

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`

December 12, 2009

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	8
3	Acquisition Histories	9
3.1	Definitions	10
3.2	Interleavability	10
3.3	Used monitors	11
3.4	Ordering	11
3.5	Acquisition histories of executions	11
3.6	Acquisition history backward update	12
4	Labeled transition systems	13
4.1	Definitions	13
4.2	Basic properties of transitive reflexive closure	13
4.2.1	Appending of elements to paths	14
4.2.2	Transitivity reasoning setup	15
4.2.3	Monotonicity	15
4.2.4	Special lemmas for reasoning about states that are pairs	15
4.2.5	Invariants	15
5	Thread Tracking	15
5.1	Semantic on multiset configuration	16
5.2	Invariants	17
5.3	Context preservation assumption	17
5.4	Explicit local context	18
5.4.1	Lifted step datatype	18
5.4.2	Definition of the loc/env-semantics	20
5.4.3	Relation between multiset- and loc/env-semantics	20
5.4.4	Invariants	20
6	Flowgraphs	21
6.1	Definitions	21
6.2	Basic properties	22
6.3	Extra assumptions for flowgraphs	23
6.4	Example Flowgraph	23
7	Operational Semantics	24
7.1	Configurations and labels	24
7.2	Monitors	24
7.3	Valid configurations	26

7.4	Configurations at control points	27
7.5	Operational semantics	28
7.5.1	Semantic reference point	28
7.6	Basic properties	29
7.6.1	Validity	29
7.6.2	Equivalence to reference point	30
7.6.3	Case distinctions	30
7.7	Advanced properties	32
7.7.1	Stack composition / decomposition	32
7.7.2	Adding threads	33
7.7.3	Conversion between environment and monitor restrictions	33
8	Normalized Paths	35
8.1	Semantic properties of restricted flowgraphs	35
8.2	Definition of normalized paths	36
8.3	Representation property for reachable configurations	36
8.4	Properties of normalized path	38
8.4.1	Validity	39
8.4.2	Monitors	39
8.4.3	Modifying the context	41
8.4.4	Altering the local stack	43
8.5	Relation to monitor consistent interleaving	44
8.5.1	Abstraction function for normalized paths	44
8.5.2	Monitors	45
8.5.3	Interleaving theorem	46
8.5.4	Reverse splitting	48
9	Constraint Systems	49
9.1	Same-level paths	50
9.1.1	Definition	50
9.1.2	Soundness and Precision	51
9.2	Single reaching path	52
9.2.1	Constraint system	52
9.2.2	Soundness and precision	54
9.3	Simultaneously reaching path	55
9.3.1	Constraint system	55
9.3.2	Soundness and precision	56
10	Main Result	57
11	Conclusion	58

1 Introduction

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

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

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

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

locks¹. We use monitors instead of locks. Monitors are syntactically bound to the program structure and thus well-nestedness is guaranteed statically. Additionally we directly support reentrant monitors. Our model cannot simulate well-nested locks where a lock statement and its corresponding unlock statement may be in different procedures (as in [5, 4]). As common programming languages like Java also use reentrant monitors rather than locks, we believe our model to be useful as well.

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

2 Monitor Consistent Interleaving

```
theory ConsInterleave
imports Interleave Misc
begin
```

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

2.1 Monitors of lists of monitor pairs

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

definition

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

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

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

<proof>
lemma *mon-pl-cons[simp]*: $\text{mon-pl } (e\#w) = \text{fst } e \cup \text{snd } e \cup \text{mon-pl } w$
<proof>

lemma *mon-pl-unconc*: $\text{!!}b. \text{mon-pl } (a@b) = \text{mon-pl } a \cup \text{mon-pl } b$
<proof>

lemma *mon-pl-ileq*: $w \preceq w' \implies \text{mon-pl } w \subseteq \text{mon-pl } w'$
<proof>

lemma *mon-pl-set*: $\text{mon-pl } w = \bigcup \{ \text{fst } e \cup \text{snd } e \mid e. e \in \text{set } w \}$
<proof>

fun

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

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

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

lemma *cil-last-case-split[cases set, case-names left right]*:
 $\llbracket w \in e1\#w1 \otimes_{\alpha} e2\#w2;$
 $\text{!!}w'. \llbracket w = e1\#w'; w' \in (w1 \otimes_{\alpha} e2\#w2);$
 $\text{fst } (\alpha e1) \cap \text{mon-pl } (\text{map } \alpha (e2\#w2)) = \{\} \rrbracket \implies P;$
 $\text{!!}w'. \llbracket w = e2\#w'; w' \in (e1\#w1 \otimes_{\alpha} w2);$
 $\text{fst } (\alpha e2) \cap \text{mon-pl } (\text{map } \alpha (e1\#w1)) = \{\} \rrbracket \implies P$
 $\rrbracket \implies P$
<proof>

