rem\_s\ w1)) = \{ \} \]
\[ by \ (cases\ e)\ (simp\ add: cil-ctl-cons-helper, fastforce\ simp\ add: cil-ctl-cons-helper) \]
\[ from\ I(7)\ 1\ EE(1)\ show\ mon\_c\ fg\ ((\#s\#) + ca) \cap (mon\_c\ fg\ cbh \cup mon\_ww\ fg\ wb') = \{ \} \ by \ auto \]
\[ qed\ (auto\ simp\ add: EE(3)\ I(5)\ SPLIT(2)) \]
\[ finally\ show \ ?case . \]
\[ qed \]

\[ with\ A\ B\ show \ ?thesis\ by\ blast \]
\[ qed \]

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

**Theorem (in flowgraph) ntr-interleave: valid fg (ca+cb) \[\exists (ca+cb,w,c)\in trcl (ntrp\ fg) \]
\[ ca' cb' wa wb. \]
\[ c'=ca'+cb' \wedge \]
\[ w\in (wa\otimes ca\ fg\ wb) \wedge \]
\[ mon\_c\ fg\ ca \cap (mon\_c\ fg\ cb \cup mon\_ww\ fg\ wb) = \{ \} \wedge \]
\[ mon\_c\ fg\ cb \cap (mon\_c\ fg\ ca \cup mon\_ww\ fg\ wa) = \{ \} \wedge \]
\[(ca,wa,ca') \in \text{trcl} (ntr \ fg) \land (eb,wb,eb') \in \text{trcl} (ntr \ fg)\]

by \textbf{(blast intro: ntr-split ntr-unsplit)}

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

\textbf{Theorem (in flowgraph) ntrp-interleave:}

\textbf{valid fg} \( \{\#s\#\} + c1 + c2 \) \( \implies \)
\( ((s,c1 + c2),w,(s',c1')) \in \text{trcl} (ntrp \ fg) \leftarrow\rightarrow\)
\( \exists \ w1 \ w2 \ c1' \ c2'.\)
\( w \in w1 \odot_{\alpha n \ fg} (\text{map ENV } w2) \land\)
\( c' = c1' + c2' \land\)
\( ((s,c1),w1,(s',c1')) \in \text{trcl} (ntrp \ fg) \land\)
\( (c2,w2,c2') \in \text{trcl} (ntrp \ fg) \land\)
\( \text{mon-ww fg} (\text{map le-rem-s } w1) \cap\)
\( \text{mon-c fg} c2 = \{\} \land\)
\( \text{mon-ww fg} w2 \cap \text{mon-c fg} (\{\#s\#\} + c1) = \{\})\)

\textbf{apply} \textbf{(intro iffI)}

\textbf{apply} \textbf{(blast intro: ntrp-split)}

\textbf{apply} \textbf{(auto intro!: ntrp-unsplit simp add: valid-unconc)}

\textbf{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.

\textbf{Corollary (in flowgraph) ntrp-unsplit:}

\( (c,w,c) \in \text{trcl} (ntrp \ fg);\)
\( \text{valid fg} \{\#s\#\} + c e \land\)
\( c = c e + c e' \land\)
\( w \in w1 \odot_{\alpha n \ fg} w2 \land\)
\( \text{mon-s fg} s w2 \cap (\text{mon-c fg} ce \cup \text{mon-ww fg} w2) = \{\} \land\)
\( \text{mon-c fg} ce \cap (\text{mon-s fg} s \cup \text{mon-ww fg} w1) = \{\} \land\)
\( (\{\#s\#\},w1,\{\#s1\} + ce 1') \in \text{trcl} (ntrp \ fg) \land\)
\( (ce,w2,ce2') \in \text{trcl} (ntrp \ fg)\)

\textbf{apply} \textbf{(intro iffI)}

\textbf{apply} \textbf{(auto intro!: ntrp-unsplit simp add: valid-unconc)}

\textbf{done}

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.

\textbf{Lemma (in flowgraph) ntr-reverse-split:} \( \forall w \ s' c c'.\)

\( (c,w,\{\#s\#\} + ce) \in \text{trcl} (ntrp \ fg);\)
\( \text{valid fg} \ c w \implies\)
\( \exists s w1 w2 ce1' ce2'.\)
\( c = \{\#s\#\} + ce \land\)
\( ce = ce1' + ce2' \land\)
\( w \in w1 \odot_{\alpha n \ fg} w2 \land\)
\( \text{mon-s fg} s w2 \cap (\text{mon-c fg} ce \cup \text{mon-ww fg} w2) = \{\} \land\)
\( \text{mon-c fg} ce \cap (\text{mon-s fg} s \cup \text{mon-ww fg} w1) = \{\} \land\)
\( (\{\#s\#\},w1,\{\#s'\#\} + ce1') \in \text{trcl} (ntrp \ fg) \land\)
\( (ce,w2,ce2') \in \text{trcl} (ntrp \ fg)\)

— The proof works by induction on the initial configuration. Note that configurations consist of finitely many threads only

84
— 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 \texttt{ntr-split[OF add.prems(1,2)]} obtain \texttt{ce1' ce2'} \texttt{w1 w2} where

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

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

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

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

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

by auto

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

**from SPLIT(1) show ?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\#\))) \texttt{\in \text{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-uncnc)**

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

**IHAPP:**

\texttt{ce=\{(st\#\)} + cet

\texttt{ce2' - (#s\#\) = ce21' + ce22'}

\texttt{w2 \in w21 @\alpha fg w22}

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

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

\texttt{\{(st\#\),w21,\{(st\#\} + ce21'\} \in \text{trcl (ntr fg)}}

\texttt{(cet, w22, ce22') \in \text{trcl (ntr fg) by blast}

— And finally we add the path from s again. This requires some monitor sorting and the associativity of the consistent interleaving operator.

**from cil-assoc2 [of w w1 w2 w22 w21] SPLIT(2)** \texttt{IHAPP(3) obtain w1 where**

**CASSOC:** w\in w21 @\alpha fg w1 \texttt{w\in w1 @\alpha fg w22 by (auto simp add: cil-commute)**

**from CASSOC IHAPP(1,3,4,5)** \texttt{SPLIT(3,4) have COMBINE:** \{(s\#\} + cet, w1, ce1' + ce22'} \texttt{\in \text{trcl (ntr fg) by (rule-tac ntr-uncsplit[OF CASSOC(2)**

**SPLIT(5)] IHAPP(7)) (auto simp add: mon-c-uncnc mon-wu-cil)**

**moreover from CASSOC IHAPP(1,3,4,5)** \texttt{SPLIT(3,4) have mon-s fg st \cap**

\texttt{(mon-c fg \{(s\#\} + cet) \cup \text{mon-ww}\ fg w1) = \{}\ mon-c fg \{(s\#\} + cet) \cap (mon-s fg st \cup \text{mon-wu}\ fg w21) = \{}

**by (auto simp add: mon-c-uncnc mon-wu-cil)**

**moreover from right IHAPP(1,2) have \{(s\#\} + cet = \{(s\#\} + cet)
\[ ce' = (ce1' + ce2') \] 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:

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

For some parameter \( k \) and a flowgraph with nodes \( N \) and edges \( E \), this
generates a constraint system over the variables \( \{S^k[v] \mid v \in N\} \). One
constraint is generated for each call edge. While we use a powerset lattice
here, we can in general use any complete lattice. However, all the constraint
systems needed for our conflict analysis are defined over powerset lattices
\((\mathcal{P}(\alpha), \subseteq)\) for some type \( \alpha \). 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, Call q, v) \in \text{edges } fg; (u, M, P) \in S-cs fg k; \\
\quad (\text{return } fg q, M, P) \in S-cs fg k; P' \leq \# P + Ps; \text{ size } P' \leq k \]
\implies (v, mon fg q \cup M \cup Ms, P') \in S-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 \((u, M, P) \in S-cs fg k\) means that there is a same-level path from the entry node of the procedure of \(u\) to \(u\), that uses the monitors \(M\) and spawns at least the threads in \(P\).

\textbf{inductive-set}
\[ S-cs :: (\text{'n, 'p, 'ba, 'm, 'more}) \text{ flowgraph-rec-scheme} \Rightarrow \text{nat} \Rightarrow (\text{'n \times 'm set} \times 'p multiset) \text{ set} \]
\textit{for } fg k

\textit{where}

\[ S-init: (\text{entry } fg p, \{} \{\}\} \in S-cs fg k \]
\[ S-base: [(u, Base a, v) \in \text{edges } fg; (u, M, P) \in S-cs fg k] \implies (v, M, P) \in S-cs fg k \]
\[ S-call: [(u, Call q, v) \in \text{edges } fg; (u, M, P) \in S-cs fg k; \\
\quad (\text{return } fg q, M, P) \in S-cs fg k; P' \leq P + Ps; \text{ size } P' \leq k \]
\[\implies (v, mon fg q \cup M \cup Ms, P') \in S-cs fg k \]
\[ S-spawn: [(u, Spawn q, v) \in \text{edges } fg; (u, M, P) \in S-cs fg k; \\
\quad P' \leq \# q + P; \text{ size } P' \leq k] \]
\[\implies (v, M, P') \in S-cs fg k \]

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

9.1.2 Soundness and Precision

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

As usual the soundness proof works by induction over the length of execution paths. The base case (empty path) trivially follows from the $S$-init constraint. In the inductive case, we consider the edge that induces the last step of the path; for a return step, this is the corresponding call edge (cf. Lemma \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.

\textbf{lemma} \textbf{(}in \textit{flowgraph}) \textbf{S-sound}: \( \forall p v c' P. \)
\[
[[([\text{entry}\ fg\ p],[\#]),w,([v],c'))\in\text{trcl}\ (\text{trss}\ fg); \]
\[
\text{size}\ P\leq k; (\lambda p. [\text{entry}\ fg\ p]) \ 'P\leq c']
\]
\[
\Rightarrow (v,\text{mon-w}\ fg\ w,P)\in S\text{-cs}\ fg\ k
\]
\textbf{proof (induct} w \textbf{rule: length-compl-rev-induct)}
\begin{itemize}
  \item \textbf{case Nil} thus \?case by \textbf{(auto intro: S-init)}
  \item \textbf{next}
  \item \textbf{case} \textbf{(}snoc\ w\ c\text{)} then obtain \textbf{sh}\ ch \textbf{where} \textbf{SPLIT}: \((([\text{entry}\ fg\ p],[\#]),w,(\text{sh},\text{ch}))\in\text{trcl}\ (\text{trss}\ fg)\ ((\text{sh},ch),c,([v],c'))\in\text{trss}\ fg\ by\ (\text{fast dest: trcl-rev-uncons})\)
  \item \textbf{from SPLIT'(2) show} \?case \textbf{proof (cases rule: trss.cases)}
    \item \textbf{case} \textbf{trss-base} then obtain \textbf{u a where} \textbf{CASE}: \( e=L\text{Base}\ a \text{ sh}=[u] \text{ ch}=c' \)
      \( (u,\text{Base}\ a,v)\in\text{edges}\ fg\ by\ \textbf{auto} \)
      \item \textbf{with} \textbf{snoc.hyps[of\ w\ p\ u\ c', \text{OF} - - \text{snoc.prems}(2,3)]} \textbf{SPLIT(1)} \textbf{have} \((u,\text{mon-w}\ fg\ w,P)\in S\text{-cs}\ fg\ k\) \textbf{by} \textbf{blast}
      \item \textbf{moreover from} \textbf{CASE(1)} \textbf{have} \textbf{mon-e}\ \textbf{fg}\ \textbf{e} = \textbf{\{\}} \textbf{by} \textbf{simp}
      \item \textbf{ultimately show} \?\textbf{thesis using} \textbf{S-base[of\ \text{OF}\ CASE(4)]} \textbf{by} \textbf{(auto simp add: mon-w-uncore)}
    \item \textbf{next}
    \item \textbf{case} \textbf{trss-ret} then obtain \textbf{q} \textbf{where} \textbf{CASE}: \( e=L\text{Ret}\ \text{sh}=\text{return}\ \text{fg}\ q\#[v] \text{ ch}=c' \)
      \( \text{by}\ \textbf{auto} \)
      \item \textbf{with} \textbf{SPLIT(1)} \textbf{have} \((([\text{entry}\ fg\ p],[\#]),w,\text{[return}\ \text{fg}\ q,v],c'))\in\text{trcl}\ (\text{trss}\ fg)\ by\ \textbf{simp}
    \item \textbf{from} \textbf{trss-find-call'(OF\ this)} \textbf{obtain} \textbf{at}\ ct\ w1\ w2 \textbf{where} \textbf{FC}:
      \( w=w1@L\text{Call}\ q\#w2 \)
      \item \textbf{from} \textbf{trss-drop-all-context[OF\ FC(5)]} \textbf{obtain} \textbf{esp' where} \textbf{SLP}: \( c'=ct+csp' \)
\end{itemize}

88
from FC(1) have LEN: length w1 \leq length w length w2 \leq length w by auto
from mset-map-split-org-le SLP(1) snoc.prem(3) obtain P1 P2 where
PSPLIT: P=P1+P2 (\lambda p. [entry fg p]) ' # P1 \leq cl (\lambda p. [entry fg p]) ' # P2
\leq \text{exp}' by blast
with snoc.prem(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: (u, mon-w fg w1, P1)
\in S-cs fg k (\text{return} fg q, mon-w fg w2, P2) \in S-cs fg k .
from S-call[OF FC(4)] IHAPP mset-le-cq-refl[OF PSPLIT(1)] snoc.prem(2)]
FC(1) CASE(1) show (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=LSpawn q sh=[u] c'=[\#[entry fg q]#]+ch (u,Spawn q,v)\in edges fg by auto
from mset-map-split-org-le CASE(3) snoc.prem(3) obtain P1 P2 where
PSPLIT: P=P1+P2 (\lambda p. [entry fg p]) ' # P1 \leq \{\#[entry fg q]#\} (\lambda p. [entry fg p]) ' # P2 \leq ch by blast
with snoc.prem(2) have PSIZE: size P2 \leq k by simp
from snoc.hyps[OF - - PSIZE PSPLIT(3)] SPLIT(1) CASE(2) have IHAPP:
(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.prem(2)] CASE(1) show
(v, mon-w fg (w @ [c]), P) \in S-cs fg k by (auto simp add: mon-w-unconc)

 qed

Precision means that all entries appearing in the smallest solution of the constraint system are justified by some path in the operational characterization. For proving precision, one usually shows that a family of sets derived as an abstraction from the operational characterization solves all constraints.
In our formalization of constraint systems as inductive sets this amounts to constructing for each constraint a justifying path for the entries described on the conclusion side of the implication – under the assumption that corresponding paths exists for the entries mentioned in the antecedent.

lemma (in flowgraph) S-precise: (v,M,P)\in S-cs fg k
\Rightarrow \exists p c' w.
((\{entry fg p\},\#),w,(\{v\},c')]\in trcl (trss fg) \land
size P\leq k \land
(\lambda p. [entry fg p]) ' # P \leq c' \land
M=mon-w fg w

proof (induct rule: S-cs.induct)
case (S-init p) have ((\{entry fg p\},\#),\],\{entry fg p\},\#))\in trcl (trss fg) by simp-all
thus ?case by fastforce
next case (S-base u a v M P) then obtain p c' w where IHAPP: \(((\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 by blast}
note IHAPP(1)
also from S-base have \(((\omega), \text{LBase a,} ([v], c'))\in \text{trss fg by (auto intro: trss-base)}
finally have \(((\text{entry fg } p), \{\#\}), w \circ \text{LBase a,} [v], c') \in \text{trcl (trss fg)} .
moreover from IHAPP(1) have M=mon-w fg (w \circ \text{LBase a) by (simp add: mon-w-unconc)}
ultimately show \?case using IHAPP(2,3,4) by blast
next case (S-call u q v M P ms P s) then obtain p csp1 w1 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 by blast}
from S-call obtain csp2 w2 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}
\quad by (blast dest: trss-er-path-proc-const)
from trss-c-no-mon[OF REACHING-PATH(1)] trss-c-no-mon[OF SL-PATH(1)] have NOMON: mon-c fg csp1 = {} mon-c fg csp2 = {} by auto
have \(((\text{entry fg } p), \{\#\}), w1@LCall q#w2@LRef,([v],csp1+csp2))\in \text{trcl (trss fg)} proof —
\quad note REACHING-PATH(1)
\quad also from trss-call[OF S-call(1)] NOMON have \(((\omega), \text{csp1}), LCall q,([\text{entry fg } q,v],csp1))\in \text{trss fg by (auto)}
\quad also from trss-add-context[OF trss-stack-comp[OF SL-PATH(1)]]] NOMON have \(((\text{entry fg } q,v),csp1),w2,([\text{return fg q,v},csp1+csp2])\in \text{trcl (trss fg)} by (simp add: union-ac)
\quad also have \(((\text{return fg q,v},csp1+csp2),LRef,([v],csp1+csp2))\in \text{trss fg by (rule trss-ret)}
\quad finally show \?thesis by simp
qed
moreover from REACHING-PATH(4) SL-PATH(4) have mon fg q \cup M \cup Ms = mon-w fg (w1@LCall q#w2@LRef) by (auto simp add: mon-w-unconc)
moreover have (\lambda p. [\text{entry fg } p]) \quad \# (P') \leq csp1+csp2 (is \?f \# P' \leq -) proof —
\quad from mset-map-le[OF S-call(6)] have \?f \# P' \leq \?f \# P + \?f \# Ps by (auto simp add: mset-map-union)
\quad also from mset-le-mono-add[OF REACHING-PATH(3) SL-PATH(3)] have \ldots \leq csp1+csp2 .
\quad finally show \?thesis .
qed
moreover note S-call(7)
ultimately show \?case by blast
next case (S-spawn u q v M P P') then obtain p c' w where IHAPP:
\(((\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 by blast}
note IHAPP(1)
also from S-spawn(1) have \(((u,c')\), \text{LSpawn } q, ([v], \#[entry fg q]# + c') \in \text{trss fg} by (rule trss-spawn)

finally have \(((\text{entry fg p}, [\#]), w @ \text{LSpawn q}, [v], \#[entry fg q]# + c') \in \text{trel (trss fg)}\).

moreover from IHAPP(4) have M=mon-w fg \((w @ \text{LSpawn q})\) by (simp add: mon-w-anconc)

moreover have \((\lambda p. \text{entry fg p}) \ 'P' \leq \#[entry fg q]# + c' (is \ ?f \ '# - \leq -) proof –

from mset-map-le[OF S-spawn(4)] have \ ?f 'P' \leq \#[entry fg q]# + \?f '# P by (auto simp add: mset-map-union)

also from mset-le-mono-add[OF - IHAPP(3)] have \ldots \leq \#[entry fg q]# + c' by (auto intro: IHAPP(3))

finally show ?thesis .

qed

case by blast

mu

Finally we can state the soundness and precision as a single theorem
theorem (in flowgraph) S-sound-precise:
\((v,M,P) \in S\)-cs fg k \iff
\((\exists p c' w. \(((\text{entry fg p}), [\#]), w, ([v], c') \in \text{trcl (trss fg)} \land
\quad \text{size P} \leq k \land (\lambda p. \text{entry fg p}) \ 'P' \leq c' \land M=mon-w fg w)

using S-sound S-precise by blast

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

lemma (in flowgraph) S-sound-ntrp:
assumes A: \(((u,\#), \text{eel}, (sh, ch)) \in \text{ntrp fg and}
CASE: \!
| eel=LOC (LCall p\# w);
| (u, Call p, u) \in \text{edges fg};
| sh=\[v,u\];
| \text{proc-of fg v = p;}
| \text{mon-c fg ch = \{}\};
| \!
s. s:\# ch \Rightarrow \exists p u v. s=\text{[entry fg p]} \land
| \quad (u, \text{Spawn p, v}) \in \text{edges fg} \land
| \quad \text{initialproc fg p;}
| \!
| \quad \!
s. s:\# ch \Rightarrow \exists p u v. s=\text{[entry fg p]} \land
| \quad (u, \text{Spawn p, v}) \in \text{edges fg} \land
| \quad \text{initialproc fg p by (auto intro: ntrp-c-cases-s[OF EE(2)])}

| \Rightarrow Q

\]

shows Q

proof –

from A obtain ee where EE: eel=LOC ee \(((u,\#), \text{eel}, (sh, ch)) \in \text{ntrps fg} by
(auto elim: gtrp.cases)

have CHFMT: \!
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 by (auto intro: ntrp-c-cases-s[OF EE(2)])}

91
with \(c\)-of-initial-no-mon have CHNOMON: \(\text{mon-c fg ch} = \{\}\) by blast

from EE(2) obtain \(p u' v w\) where FIRSTSPLIT: \(\exists = \text{LCall p\# w }(([u],\{\#\}), \text{LCall p,([entry fg p],[u],\{\#\}))\in trcl \text{ trss fg sh}=[v,u'][([\text{entry fg p}],[\#]), w,([v],ch)]\in trcl \text{ trss fg})\text{ by (auto elim! ntrs.cases[simplified])}\)
from FIRSTSPLIT have EDGE: \((u, \text{LCall p, u'} )\in \text{edges fg by (auto elim! trss.cases)}\)
from trss-bot-proc-const[where \(s=[]}\) and \(s'=\[]\), simplified, OF FIRSTSPLIT(4)]

have \(\text{PROC-OF-V: proc-of fg v = p by simp}\)

have \(!P. (\lambda p. [\text{entry fg p}]\text{ }\# P \leq ch \implies (v,mon-w fg w, P)\in S-cs fg \text{ (size P)}\)

proof –

\[\text{fix P assume (λp. [entry fg p]) }\# P \leq ch\]
from S-sound[OF FIRSTSPLIT(4) - this, of size P] show ?thesis P by simp

ded with EE(1) FIRSTSPLIT(1,3) EDGE PROC-OF-V CHNOMON CHFMT show Q by (rule_tac CASE) auto

ded

lemma (in flowgraph) S-precise-ntrp:
assumes \(\text{ENTRY: } (v,M,P)\in S-cs fg k\) and
\(P: \text{proc-of fg v = p and}
EDGE: (u, \text{Call p, u'} )\in \text{edges fg}\)

shows \(\exists w ch.\)

\(((u),\{\#\}), \text{LOC } (\text{LCall p\# w}), ([v,u'], ch))\in ntrp fg \land\)
size \(P \leq k \land\)
\(M=\text{mon-w fg w} \land\)
\(\text{mon-n fg v = mon fg p} \land\)
\((\lambda p. \text{[entry fg p]}) \# P \leq ch \land\)
\(\text{mon-c fg ch} = \{\}\)

proof –

from P S-precise[OF ENTRY, simplified] trss-bot-proc-const[where \(s=[]\) and \(s'=\[]\), simplified] obtain wsl ch where

SLPATH: \(((\text{[entry fg p]}), \{\#\}), wsl, [v], ch) \in trcl \text{ trss fg} \text{ size } P \leq k \text{ (λp. [entry fg p]}) \# P \leq ch M = \text{mon-w fg w} \text{ wsl by fastforce}\)
from mon-n-same-proc[OF trss-bot-proc-const[where \(s=[]\) and \(s'=\[]\), simplified, OF SLPATH(1)]] have MON-V: \(\text{mon-n fg v = mon fg p} \text{ by (simp)}\)
from trss-cases[OF SLPATH(1), simplified] have CHFMT: \(\lambda s. s : \# ch \implies \exists p. s = [\text{entry fg p}] \land (\exists u v. (u, \text{Spawn p, v}) \in \text{edges fg}) \land \text{initialproc fg p by blast}\)

with c-of-initial-no-mon have CHNOMON: \(\text{mon-c fg ch} = \{\}\) by blast
— From the constraints prerequisites, we can construct the first step

have FS: \(((u), \{\#\}), \text{LCall p\# wsl}, ([v,u'], ch))\in ntrp fg \text{ proof (rule ntrs-step[where } r=[], \text{ simplified})}\)

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

qed (rule SLPATH(1))

hence FSP: \(((u), \{\#\}), \text{LOC } (\text{LCall p\# wsl}), ([v,u'], ch))\in ntrp fg \text{ by (blast intro: gtrp-loc)}\)

from FSP SLPATH(2,3,4) CHNOMON MON-V show ?thesis by blast

ded

92
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 \((u, M_l, M_e, h) \in RU-cs \) 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 RU-spawn-constraint describes the case that \( U \) is reached from one of the spawned threads. In the two latter cases, we have to check whether prepending the macrostep to the reaching path is allowed or not due to monitor restrictions. In the call case, the procedure of the initial node must not own monitors that are used in the environment steps of the appended reaching path \( \text{mon-n fg u} \cap \text{Me} = \{\} \). As we only test disjointness with the set of monitors used by the environment, reentrant monitors can be handled. In the spawn case, we have to check disjointness with both, the monitors of local and environment steps of the reaching path from the spawned thread, because from the perspective of the initial thread, all these steps are environment steps \((\text{mon-n fg u} \cup \text{mon fg p}) \cap (\text{Ml} \cup \text{Me})=\{\})\). Note that in the call case, we do not need to explicitly check that the monitors used by the environment are disjoint from the monitors acquired by the called procedure because this already follows from the existence of a reaching path, as the starting point of this path already holds all these monitors.

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

9.2.2 Soundness and precision

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

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

**lemma (in flowgraph) ntr-reverse-split-atU:**

assumes \( V: \text{valid fg c} \) and
- \( A: \text{atU U c'}\) and
- \( B: (c, w, c') \in \text{trcl (ntr fg)}\)

shows \( \exists s \ w' \ c'\).
- \( s: #c \land w' \sqsubseteq w \land c1' \leq c' \land \)
- \( \text{atU U c1'} \land ((\#s\#, w1, c1') \in \text{trcl (ntr fg)})\)

**proof** –

**obtain ui r ce' where C’FMT: c'={#ui#r#}+ce' ui\in U by (rule atU-fmt[OF A], simp only: mset-contains-eq) (blast dest: sym)**

**with ntr-reverse-split[OF - V] B obtain s ce w1 w2 ce1' ce2' where R_SPLIT:**

\( c=\{#s\#\}+ce ce'='ce1'+ce2' w\in w1 \otimes \alpha_n fg w2 (\{#s\#\}, w1, \{#ui#r#\} + ce1') \in \text{trcl (ntr fg)}\) by blast

**with C’FMT have s: #c \sqsubseteq w \{#ui#r#\}+ce1' \leq c' \text{ atU U (\{#ui#r#\}+ce1')}**

by (auto dest: cil-ileq)

**with R_SPLIT(4) show ?thesis by blast**

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

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

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

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

**lemma** (in flowgraph) RU-sound:

$$\begin{align*}
!!u s' c'. \left[ (((u, \#), w, (s', c')) \in \text{trcl} (\text{ntrp \ fg}); \ atU U (\# s'\# + c') \right]
\implies \exists Ml Me h. \left[ (((u, \#), w, (s', c')) \in \text{trcl} (\text{ntrp \ fg}); \ atU U (\# s'\# + c') \right]
\end{align*}$$

— For reaching paths of length zero, the proposition follows immediately by the constraint RU-init

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

— For a reaching path of length zero, the proposition follows immediately by the constraint RU-init

**next**

— For a non-empty path, we regard the first step and the rest of the path

**then obtain** sh ch where SPLIT:

$$\begin{align*}
(((u, \#), w, (s', c')) \in \text{ntrp \ fg}\\
((sh, ch), wwl, (s', c')) \in \text{trcl} (\text{ntrp \ fg})
\end{align*}$$

by (fast dest: trcl-uncors)

**obtain** p $u' v$ w where

— The first step consists of an initial call and a same-level path

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

**proc-of fg v = p mon-c fg ch = {}**

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

**and CHFMT:** $\land s, s': ch \implies \exists p u v. s = \text{entry \ fg \ p} \land (u, \text{Spawn p, v}) \in \text{edges} \ fg \wedge \text{initialproc \ fg \ p}$

— For the same-level path, we find a corresponding entry in the $S$-cs-constraint
system

and S-ENTRY-PAT: \( \forall P. (\lambda p. [entry fg p]) \downarrow P \leq ch \implies (v, mon-w fg w, P) \in S-cs fg \) (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
LESPLIT:
\[ wwl \in w1 \circ_{\alpha\lambda} fg \] map ENV w2
\[ c' = c1' + c2' \]
\[ ((sh, \{\#\}), w1, s', c1') \in \text{trcl} (\text{ntrp} fg) \]
\[ (ch, w2, c2') \in \text{trcl} (\text{ntrp} fg) \]
\[ \text{mon-ww} fg (\text{map} \text{le-rem-s} w1) \cap \text{mon-c} fg ch = \{\} \]
\[ \text{mon-ww} fg w2 \cap \text{mon-s} fg sh = \{\} \]
by blast
— We make a case distinction whether \( U \) was reached from the local thread or from the spawned threads

from Cons.prems(2) LESPLIT(2) have atU U \((\{\#s'\#\} + c1') + c2'\) by (auto simp add: union-ac)
thus ?case proof (cases rule: atU-union-cases)
case left — \( U \) was reached from the local thread
from cil-ileq[OF LESPLIT(1)] have ILEQ: \( w1 \leq wwl \) and LEN: length \( w1 \leq \) length \( wwl \) by (auto simp add: le-list-length)
— We can cut off the bottom stack symbol from the reaching path (as always possible for normalized paths)

from FS-FMT(3) LESPLIT(3) ntrp-stack-decomp[of v] \( \{\#\} \) \( w1 s' c1' fg, \) simplified obtain \( v' rr \) where DECOMP: \( s' = v'\#rr@u' \) \((\{v\}, \{\#\}), w1, (v'\#rr, c1')\) \in \text{trcl} (\text{ntrp} fg) by auto
— This does not affect the configuration being at \( U \)
from atU-xchange-stack left DECOMP(1) have ATU: \( \text{atU} U (\{\#v'\#rr\} + c1') \)
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: \( (v, Ml, Me, h) \in \text{RU-cs} fg U \) \( Ml \subseteq \text{mon-loc} fg w1 Me \subseteq \text{mon-env} fg w1 h \leq \alpha\lambda (\text{map} \text{of} \text{fg} w1) \) by blast
— Next, we have to apply the constraint RU-call

from S-ENTRY-PAT[of \{\#\}, simplified] have S-ENTRY: \( (v, \text{mon-w} fg w, \{\#\}) \in S-cs \) \( fg 0 \) .
have MON-U-ME: \( \text{mon-n} fg u \cap Me = \{\} \) proof —
from ntrp-mon-env-a-no-ctx[OF Cons.prems(1)] have mon-env fg \( wwl \cap \text{mon-n} fg u = \{\} \) 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 \( v, \)
mon fg p ∪ mon-w fg w ∪ Ml, Me, ah-update h (mon fg p, mon-w fg w) (Ml ∪ Me) ∈ RU-cs fg U.

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

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

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

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

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

from IHAPP(ד) have h ≤ αah (map (anl fg) w1).

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

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

next

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

next

from IHAPP(2,ג) have (Ml ∪ 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 (Ml ∪ Me) ⊆ mon-pl (map (anl fg) wwl).

qed

qed

ultimately show ∗thesis by blast

next

case right — U was reached from the spawned threads

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

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

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

from ntr-reverse-split-atU[OF CHVALID(ד) right LESPLIT(ג)] obtain q wr cr' where RI: {[entry fg q] = ch wr≤w2 cr'≤c' atU U cr' (♯entry fg q♯)} wr,cr' ∈ trcl (ntr fg) by (blast dest: CHFM T)

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

from ntrs.gtr2gt6[rwhere c=♯, simplified, OF RI(ג)] obtain sr' cre' wwr

where RI-NTRP: cre' =♯sr♯ + cre' wwr = map le-rem-s wwr ([{entry fg q},♯]), wwr,(sr',cre')) ∈ trcl (ntrp fg) by blast

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

— The induction hypothesis yields a constraint system entry

from Cons.hyps[OF LEN’ RI-NTRP(ד)] RI-NTRP(ג) RI(ד) 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 ≤ αah (map (anl fg) wwr) by auto

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

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

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

have $MON$-MLE-ENV: $Ml \cup Me \subseteq \text{mon-env fg wwl}$ proof —
from $IHAPP(2,3)$ have $Ml \cup Me \subseteq \text{mon-loc fg wwr} \cup \text{mon-env fg wwr}$ by auto

also from $\text{mon-ww-of-le-rem}$[symmetric] $RI$-NTRP(2) have $\ldots = \text{mon-ww fg wr}$ by fastforce

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

finally show $\vdash \text{thesis}$ by auto

qed

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

from $RU$-spawn[OF $FS$-FMT(2,4)] $S$-ENTRY - $IHAPP(1)$ $MON$-MLE

have $(u, \text{mon fg p }\cup \text{mon-w fg w}, Ml \cup Me, \text{ah-update h } (\text{mon fg p, mon-w fg w}) (Ml \cup Me)) \in RU$-cs fg $U$ by simp

— Next we have to assemble the rest of the proposition

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

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

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

proof (simp add: $\text{ah-update-cons}$)

have $MAP$-HELPER: $\text{map (anl fg) wwr} \leq \text{map (anl fg) wwl}$ proof —
from $RI$-NTRP(2) have $\text{map (anl fg) wwr} = \text{map (anl fg) wr}$ by (simp add: $\text{anl-anl}$)

also from $\text{le-list-map}$[OF $RI(2)$] have $\ldots \leq \text{map (an fg) w2}$.

also have $\ldots = \text{map (anl fg) (map ENV w2)}$ by simp

also from $\text{le-list-map}$[OF ILEQ] have $\ldots \leq \text{map (anl fg) wwl}$.

finally show $\vdash \text{thesis}$.

qed

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

from $IHAPP(4)$ have $h \leq \alpha_{ah} (\text{map (anl fg) wwr})$.

also have $\ldots \leq \alpha_{ah} (\text{map (anl fg) wwl})$ by (rule $\alpha_{ah-ileq}$[OF $MAP$-HELPER])

finally show $h \leq \alpha_{ah} (\text{map (anl fg) wwl})$.

next

from $FS$-FMT(1) show $(\text{mon fg p, mon-w fg w}) = \alpha_{anl fg eel}$ by simp
∃ blast fastforce by (Ur = h)

next
from IHAPP(2,3) mon-pl-ileq[OF MAP-HELPER] show Ml ∪ Mel ⊆ mon-pl (map (and fg) wsl) by (auto simp add: mon-pl-of-casl)
qed
qed
ultimately show ?thesis by blast
qed
qed

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

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

**Lemma (in flowgraph)** RU-precise: (u,Ml,Me,h) ∈ RU-cs fg U

\[ \Rightarrow \exists w s' c'. (([u],\#),w,(s',c')) \in trcl (ntrp fg) \land atU U ((\#s'\#)+c') \land mon-loc fg w = Ml \land mon-env fg w = Me \land \alpha h (map (\alpha nl fg) w) = h \]

**Proof** (induct rule: RU-cs.induct)

— The RU-init constraint is trivially covered by the empty path

**Case (RU-init u)** thus ?case by (auto intro: exI[of _ []])

**Next**

— Call constraint

**Case (RU-call p u u' v P Ml Me h)**

then obtain w s' c' where IHAPP: \(((v), \#), w, s', c') \in trcl (ntrp fg) atU U ((\#s'\#)+c') mon-loc fg w = Ml mon-env fg w = Me \alpha ah (map (\alpha nl fg) w) = h by blast

from RU-call.hyps(2) S-precise[OF RU-call.hyps(3), simplified] trss-bot-proc-cons\[\text{where} s=[] \text{and} s'=[], \text{simplified}\] obtain wsl ch where

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

by fastforce

from trss-c-cases[OF SLPATH(1), simplified] have CHFMT: \(\wedge s. s :\# ch \Rightarrow \exists p. s = [\text{entry fg p}] \land (\exists u v. (u, \text{Spawn p, v}) \in edges fg) \land \text{initialproc fg p}\) by blast

with c-of-initial-no-mon have CHNOMON: mon-c fg ch = {} by blast

— From the constraints prerequisites, we can construct the first step

have FS: \(((u),\#), LCall p#wsl,(v,u',ch)) \in ntrps fg\] proof (rule ntrp-step\[\text{where} r=[]\], simplified)]

from RU-call.hyps(1) show \(((u),\#), LCall p, [\text{entry fg p, u'}], \#) \in trss fg\] by (auto intro: trss-call)
qed (rule SLPATH(1))

hence FSP: \(((\{u\}, \{\#\}), \text{LOC} ((\text{LCall } p \# \text{wsl}) , ( [v, u'], ch)) \in \text{ntrp } fg)\) by (blast intro: gtrp-loc)

also

\(\vdash\) The rest of the path comes from the induction hypothesis, after adding the rest of the threads to the context

have \(((\{v, u', ch\}, w, s' \in \{u', c' + ch\}) \in \text{trcl (ntrp } fg)\) proof (rule ntrp-add-context[OF ntrp-stack-comp[ 1 OF IHAPP](1), \text{where } r=[u'], \text{where } cn=ch, simplified])

from RU-call.hyps(1, 6) IHAPP(4) show mon-n fg u' \cap \text{mon-env } fg \ w = \{}

by (auto simp add: mon-n-def edges-part)

from CHNOMON show mon-ww fg (map le-rem-s w) \ \cap \ \text{mon-c } fg \ ch \ = \ {} \ by auto

qed

finally have \(((\{u\}, \{\#\}), \text{LOC} ((\text{LCall } p \# \text{wsl}) \# w, s' \in \{u', c' + ch\}) \in \text{trcl (ntrp } fg)\) .

\(\vdash\) 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{atU} U (\{\# s'=\{u\} \#\} + (c'+ch)) by auto

moreover have mon-loc fg (LOC (\text{LCall } p \# \text{wsl}) \# w) = mon fg p \cup M \cup Ml

using SLPATH(2) IHAPP(3) by auto

moreover have mon-env fg (LOC (\text{LCall } p \# \text{wsl}) \# w) = Ml using IHAPP(4)

by auto

moreover have \text{oah} (\text{map (anl } fg) (\text{LOC (LCall } p \# \text{wsl}) \# w)) = ah-update h (mon fg p, M) (Ml \cup Ml) proof -

\begin{itemize}
  \item have \text{oah} (\text{map (anl } fg) (\text{LOC (LCall } p \# \text{wsl}) \# w)) = ah-update (\text{cah} (\text{map (anl } fg) w)) (mon fg p, mon-w fg wsl) (\text{mon-pl (map (anl } fg) w}) by (auto simp add: ah-update-cons)
\end{itemize}

also have \(\vdash\) \text{ah-update} h (mon fg p, M) (\text{Ml } \cup \text{Ml}) proof -

from IHAPP(5) have \text{oah} (\text{map (anl } fg) w) = h .

moreover from SLPATH(2) have \text{mon-pl (map (anl } fg) w}) = (\text{mon fg p}, M) by (simp add: mon-pl-of-anl)

moreover from IHAPP(3, 4) have \text{mon-pl (map (anl } fg) w}) = \text{Ml } \cup \text{Ml}

by (auto simp add: mon-pl-of-anl)

ultimately show \text{thesis by simp}

qed

finally show \text{thesis} .

qed

ultimately show \text{thesis by blast}

next

\begin{itemize}
  \item Spawn constraint
\end{itemize}

case (RU-spawn u p u' v M P q Me h) then obtain w s' c' where IHAPP:

\begin{itemize}
  \item have (\text{entry fg } q) , l, s, s' \in \text{trcl (ntrp } fg) \ \text{atU} U (\{\#s\#\} + c') \text{mon-loc fg w}
  \item = Ml \text{mon-env } fg \ w = Ml \text{ah } (\text{map (anl } fg) \ w) = h \ \text{by blast}
\end{itemize}

from RU-spawn.hyps(2) S-precise[OF RU-spawn.hyps(3), simplified] trss-bot-prec-const[where s='' and s''=''\], simplified obtain wsl ch where

SLPATH: \(((\text{entry fg } p), \{\#\}), \text{wsl}, [v], ch) \in \text{trcl (trss } fg) \ M = \text{mon-w fg wsl size P } \leq 1 \ \text{\lambda p. (entry fg } p) \c' \ # P \ \leq \ ch \ \text{by fastforce}

with RU-spawn.hyps(4) obtain che where PFMT: P=\{\#q\} \ ch = \{\# [\text{entry fg } q]\#\} + \ c' \ \text{by (auto elim!: mset-size-le1-cases mset-le-addE)
from trss-cases[OF SLPATH(1), simplified] have CHFMT: \( \land s. s : \# c \implies \exists p. s = [entry fg p] \land (\exists u v. (u, Spawn p, v) \in edges fg) \land\ initialproc fg p \text{ by blast} \)

with c-of-initial-no-mon have CHNOMON: mon-c fg ch = {} by blast

have FS: \(((\{u\},\{\#\}),\text{LCall} p \# \text{wsl}, ([v,u'], ch)) \in ntrs fg\) proof (rule ntrs-step[where r=[], simplified])

from RU-spawn.hyps(1) show \(((\{u\},\{\#\}),\text{LCall} p, [entry fg p, u'], \{\#\}) \in trss fg \text{ by (auto intro: trss-call)}\)
qed (rule SLPATH(1))

hence FSP: \(((\{u\},\{\#\}),\text{LOC} (LCall p \# \text{wsl}), ([v,u'], ch)) \in ntrs fg \text{ by (blast intro: gtrp-loc)}\)
also have \(((\{v,u', ch\}), \text{map ENV (map le-rem-s w)}, [v,u'], \text{che} + \{\text{#s'\#} + c'\}) \in trcl (ntrp fg)\) proof –

from IHAPP(3,4) have mon-ww fg (map le-rem-s w) \(\subseteq Ml \cup Me\) by (auto simp add: mon-ww-of-le-rem)

with RU-spawn.hyps(1,2,7) have (mon-n fg v \(\cup\) mon-n fg u') \(\cap\) mon-ww fg (map le-rem-s w) = {} by (auto simp add: mon-n-def edges-part)

with ntr2ntrp[OF gtrp2gtr[OF IHAPP(1)], of \([v,u']\) che] PFMT(2) CHNOMON show ?thesis by (auto simp add: union-ac mon-c-unconc)
qed

finally have \(((\{u\},\{\#\}),\text{LOC} (LCall p \# \text{wsl}) \# \text{map ENV (map le-rem-s w)}, [v,u'], \text{che} + \{\text{#s'\#} + c'\}) \in trcl (ntrp fg)\).

moreover have from IHAPP(2) have atU U \((\#[v,u']\#) + (\text{che} + \{\text{#s'\#} + c'\})\) by simp

moreover have mon-loc fg (LOC (LCall p \# \text{wsl}) \# \text{map ENV (map le-rem-s w)}) = mon fg p \(\cup\) M using SLPATH(2) by (auto simp del: map-map)

moreover have mon-env fg (LOC (LCall p \# \text{wsl}) \# \text{map ENV (map le-rem-s w)}) = Ml \(\cup\) Me using IHAPP(3,4) by (auto simp add: mon-ww-of-le-rem simp del: map-map)

moreover have \(\alpha\alpha h\) (map (\(\alpha n\) fg) (LOC (LCall p \# \text{wsl}) \# map ENV (map le-rem-s w))) = ah-update h (mon fg p, M) (Ml \(\cup\) Me) proof –

have \(\alpha\alpha h\) (map (\(\alpha n\) fg) (LOC (LCall p \# \text{wsl}) \# map ENV (map le-rem-s w))) = ah-update (\(\alpha\alpha h\) (map (\(\alpha n\) fg) (map le-rem-s w))) (mon fg p, mon-w fg wsl)

\(\alpha n\)-pl (map (\(\alpha n\) fg) (map le-rem-s w))) by (simp add: ah-update-cons o-assoc)

also have \(\ldots = ah\text{-update}\) h (mon fg p, M) (Ml \(\cup\) Me) proof –

from IHAPP(5) have \(\alpha\alpha h\) (map (\(\alpha n\) fg) (map le-rem-s w)) = h by (simp add: \(\alpha n\)-cons)

moreover from SLPATH(2) have (mon fg p, mon-w fg wsl) = (mon fg p, M) by simp

moreover from IHAPP(3,4) have mon-pl (map (\(\alpha n\) fg) (map le-rem-s w)) = Ml \(\cup\) Me by (auto simp add: mon-pl-of-\(\alpha n\)-cons)

ultimately show ?thesis by simp
qed

finally show ?thesis .
qed
ultimately show ?case by blast
qed
9.3 Simultaneously reaching path

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

9.3.1 Constraint system

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

\[\text{inductive-set}\]

\[\begin{align*}
\text{RUVC} &::= (\text{\textbf{\textquotesingle \textbf{n},\textbf{\textquotesingle p,\textquotesingle ba,\textquotesingle m,\textquotesingle more}} flowgraph-rec-scheme} \Rightarrow \text{\textbf{\textquotesingle n set} \Rightarrow \text{\textbf{\textquotesingle n set \times \textbf{\textquotesingle m set} set)}} for \text{\textbf{fg U V}} \\
\text{\textbf{\textquotesingle n set}} &\Rightarrow \text{\textbf{\textquotesingle n set \times \textbf{\textquotesingle m set} set}} for \text{\textbf{fg U V}} \\
\text{\textbf{\textquotesingle n set \times \textbf{\textquotesingle m set} set}} &\Rightarrow \text{\textbf{\textquotesingle n set \times \textbf{\textquotesingle m set} set}} for \text{\textbf{fg U V}}
\end{align*}\]

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:

**RUVCALL** Both, \( U \) and \( V \) are reached from the initial thread.

**RUVPICK** Both, \( U \) and \( V \) are reached from a single spawned thread.

**RUVRIGHT** \( U \) is reached from the initial thread, \( V \) is reached from a spawned thread.

**RUVEARLY** \( V \) is reached from the initial thread, \( U \) is reached from a spawned thread.

**RUVEARLY** 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) RUVC-Sound:**

\[
\begin{align*}
& \left( ((u,\#),w,(s',c')) \in \text{trcl}(\text{ntrp} f g): \text{atUV} U V (\{\#s'\#\}+c') \right) \\
\Rightarrow & \exists Ml Me. \\
& (u,Ml,Me)\in \text{RUVCs} f g U V \land \\
& Ml \subseteq \text{mon-loc} f g w \land \\
& Me \subseteq \text{mon-env} f g w
\end{align*}
\]

— 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** \( (\text{Cons} ee \, ww) \) then obtain \( sh \, ch \) where SPLIT: \( ((u,\#),ee,(sh,ch))\in \text{ntrp} f g ((sh,ch),ww,(s',c'))\in \text{trcl}(\text{ntrp} f g) \) by \( \text{fast dest: trcl-uncs} \)

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

**LESPLIT:** \( ww \in w1 \otimes_{\text{ord} f g} \text{map} \, \text{ENV} \, w2 \, c' = c1' + c2' ((sh, \{\#\}), w1, s', c1') \in \text{trcl}(\text{ntrp} f g) \) \hfill (ch, w2, c2') \in \text{trcl}(\text{ntrp} f g) \text{ mon-ww f g (map le-rem-s w1) } \cap \\
\text{ mon-c f g ch } = \{\} \text{ mon-ww f g w2 } \cap \text{ mon-s f g sh } = \{\}

by blast

**Obtain** \( p \, u' \, v \, w \) where

**FS-FMT:** \( ee = \text{LOC} (\text{LCall} p \ # \ w) \) \( (u, \text{Call} p, u') \in \text{edges} f g \) \( sh = [v, u'] \)

**proc-of** \( f g \, v = p \) \text{ mon-c f g ch } = \{\}

103
and CHFMT: $\forall s, s \neq \emptyset$ ch $\implies$ $\exists p u v. s = [\text{entry } fg p] \land (u, \text{Spawn } p, v) \in edges fg p \land \text{initialproc } fg p$

and S-ENTRY-PAT: $\forall (p, [\text{entry } fg p]) \# P \leq ch = \implies (v, \text{mon-w } fg w, P) \in S$-cs fg (size $P$)

by (rule S-sound-ntrp[OF SPLIT(1)]) blast

from ntrp-mon-env-u-no-ctz[OF SPLIT(2)] $\text{FS-FMT}(3,4)$ edges-part[OF $\text{FS-FMT}(2)$]

have MON-PU: $\text{mon-env } fg w w \cap (\text{mon } fg p \cup \text{mon-n } fg w) = \{}$ by (auto simp add: mon-n-def)

from cil-ileq[OF $\text{LESPLIT}(1)$] $\text{mon-loc-ileq}[of w1 \text{ w} \text{fg} w]$ $\text{mon-ileq}[of w1 \text{ w} \text{fg}]$

have MON1-LEQ: $\text{mon-loc } fg w1 \subseteq \text{mon-loc } fg w2 \text{ w1} \subseteq \text{mon-loc } fg \text{ w2}$ by auto

from cil-ileq[OF $\text{LESPLIT}(1)$] $\text{mon-ileq}[of \text{map ENV } w2 \text{ w} \text{fg}]$ have MON2-LEQ: $\text{mon-wv } fg w2 \subseteq \text{mon-wv } fg w1$ by simp

from LESPLIT(3) $\text{FS-FMT}(3)$ ntrp-stack-decomp[of $\{\#\} w1 \text{ s' c1}', \text{simplified}$]

obtain $v' r r$ where DECOMP-LOC: $s' = v'##rr@u' ([v],[\#],w1,(v'##rr,c1')) \in \text{trcl}$ (ntrp fg) by (simp, blast)

from Cons.prems(2) $\text{LESPLIT}(2)$ $\text{atUV} U' V' ((\{\# s'\} + c1') + c2')$ by (simp add: union-cc)

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

case left with DECOMP-LOC(1) have ATUV: $\text{atUV} U V ((\{v'##rr\} + c1')$

by simp

from Cons.hyps[OF $\text{DECOMP-LOC}(2)$ $\text{ATUV}$] cil-length[OF $\text{LESPLIT}(1)$]

obtain Ml Me where IHAPP: $(v, \text{ Ml, Me}) \in \text{RUV-cs } fg U V Ml \subseteq \text{mon-loc } fg w1 \text{ Me} \subseteq \text{mon-env } fg w1$ by auto

from RUV-call[OF $\text{FS-FMT}(2,4)$ $\text{S-ENTRY-PAT}$[of $\{\#\}$, simplified] IHAPP(1)]

have $(u, \text{ mon } fg p \cup \text{mon-u } fg w \cup \text{ Ml, Me}) \in \text{RUV-cs } fg U V \text{ using IHAPP}(3)$

MON-PU MON1-LEQ by fastforce

moreover have $\text{mon } fg p \cup \text{mon-w } fg w \cup \text{ Ml} \subseteq \text{mon-loc } fg (\text{ee##ww})$ using

$\text{FS-FMT}(1)$ IHAPP(2) $\text{MON1-LEQ}$ by auto

moreover have $\text{Me} \subseteq \text{mon-env } fg (\text{ee##ww})$ using IHAPP(3) MON1-LEQ by auto

ultimately show ?thesis by blast

next

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

then obtain $s1' s2'$ where R-STACKS: $\{##s1'##\} + \{##s2'##\} \leq c2'$ at U-S $U s1'$ at U-S $V s2'$ by (unfold atUV-def) auto

then obtain $c2'$ where C2'FMT: $c2' = \{##s1'##\} + (\{##s2'##\} + c2')$ by (auto simp add: mset-le-exists-conv union-cc)

obtain $q ceh w21 w22 c21' c22'$ where

$\text{REV_SPLIT}: ch = \{##[\text{entry } fg q]##\} + \text{ceh } \{##s2'##\} + c2' = \text{ce21'} + \text{ce22'} w2 \in w21 \land w22$ mon $fg q \cap (\text{mon-c } fg ceh \cup \text{mon-wv } fg w22) = \{}$ mon-c $fg ceh \cap (\text{mon } fg q \cup \text{mon-wv } fg w21) = \{}$

$(\{##[\text{entry } fg q]##\}, w21, \{##s1'##\} + c21') \in \text{trcl} \ (\text{ntrp } fg) (\text{ceh22}, \text{w22}, \text{c22'}) \in \text{trcl}$ (ntrp $fg$)

proof —

case goa1

from ntr-reverse-split[of ch $w2 s1'$ $\{##s2'##\} + c2' \] ntrp-valid-preserve-s[OF
SPLIT(1), simplified] C2’FMT LESPLIT(4)
  obtain seh ceh w21 w22 ce21’ ce22’
  where
  ch={# seh#} + ceh {#s2’#} + ce2’ = ce21’ + ce22’ w2 ∈ w21 ⊕ αn fg w22 mon-s
  fg seh ∩ (mon-c fg ceh ∪ mon-ww fg w22) = {} mon-c fg ceh ∩ (mon-s fg seh ∪ mon-ww fg w21) = {} ({} seh#, w21, {} s1’#) ∈ trcl (ntr fg) (ceh, w22, ce22’) ∈ trcl (ntr fg)
  by (auto simp add: valid-anconc)
  moreover from this(1) CHFMT[of seh] obtain q where seh=[entry fg q]
  by auto
  ultimately have ch={# [entry fg q]#} + ceh {#s2’#} + ce2’ = ce21’ + ce22’ w2 ∈ w21 ⊕ αn fg w22 mon fg q ∩ (mon-c fg ceh ∪ mon-ww fg w22) = {} mon-c fg ceh ∩ (mon fg q ∪ mon-ww fg w21) = {}
  ({} seh#, w21, {} s1’#) ∈ trcl (ntr fg) (ceh, w22, ce22’) ∈ trcl (ntr fg) by auto
  thus ?thesis using goal1 by (blast)
  qed

  — For applying the induction hypothesis, it will be handy to have the reaching
  path in loc/env format:
  from ntrfg.trgtr2grtp[where c={#}, simplified, OF REV SPLIT(6)] obtain sq’
  csp-q ww21 where
  R-CONV: {#s1’#} + ce21’ = {#sq’#} + csp-q w21 = map le-rem-s ww21
  (((entry fg q), {#}), ww21, sq’, csp-q) ∈ trcl (ntrp fg) by blast
  from cil-ileq[OF REV SPLIT(3)] mon-ww-ileq[of w21 w2 fg] mon-ww-ileq[of w22 w2 fg] have MON2N-LEQ: mon-ww fg w21 ⊆ mon-ww fg w22 mon-ww fg w22 ⊆ mon-ww fg w2 by auto
  from REV SPLIT(2) show ?thesis proof (cases rule: mset-unplsm-dist-cases[case-names\ left’ right’])
  case left’ — Both nodes are reached from the same thread
  have ATUV: atUV U V {{#sq’#} + csp-q} using right C2’FMT R-STACKS(2,3)
  by (subst R-CONV(1)[symmetric], subst left’(1)) simp
  from Cons.hyps[OF - R-CONV(1) ATUV] cil-length[OF REV SPLIT(3)]
 cil-length[OF LESPLIT(1)] R-CONV(2) obtain Mi Me where IHAPP: (entry fg q, Mi, Me) ∈ RUV-cs fg U V Mi ⊆ mon-loc fg ww21 Me ⊆ mon-env fg ww21
  by auto
  from REV SPLIT(1) S-ENTRY-PAT[of {#sq’#}, simplified] have S-ENTRY:
  (v, mon-ww fg w, {#q#}) ∈ S-cs fg 1 by simp
  have MON-COND: (mon-n fg u ∪ mon fg p) ∩ (Mi ∪ Me) = {} proof
  — from R-CONV(2) have mon-ww fg w21 = mon-loc fg ww21 ∪ mon-env fg
  ww21 by (simp add: mon-ww-of-le-rem)
  with IHAPP(2,3) MON2N-LEQ(1) MON-PU MON2-LEQ show ?thesis
  by blast
  qed
  from RUV-spawn[OF FS-FMT(2) FS-FMT(4) S-ENTRY - IHAPP(1)]
  MON-COND] have (u, mon fg p ∪ mon-w fg w, Mi ∪ Me) ∈ RUV-cs fg U V
  by simp
  moreover have mon fg p ∪ mon-w fg w ⊆ mon-loc fg (ce#ww) using
  FS-FMT(1) by auto
  moreover have Mi ∪ Me ⊆ mon-env fg (ce#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

case right' — The nodes are reached from different threads

from R-STACKS(2,3) have ATUV: \( atU U \{ \{ \# sq' \} \} + csp-q \) \( atU V \) \( ce22' \)
by (−) (subst R-CONV(1)[symmetric], simp, subst right'(1), simp)
— We have to reverse-split the second path again, to extract the second interesting thread

obtain \( q' \) \( w22' \) \( ce22e' \) where REVSPLIT': \( [\text{entry fg q}]:\# \text{ ceh w22' } \leq w22 \) \( ce22' \leq ce22' \) \( atU V \) \( ce22e' \) ((\( \{ \# [\text{entry fg q}][\#] \}, w22', ce22e' \) \( \in \text{trcl (ntr fg)} \)) \textit{proof}

—
case goal1

from ntr-reverse-split-atU[OF - ATUV(2) REVSPLIT(7)] ntrp-valid-preserve-s[OF SPLIT(1), simplified] REVSPLIT(1) obtain \( sq'' w22' ce22e' \) where
\( sq'':\# \text{ ceh w22' } \leq w22 \) \( ce22e' \) \( \leq ce22' \) \( atU V \) \( ce22e' \) ((\( \{ \# sq'' \} \), w22', ce22e') \( \in \text{trcl (ntr fg)} \)) by (auto simp add: valid-unconc)

— moreover from CHFMT[of \( sq'' \)] REVSPLIT(1) this(1) obtain \( q' \) where
\( sq'' = [\text{entry fg q}][\#] \) by auto

ultimately show \( \text{thesis using goal1 by blast} \)

qed

from ntrs.rtr2trp[where \( c = \{ \# \} \), simplified, OF REVSPLIT'(5)] obtain \( sq'' ce22e' w22' \) where R-CONV': \( ce22e' = \{ \# sq'' \} + ce22e' \) \( w22'' = \text{map le-rem-s w22' ([\{ entry fg q' ][\#] ), w22', (sq'', ce22e')} \) \( \in \text{trcl (ntr fg)} \) by blast

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

from RU-sound[OF R-CONV(3) ATUV(1)] obtain \( Ml Me h \) where RU:
\( (\text{entry fg q}, Ml, Me, h) \in RU-cs fg U Ml \subseteq \text{mon-loc fg w2w1 Me} \subseteq \text{mon-env fg w2w1 h} \leq a\alpha h \) (map (\( a\alpha h fg \)) w2w1) by blast

—
from RU-sound[OF R-CONV(3), of V'] REVSPLIT'(4) R-CONV'(1) obtain \( Ml Me' h' \) where RV:
\( (\text{entry fg q', Ml', Me', h'}) \in RU-cs fg V Ml' \subseteq \text{mon-loc fg w2w2' Me'} \subseteq \text{mon-env fg w2w2' } h' \leq a\alpha h \) (map (\( a\alpha h fg \)) w2w2') by auto

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

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

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

next

from ab-interleavable[OF REVSPLIT(3)] have aαh (map (\( a\alpha h fg \)) w2w1)
\([*] a\alpha h (\text{map (a\alpha h fg) w2w1}) \).

[...] h' \textit{proof}

erule-tac ab-ileq-il

— note RU(4)

— also have \( \text{map (a\alpha n fg) w2w1 } \leq \text{map (a\alpha n fg) w2w1 using R-CONV(2)} \)
by (simp add: a\alpha n-ileq)

— hence \( \text{a\alpha h (map (a\alpha n fg) w2w1) } \leq \text{a\alpha h (map (a\alpha n fg) w2w1) by (rule a\alpha h-ileq} \)
finally show \( h \leq \alpha\text{ah} (\text{map} \ (\alpha\text{nf} \ fg \ w21) \ . \)

next

note \( RV(4) \)

also have \( \text{map} \ (\alpha\text{nl} \ fg \ w22') \preceq \text{map} \ (\alpha\text{nf} \ fg \ w22) \) using \( \text{R-CONV}'(2) \) \( \text{REVSP\textit{LIT}}'(2) \) by \( (\text{simp add:} \alpha\text{nl}[\text{symmetric}] \text{ le-list-map map-map}[\text{symmetric}] \) del: map-map)

hence \( \alpha\text{ah} \ (\text{map} \ (\alpha\text{nl} \ fg \ w22') \leq \alpha\text{ah} \ (\text{map} \ (\alpha\text{nf} \ fg \ w22) \) by \( (\text{rule} \ \alpha\text{ah-ileq}) \)

finally show \( h' \leq \alpha\text{ah} \ (\text{map} \ (\alpha\text{nf} \ fg \ w22) \ . \)

qed

qed \( (\text{simp}) \)

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

moreover have \( \text{Ml} \cup \text{Me} \cup \text{Me}' \subseteq \text{mon-env} \ fg \ (ee\#ww) \) using \( RV(2,3) \) \( \text{RU}(2,3) \) \( \text{mon-ww-ileq}[\text{OF REVSP\textit{LIT}}'(2), \text{of fg}] \) \( \text{MON2-LEQ} \) \( \text{R-CONV}'(2) \) \( \text{R-CONV}'(2) \)

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

qed

next

case \( \ell \) — The first node is reached from the local thread, the second one from a spawned thread

from \( \text{RU-sound}[\text{OF DECOMP-LOC}(2), \text{of U}] \) \( \text{br}(1) \) \( \text{DECOMP-LOC}(1) \) obtain \( \text{Ml} \text{ Me} \text{ h} \) where \( \text{RU}: \ (v, \text{ Ml}, \text{ Me}, \text{ h}) \in \text{RU-cs fg} \ U \text{ Ml} \subseteq \text{mon-loc} \ fg \ w1 \text{ Me} \subseteq \text{mon-env} \ fg \ w1 \) \( \text{Me} \subseteq \text{mon-env} \ fg \ w1 \) \( \text{Me} \subseteq \text{mon-env} \ fg \ w1 \) by \( \text{auto} \)

obtain \( \text{Ml}' \text{ Me}' \ h' \) \( q' \) where \( \text{RV}: \ (\text{entry fg} \ q') :# \ ch \ (\text{entry fg} \ q', \text{ Ml}', \text{ Me}', \ h') \in \text{RU-cs fg} \ V \text{ Ml}' \subseteq \text{mon-ww} \ fg \ w2 \text{ Me}' \subseteq \text{mon-ww} \ fg \ w2 \ h' \leq \alpha\text{ah} \ (\text{map} \ (\alpha\text{nf} \ fg \ w2) \)

proof —

case \( \text{goal1} \)

— We have to extract the interesting thread from the spawned threads in order to get an entry in \( \text{RU} \ fg \ V \)

obtain \( q' \text{ w2}' \text{ c2i}' \) where \( \text{REVSP\textit{LIT}}: \ (\text{entry fg} \ q' \) :# \ ch \text{ w2}' \leq \text{w2} \text{ c2i}' \leq \text{c2}' \text{ atU} \ V \text{ c2i}' \)

using \( \text{ntr-reverse-split-atU}[\text{OF - br}(2) \) \( \text{LESP\textit{LIT}}(4)] \) \( \text{ntrp-valid-preserve-s}[\text{OF SPLIT}(1), \text{simplied}] \) \( \text{CHFMT} \) by \( (\text{simp add:} \text{ valid-unconc} \) blast

from \( \text{ntrs\text{tr}2\text{trrp}[\text{where c=}{\#}, \text{ simplified}, \text{ OF REVSP\textit{LIT}}(5)] \) obtain \( \text{s2i}' \text{ c2i}' \text{ w2w}' \) where \( \text{R-CONV:} \text{ c2i}'=\{\#s2i'\#\}+\text{c2i}' \text{ w2w}'=\text{le-rem-s w2w}' \)

from \( \text{RU-sound}[\text{OF R-CONV}(3), \text{of V}] \) \( \text{REVSP\textit{LIT}}(4) \) \( \text{R-CONV}(1) \) obtain \( \text{Ml'} \text{ Me'} \ h' \) where \( \text{RV}: \ (\text{entry fg} \ q', \text{ Ml'}, \text{ Me'}, \ h') \in \text{RU-cs fg} \ V \text{ Ml}' \subseteq \text{mon-loc} \ fg \ w2' \text{ Me}' \subseteq \text{mon-env} \ fg \ w2' \ h' \leq \alpha\text{ah} \ (\text{map} \ (\alpha\text{nl} \ fg \ w2') \) by \( \text{auto} \)

moreover have \( \text{mon-loc} \ fg \ w2' \subseteq \text{mon-ww} \ fg \ w2 \text{ mon-env} \ fg \ w2' \subseteq \text{mon-ww} \ fg \ w2 \) using \( \text{mon-ww-ileq}[\text{OF REVSP\textit{LIT}}(2), \text{of fg}] \) \( \text{R-CONV}(2) \) by \( (\text{auto simp add:} \text{ mon-ww-of-le-rem}) \)

moreover have \( \alpha\text{ah} \ (\text{map} \ (\alpha\text{nl} \ fg \ w2') \leq \alpha\text{ah} \ (\text{map} \ (\alpha\text{nf} \ fg \ w2) \) using \( \text{REVSP\textit{LIT}}(2) \) \( \text{R-CONV}(2) \) by \( (\text{auto simp add:} \alpha\text{nl}[\text{symmetric}] \text{ le-list-map map-map}[\text{symmetric}] \) simp del: map-map intro: \( \alpha\text{ah-ileq del: predicate}\) \( \text{I} \)

ultimately show \( ?\text{thesis using goal1 REVSP\textit{LIT}}(1) \) by \( (\text{blast intro: order-trans}) \)

qed

from \( \text{S-ENTRY-PAT}[\text{of} \ \{\#q'\#\}, \text{ simplified}] \) \( \text{RV}(1) \) have \( \text{S-ENTRY}: \ (v,
\[ \text{proof} \]

\[ \text{proof (rule RUV-split-le (OF FS-FMT (2,4) S-ENTRY - RU (1) RV (2)))} \]

\[ \text{from MON-PU MON1-LEQ MON2-LEQ RV (3,4) show (mon-n fg u \cup mon-fg w) \cap (Me \cup Mi' \cup Me') = \{\} by blast} \]

\[ \quad \text{next} \]

\[ \quad \text{from ah-interleavable (OF LESPLIT (1)) have ah (map (onl fg) w1) [s] \quad \text{aah (map (onl fg) w2) by simp}} \]

\[ \quad \text{thus h [s] h' using RV (4) RV (5) by (auto elim: ah-leq-il)} \]

\[ \quad \text{qed (simp)} \]

\[ \text{moreover have mon-fg p \cup mon-w fg w \cup Mi \subseteq mon-loc fg (ee \# ww) using FS-FMT (1) MON1-LEQ RV (2) by (simp) blast} \]

\[ \text{moreover have Me \cup Mi' \cup Me' \subseteq mon-env fg (ee \# ww) using MON1-LEQ MON2-LEQ RV (3) RV (3,4) by (simp) blast} \]

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

\[ \text{next} \]

\[ \text{case rl} \quad \text{— The second node is reached from the local thread, the first one from a spawned thread. This case is symmetric to the previous one} \]

\[ \text{from RU-sound (OF DECOMP-LOC (2), of V) rl (1) DECOMP-LOC (1) obtain Ml Me h where RV: (v, Ml, Me, h) \in RU-cs fg V Ml \subseteq mon-loc fg w1 Me \subseteq mon-env fg w1 h \leq aah (map (onl fg) w1) by auto} \]

\[ \text{obtain Ml' Me' h' q' where RV: \{entry fg q' \} : \#} \quad \text{ch (entry fg q', Ml', Me', h')} \in RU-cs fg U Ml' \subseteq mon-ww fg w2 Me' \subseteq mon-ww fg w2 h' \leq aah (map (onl fg) w2) \]

\[ \quad \text{proof} \quad \text{—} \]

\[ \quad \text{case goal1} \]

\[ \quad \text{— We have to extract the interesting thread from the spawned threads in order to get an entry in RU fg V} \]

\[ \quad \text{obtain q' w2' c2'i where REV_SPLIT: \{entry fg q' \} : \#} \quad \text{ch w2' \leq w2 c2'i \leq c2' at U} \quad \text{c2' \in trcl (ntrp fg)} \]

\[ \quad \text{using ntr-reverse-split-atU (OF - rl (2) LESPLIT (4)) ntrp-valid-preserve-s (OF SPLIT (1), simplified) CHFM} \quad \text{by (simp add: valid-unconc) blast} \]

\[ \quad \text{from ntrs, gtr2gtrp (where c=\{\}, simplified, OF REV_SPLIT (5)) obtain s2'i c2'ie w2o' where R-CONV: c2'i=(\#s2'i\#)+c2'ie \leq \text{map le-rem-s w2w}'} \]

\[ \quad \text{\texttt{(entry fg q', \{\}, \#)}}, \quad \text{w2w', s2'i, c2'ie} \in \text{trcl (ntrp fg)} \]

\[ \quad \text{\texttt{from RU-sound (OF R-CONV (3), of U) REV_SPLIT (4) R-CONV (1) obtain Ml' Me' h' where RU: (entry fg q', Ml', Me', h') \in RU-cs fg U Ml' \subseteq mon-loc fg w2w' Me' \subseteq mon-env fg w2w' h' \leq aah (map (onl fg) w2w') by auto} \]

\[ \quad \text{moreover have mon-loc fg w2w' \subseteq mon-ww fg w2 mon-env fg w2w' \subseteq mon-ww fg w2 using mon-ww-ileq (OF REV_SPLIT (2), of fg) R-CONV (2) by (auto simp add: mon-ww-of-le-rem)} \]

\[ \quad \text{moreover have aah (map (onl fg) w2w') \leq aah (map (onl fg) w2) using REV_SPLIT (2) R-CONV (2) by (auto simp add: onl-onl [symmetric] le-list-map map-map [symmetric] simp del: map-map intro: aah-ileq del: predicate2I)} \]

\[ \quad \text{ultimately show ?thesis by blast intro: order-trans} \]

\[ \quad \text{qed} \]

\[ \text{from S-ENTRY-PAT (OF \{\#, \}, simplified) RU (1) have S-ENTRY: (v, mon-w fg w, \{\#, \}) \in S-cs fg f by simp} \]

\[ \text{have (u, mon-fg p \cup mon-w fg w \cup Mi, Me \cup Mi' \cup Me') \in RU-cs fg U V by simp} \]

\[ \text{proof (rule RUV-split-el (OF FS-FMT (2,4) S-ENTRY - RV (1) RU (2)))} \]

\[ \text{108} \]
\textbf{lemma} (in flowgraph) RUV-precise: \((u, Ml, Me) \in RUV-cs fg U V\)
\[
\implies \exists w s' c'.
\]
\[
\{([u], \emptyset), w, (s', c')\} \in \text{trcl} (\text{ntrp} fg) \land
\]
\[
\text{atUV } U V (\{\#s\#\} + c') \land
\]
\[
\text{mon-loc } fg w = Ml \land
\]
\[
\text{mon-env } fg w = Me
\]
\textbf{proof} (induct rule: RUV-cs.induct)
\begin{enumerate}
\item \textbf{case} \((\text{RUv-call } u p u' v M P Me)\) then obtain \(ww s' c'\) where IH: \((\{[v], \emptyset\}), \text{LOC } (LCall p \# w), [v, u'], ch) \in \text{ntrp} fg P = \emptyset\) \(M = \text{mon-w } fg w\) \(\text{mon-n } fg v = \text{mon-c } fg ch = \emptyset\) by blast
\item from \(S\text{-precise-ntrp}[\text{OF RUV-call}(3,2,1), \text{simplified}]\) obtain \(w ch\) where FS: \((\{[u], \emptyset\}), \text{LOC } (LCall p \# w), [v, u'], ch) \in \text{ntrp} fg P = \emptyset\) \(M = \text{mon-w } fg w\) \(\text{mon-n } fg v \equiv \text{mon-c } fg ch = \emptyset\) by blast
\item note FS(1)
\item also have \((\{[v], u'_\upharpoonright, ch\}), \text{ww }, s' @ [u'_\upharpoonright], c' + ch) \in \text{trcl} (\text{ntrp} fg)\)
\item using \text{ntrp-addr-context}[\text{OF ntrp-stack-comp}[\text{OF IH}(1), \text{OF } [u'_\upharpoonright], \text{OF ch}, \text{simplified}]\]
\item FS(5) IH(4) RUV-call.hyps(6) \(\text{mon-n\text{-}same\text{-}proc}[\text{OF edges\text{-}part}[\text{OF RUV\text{-}call}\text{.hyps}(1)]]\) by simp
\item finally have \((\{[u], \emptyset\}), \text{LOC } (LCall p \# w) \# ww, s' @ [u'_\upharpoonright], c' + ch) \in \text{trcl} (\text{ntrp} fg)\)
\item moreover from IH(2) have \(\text{atUV } U V (\{\#s' @ [u'_\upharpoonright]\} + (c' + ch))\) by auto
\item moreover have \(\text{mon-loc } fg (\text{LOC } (LCall p \# w) \# ww) = \text{mon-fg } p \cup M \cup Ml\)
\item using IH(3) FS(3) by auto
\item moreover have \(\text{mon-env } fg (\text{LOC } (LCall p \# w) \# ww) = Me\) using IH(4) by auto
\item ultimately show \(\text{?thesis by blast}\)
\end{enumerate}
\textbf{next}
\begin{enumerate}
\item \textbf{case} \((\text{RUv-spawn } u p u' v M P q Ml Me)\) then obtain \(ww s' c'\) where IH: \((\{[\text{entry } fg q], \emptyset\}), \text{ww }, s', c') \in \text{trcl} (\text{ntrp} fg) \atUV U V (\{\#s'\#\} + c') \text{mon-loc } fg ww = Ml \text{mon-env } fg ww = Me\) by blast
\item from \(S\text{-precise-ntrp}[\text{OF RUV\text{-}spawn}(3,2,1), \text{simplified}]\) \text{mset\text{-}size\text{elem}[\text{OF - RUV\text{-}spawn}(4)]]\)
\item obtain \(w ch\) where FS: \((\{[u], \emptyset\}), \text{LOC } (LCall p \# w), [v, u'], \{\#[\text{entry } fg q]\#\} + ch) \in \text{ntrp} fg P=\{\#q\#\} \ M = \text{mon-w } fg w\) \(\text{mon-n } fg v = \text{mon-fg } p \text{mon-c } fg (\{\#[\text{entry } fg q]#\}) = \emptyset\)
\[q\#])+che\) = \{\} \text{ by (auto elim: mset-set-addE)}

moreover

have \(((\{v, u\}, \text{che} + \{\#\text{entry fg} q\#\}), \text{map} \text{ ENV} (\text{map} \text{ le-rem-s} \text{ww}), (\{v, u\}, \text{che} + \{\#s\#\} + c'))\) \in \text{trcl} (\text{ntrp fg})

using \text{ntrp2ntrp}[\text{OF} \text{grtp2grt}[\text{OF} \text{IH}(1)], \text{of} \ ([v, u], \text{che}) \text{ IH} (3, 4) \text{ RUV-spawn}(7) \text{ FS}(4, 5) \text{ mon-nsame-proc[OF} \text{edges-part}[\text{OF} \text{ RUV-spawn}(1)]\text{)]}

by (auto simp add: mon-c-unconc mon-ww-of-le-rem)

ultimately have \(((\{u\}, \{\#\}), \text{LOC} (\text{LCall} p \# w) \# \text{map} \text{ ENV} (\text{map} \text{ le-rem-s} \text{ww}), (\{v, u\}, \text{che} + \{\#s\#\} + c')) \in \text{trcl} (\text{ntrp fg})\) \text{ by (auto simp add: union-ac)}

moreover have \text{aiUV} U V ((\{\#v, u\} \#) + (\text{che} + \{\#s\#\} + c')) \text{ using IH}(2)

by auto

moreover have mon-loc fg \((\text{LOC} (\text{LCall} p \# w) \# \text{map} \text{ ENV} (\text{map} \text{ le-rem-s} \text{ww}) = \text{mon-fg} p \cup M \text{ using FS}(3) \text{ by (simp del: map-map)}\)

moreover have mon-env fg \((\text{LOC} (\text{LCall} p \# w) \# \text{map} \text{ ENV} (\text{map} \text{ le-rem-s} \text{ww}) = Ml \cup M e \text{ using IH}(3, 4) \text{ by (auto simp add: mon-ww-of-le-rem simp del: map-map} \)

ultimately show \text{case} by blast

next

case \((\text{RUV-split-le} u p u' v M P q Ml Me h Ml' Me' h')\)

— Get paths from previous results

from \text{S-precise-ntrp}[\text{OF} \text{ RUV-split-le}(3, 2, 1), \text{simplied}] \text{ mset-size1elem[OF} \text{-RUV-split-le}(4)\text{]} \text{ obtain w che where}

\text{FS:} ((\{u\}, \{\#\}), \text{LOC} (\text{LCall} p \# w), (\{v, u\}, \{\#\text{entry fg} q\#\} + \text{che}) \in \text{ntrp fg} \text{ P} = \{\#q\#\} \text{ M} = \text{mon-w fg} w \text{ mon-n fg} v = \text{mon-fg} p \text{ mon-c fg} ((\{\#\text{entry fg} q\#\} + \text{che}) = \{\}) \text{ by (auto elim: mset-set-addE)}

from \text{RU-precise}[\text{OF} \text{ RUV-split-le}(5)] \text{ obtain w1 s1' c1' where P1:} ((\{v\}, \{\#\}), \text{ww1}, s1', c1' \in \text{trcl} (\text{ntrp fg}) \text{ atU} U ((\{s1'\#\} + c1') \text{ mon-loc fg} \text{ww1} = Ml \text{ mon-env fg} \text{ww1} = Me \text{ aah (map (envl fg) \text{ww1}) = h by blast})

from \text{RU-precise}[\text{OF} \text{ RUV-split-le}(6)] \text{ obtain ww2 s2' c2' where P2:} ((\{\text{entry fg} q\}, \{\#\}), \text{ww2}, s2', c2') \in \text{trcl} (\text{ntrp fg}) \text{ atU} V ((\{s2'\#\} + c2') \text{ mon-loc fg} \text{ww2} = Ml' \text{ mon-env fg} \text{ww2} = Me' \text{ aah (map (envl fg) \text{ww2}) = h by blast})

— Get combined path from the acquisition history interleavability, need to remap loc/env-steps in second path

from \text{P2}(5) \text{ have aah (map (envl fg) (map ENV (map le-rem-s \text{ww2})) = h' by (simp add: an-envl o-assoc)}

with \text{P1}(5) \text{ RUV-split-le}(8) \text{ obtain ww where IL:} \text{ww \in} \text{ww1} @ \text{\text{aah} (map ENV (map le-rem-s \text{ww}))} \text{ by (force)}

— Use the ntrip-unsplit-theorem to combine the executions

from \text{ntrp-unsplit[where ca=\{\#\},OF} \text{IL P1}(1) \text{ grtp2grt[OF P2}(1)\text{], simplified}] \text{ have ((\{v\}, \{\#\text{entry fg} q\#\}), \text{ww}, s1', c1' + (\{\#s2'\#\} + c2')) \in \text{trcl} (\text{ntrp fg}) \text{ using FS}(4, 5) \text{ RUV-split-le}(7)\text{ by (auto simp add: mon-c-unconc \text{mon-ww-of-le-rem P2}(3, 4))}

from \text{ntrp-add-context[OF} \text{ntrp-stack-comp[OF} \text{this}, \text{of} \text{[u']}, \text{of che} \text{ have (([v] @ [u'], \{\#\text{entry fg} q\#\} + \text{che}), \text{ww}, s1' @ [u'], c1' + (\{\#s2'\#\} + c2') + \text{che}) \in \text{trcl} (\text{ntrp fg})}

using \text{mon-nsame-proc[OF} \text{edges-part[OF} \text{ RUV-split-le}(1)\text{]} \text{ mon-loc-cil[OF} \text{IL}, \text{of fg} \text{ mon-env-cil[OF} \text{IL}, \text{of fg} \text{ FS}(4, 5) \text{ RUV-split-le}(7) \text{ by (auto simp add: mon-c-unconc P1}(3, 4) \text{ P2}(3, 4) \text{ mon-ww-of-le-rem simp del: map-map})

with \text{FS}(1) \text{ have (([u], \{\#\}), \text{LOC} (\text{LCall} p \# w) \# \text{ww}, (s1' @ [u'], c1' +
[((#s2' #) + c2') + (che)) ∈ trcl (ntrp fg) by simp
  moreover have atUV U V (((#s1' @ [u'] #) + (c1' + ((#s2' #) + c2') + (che)))
using P1(2) P2(2) by auto
  moreover have mon-loc fg (LOC (LCall p ≠ w) ≠ ww) = mon fg p ∪ M ∪ Ml
using FS(3) P1(3) mon-loc-cil([OF IL, of fg]) by (auto simp del: map-map)
  moreover have mon-env fg (LOC (LCall p ≠ w) ≠ ww) = Me ∪ Ml' ∪ Me' using
P1(4) P2(3,4) mon-env-cil([OF IL, of fg]) by (auto simp del: mon-ww-of-le-rem simp del: map-map)
ultimately show ?case by blast
next
  case (RUV-split-el u p u' v M p q Ml Me h Ml' Me' h') — This is the symmetric
case to RUV-split-le, it is proved completely analogously, just need to swap U and
V.
  — Get paths from precision results
from S-precise-ntrp([OF RUV-split-el(3,2,1)], simplified] mset-size1elem([OF
- RUV-split-el(4)]) obtains w che where
  FS: (\{[u], {#}], LOC (LCall p ≠ w), [v, u'], {#[entry fg q] #} + (che) ∈ ntrp
  fg P=(#q]) M = mon-w fg w mon-n fg v = mon fg p mon-c fg (\{#[entry fg
  q] #} + (che) = {}) (auto elim: mset-leadE)
from RU-precise([OF RUV-split-el(5)]) obtain wv1 s1' c1' where P1: (\{[v],
  {#}], wv1, s1', c1') ∈ trcl (ntrp fg) atUV V (s1') + c1') mon-loc fg wv1 =
Ml mon-env fg wv1 = Ml env fg (map (λ v. fg) wv1) = h' by blast
from RU-precise([OF RUV-split-el(6)]) obtain wv2 s2' c2' where P2: (\{entry
  fg q], {#}], wv2, s2', c2'] ∈ trcl (ntrp fg) atUV U (s2') + c2') mon-loc fg
wv2 = Ml' mon-env fg wv2 = Me' env (map (λ v. fg) wv2) = h' by blast
  — Get combined path from the acquisition history interleaveability, need to remap
loc/env-steps in second path
from P2(5) have αah (map (λ v. fg) (map le-rem-s (ww2))) = h' by
(simp add: on-cntl a-assoc)
  with P1(5) RUV-split-el(8) obtain ww where IL: ww∈ww1 ⊗ αnl fg(map ENV
(map le-rem-s (ww2))) using ah-interleaveable2 by (force)
  — Use the ntrp-unspl-theorem to combine the executions
from ntrp-unsplit[where ca=\{#]] OF IL P1(1) gtrp2gtr([OF P2(1)], simplified]
have (\{[v], {#]<entry fg q] #}], ww, s1', c1' + (\{#s2' #] + (che) ∈ trcl (ntrp fg)
using FS(4,5) RUV-split-el(7)
  by (auto simp add: mon-c-uncon mon-ww-of-le-rem P2(3,4))
from ntrp-add-context([OF ntrp-stack-comp([OF this, of [u']], of che] have (\{[v]
  @ [u'], {#]<entry fg q] #}], ww, s1' @ [u'], c1' + (\{#s2' #] + (che) ∈ trcl (ntrp fg)
using mon-n-same-proc([OF edges-part([OF RUV-split-el(1)])] mon-loc-cil([OF
IL, of fg]) mon-env-cil([OF IL, of fg]) FS(4,5) RUV-split-el(7) by (auto simp add:
mon-c-uncon P1(3,4) P2(3,4) mon-ww-of-le-rem simp del: map-map)
  with FS(1) have (\{[v], {#}], LOC (LCall p ≠ w) ≠ ww, (s1' @ [u'], c1' +
(\{#s2' #] + (che) ∈ trcl (ntrp fg) by simp
moreover have atUV U V (s1' + (\{#s2' #] + (che) ∈ trcl (ntrp fg) by simp
using P1(2) P2(2) by auto
moreover have mon-loc fg (LOC (LCall p ≠ w) ≠ ww) = mon fg p ∪ M ∪ Ml
using FS(3) P1(3) mon-loc-cil([OF IL, of fg]) by (auto simp del: map-map)
moreover have mon-env fg (LOC (LCall p ≠ w) ≠ ww) = Me ∪ Ml' ∪ Me' 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 che where
\[ FS: \{ \{ [u], \{ \# \} \}, LOC (LCall p \# w), \{ [v, u'] \}, \{ [entry fg q]\# \} + \{ [entry fg q']\# \} + \{ \#[entry fg q]\# \} + \{entry fg q'\# \} + che \} \in ntrp fg P = \{ \{ q\# \} + \{ q'\# \} \} \]

—and have mon-loc fg ((\{ [entry fg q]\# \} + \{ [entry fg q']\# \} + \{ \#[entry fg q]\# \} + \{entry fg q'\# \} + che) = \{}

by (auto elim: mset-le-addE)

from RU-precise[OF RUV-split-ee(5)] obtain \(w_1 s_1' c_1'\) where \(P_1: \{ [entry fg q], \{ \# \}, w_1, s_1', c_1' \} \in trcl (ntrp fg) aU V \{ [\#s_1'\#] + c_1' \} \)

—and have mon-loc fg \((\{ [entry fg q]\# \} + \{ [entry fg q']\# \}, w_2, s_2', c_2'\} \in trcl (ntrp fg) aU V \{ [\#s_2'\#] + c_2' \}) \)

— 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 \(w_1\))) = h oah (map (on fg) (map le-rem-s \(w_2\))) = h' by (auto simp add: on-anl o-assoc)

-with RUV-split-ee(8) obtain \(w\) where \(IL: w_2 \in (map (on-rem-s \(w_1\)) \odot \text{an fg}) (map (on-rem-s \(w_2\))) \)

—and have the ntr-unsplit-theorem to combine the executions
from ntr-unsplit[OF IL gtrp2gtr[OF \(P_1(1)\) gtrp2gtr[OF \(P_2(1)\)], simplified] have \((\{ [entry fg q]\# \} + \{ [entry fg q']\# \}, w_2, s_2', c_2'\} \in trcl (ntrp fg) using ah-interleavable2 by (force simp del: map-map)

— Prepend first step
from ntr2ntrp[OF \(P_1(1)\), of \([v,u']\) che] have ((\([v,u']\), che + (\{ [entry fg q]\# \} + \{ [entry fg q']\# \}), map ENV \(w_2, [v, u']\), che + (\{ [\#s_1'\#] + c_1' + (\{ [\#s_2'\#] + c_2'\}) \})) \in trcl (ntrp fg)

using RUV-split-ee(7) \(FS(5)\) mon-ww-cil[OF IL, of fg] \(FS(4)\) mon-n-same-proc[OF edges-part][OF RUV-split-ee(1)] by (auto simp add: mon-c-anconc mon-ww-cil[OF IL, of fg] \(P_1(3,4)\) \(P_2(3,4)\))

with \(FS(1)\) have ((\([v, u']\), \{ \# \}), LOC (LCall p \# w) \# map ENV \(w_2, [v, u']\), che + (\{ [\#s_1'\#] + c_1' + (\{ [\#s_2'\#] + c_2'\}) \})) \in trcl (ntrp fg) by (auto simp add: union-ac)

moreover have aU V U V (\([v, u']\#\} + (che + (\{ [\#s_1'\#] + c_1' + (\{ [\#s_2'\#] + c_2'\})\})) using \(P_1(2) \ P_2(2)\) by auto

moreover have mon-loc fg (LOC (LCall p \# w) \# map ENV \(w\)) = mon fg p \(\cup M\) using \(FS(3)\) by auto

moreover have mon-env fg (LOC (LCall p \# w) \# map ENV \(w\)) = 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-locale. Note that we show in Section 6.4 that there is at least one non-trivial model of $eflowgraph$.
- The reference point semantics ($refpoint$) and the transitive closure operator ($trcl$).
- The definition of $atUV$.
- All dependencies of the above definitions in the Isabelle standard libraries.

theorem (in eflowgraph) RUV-is-sim-reach:
$\exists w c'. (\{#[entry fg (main fg)]#\}, w, c') \in trcl (refpoint fg) \land atUV U V c'$
$\iff (\exists Ml Me. (entry fg (main fg), Ml, Me) \in RUV$-cs $fg U V)$

— The proof uses the soundness and precision theorems wrt. to normalized paths (flowgraph.RUV-sound, flowgraph.RUV-precise) as well as the normalization result, i.e. that every reachable configuration is also reachable using a normalized path (eflowgraph.normalize) and, vice versa, that every normalized path is also a usual path (ntr-is-tr). Finally the conversion between our working semantics and the semantic reference point is exploited (flowgraph.refpoint-eq).

proof

case goal1 then obtain $w c'$ where $C$: $(\{#[entry fg (main fg)]#\}, w, c') \in trcl (tr fg) atUV U V c'$ by (auto simp add: refpoint-eq)

from normalize[OF C(1), of main fg, simplified] obtain $ww$ where $(\{#[entry fg (main fg)]#\}, ww, c') \in trcl (ntr fg)$ by blast

from ntrsnr2grp[where $c=$\{#\}, simplified, OF this] obtain $s' ce' wwl$ where $1$: $c'=$\{#s'#\}$+$ce'$ $ww = map le-rem-s$ $wwl (([entry fg (main fg)], \{#\}), wwl, s', ce') \in trcl (ntrp fg)$ by blast

with $C(2)$ have $2$: $atUV U V \ (\{#s'#\}$+$ce'$) by auto

from RUV-sound[OF 1(3) 2] show $\exists Ml Me. (entry fg (main fg), Ml, Me) \in RUV$-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 $wwl s' c'$ where $P$: $(\{([entry fg (main fg)], \{#\}), wwl, s', c') \in trcl (ntrp fg) atUV U V \ (\{#s'#\} + c')$ by blast
from gtrp2grf[OF P(1)] have \( \{\# [\text{entry } fg (\text{main } fg)] \#\} \), map le-rem-s wwl, 
\( \{\#s'\#\} + c' \in \text{trcl } (\text{ntr } fg) \) by (auto) 
from ntr-is-tr[OF this] P(2) have \( \exists w c'. (\{\# [\text{entry } fg (\text{main } fg)] \#\} \), w, c' \in \text{trcl } (\text{tr } fg) \land \text{atUV } U V c' \) by blast 
thus \( \exists w c'. (\{\# [\text{entry } fg (\text{main } fg)] \#\} \), w, c' \in \text{trcl } (\text{refpoint } fg) \land \text{atUV } U V c' \) by (simp add: refpoint-eq)

qed
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