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

$\llbracket w \in wa \otimes_{\alpha} wb;$
 $\llbracket w = \square; wa = \square; wb = \square \rrbracket \implies P;$
 $\llbracket wa = \square; w = wb \rrbracket \implies P;$
 $\llbracket w = wa; wb = \square \rrbracket \implies P;$
 $!!ea \ wa' \ w'. \llbracket w = ea \# w'; wa = ea \# wa'; w' \in wa' \otimes_{\alpha} wb;$
 $\quad \text{fst}(\alpha \ ea) \cap \text{mon-pl}(\text{map } \alpha \ wb) = \{\} \rrbracket \implies P;$
 $!!eb \ wb' \ w'. \llbracket w = eb \# w'; wb = eb \# wb'; w' \in wa \otimes_{\alpha} wb';$
 $\quad \text{fst}(\alpha \ eb) \cap \text{mon-pl}(\text{map } \alpha \ wa) = \{\} \rrbracket \implies P$
 $\rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *cil-induct'*[*case-names both-empty left-empty right-empty append*]: \llbracket
 $\bigwedge \alpha. P \ \alpha \ \square \ \square;$
 $\bigwedge \alpha \ ad \ ae. P \ \alpha \ \square \ (ad \ \# \ ae);$
 $\bigwedge \alpha \ z \ aa. P \ \alpha \ (z \ \# \ aa) \ \square;$
 $\bigwedge \alpha \ e1 \ w1 \ e2 \ w2. \llbracket$
 $\quad \llbracket \text{fst}(\alpha \ e1) \cap \text{mon-pl}(\text{map } \alpha \ (e2 \ \# \ w2)) = \{\} \rrbracket \implies P \ \alpha \ w1 \ (e2 \ \# \ w2);$
 $\quad \llbracket \text{fst}(\alpha \ e2) \cap \text{mon-pl}(\text{map } \alpha \ (e1 \ \# \ w1)) = \{\} \rrbracket \implies P \ \alpha \ (e1 \ \# \ w1) \ w2$
 $\implies P \ \alpha \ (e1 \ \# \ w1) \ (e2 \ \# \ w2)$
 $\rrbracket \implies P \ \alpha \ wa \ wb$
 $\langle \text{proof} \rangle$

lemma *cil-induct-fix* α : \llbracket
 $P \ \alpha \ \square \ \square;$
 $\bigwedge \ ad \ ae. P \ \alpha \ \square \ (ad \ \# \ ae);$
 $\bigwedge \ z \ aa. P \ \alpha \ (z \ \# \ aa) \ \square;$
 $\bigwedge \ e1 \ w1 \ e2 \ w2. \llbracket$
 $\quad \llbracket \text{fst}(\alpha \ e2) \cap \text{mon-pl}(\text{map } \alpha \ (e1 \ \# \ w1)) = \{\} \implies P \ \alpha \ (e1 \ \# \ w1) \ w2;$
 $\quad \text{fst}(\alpha \ e1) \cap \text{mon-pl}(\text{map } \alpha \ (e2 \ \# \ w2)) = \{\} \implies P \ \alpha \ w1 \ (e2 \ \# \ w2) \rrbracket$
 $\implies P \ \alpha \ (e1 \ \# \ w1) \ (e2 \ \# \ w2)$
 $\implies P \ \alpha \ v \ w$
 $\langle \text{proof} \rangle$

lemma *cil-induct-fix* α' [*case-names both-empty left-empty right-empty append*]: \llbracket
 $P \ \alpha \ \square \ \square;$
 $\bigwedge \ ad \ ae. P \ \alpha \ \square \ (ad \ \# \ ae);$
 $\bigwedge \ z \ aa. P \ \alpha \ (z \ \# \ aa) \ \square;$
 $\bigwedge \ e1 \ w1 \ e2 \ w2. \llbracket$
 $\quad \text{fst}(\alpha \ e1) \cap \text{mon-pl}(\text{map } \alpha \ (e2 \ \# \ w2)) = \{\} \implies P \ \alpha \ w1 \ (e2 \ \# \ w2);$
 $\quad \text{fst}(\alpha \ e2) \cap \text{mon-pl}(\text{map } \alpha \ (e1 \ \# \ w1)) = \{\} \implies P \ \alpha \ (e1 \ \# \ w1) \ w2$
 $\implies P \ \alpha \ (e1 \ \# \ w1) \ (e2 \ \# \ w2)$
 $\rrbracket \implies P \ \alpha \ wa \ wb$
 $\langle \text{proof} \rangle$

lemma [*simp*]: $w \otimes_{\alpha} \square = \{w\}$
 $\langle \text{proof} \rangle$

lemma *cil-contains-empty*[*rule-format, simp*]: $(\square \in wa \otimes_{\alpha} wb) = (wa = \square \wedge wb = \square)$
 $\langle \text{proof} \rangle$

lemma *cil-cons-cases*[*cases set, case-names left right*]: $\llbracket e\#w \in w1 \otimes_{\alpha} w2;$
 $\llbracket w1' . \llbracket w1 = e\#w1'; w \in w1' \otimes_{\alpha} w2; fst(\alpha e) \cap mon-pl(map \alpha w2) = \{\} \rrbracket \implies P;$
 $\llbracket w2' . \llbracket w2 = e\#w2'; w \in w1 \otimes_{\alpha} w2'; fst(\alpha e) \cap mon-pl(map \alpha w1) = \{\} \rrbracket \implies P$
 $\rrbracket \implies P$
 ⟨*proof*⟩

lemma *cil-set-induct*[*induct set, case-names empty left right*]: $\llbracket \alpha w1 w2. \llbracket$
 $w \in w1 \otimes_{\alpha} w2;$
 $\llbracket \alpha. P \rrbracket \llbracket \alpha \rrbracket \llbracket;$
 $\llbracket \alpha e w' w1' w2. \llbracket w' \in w1' \otimes_{\alpha} w2; fst(\alpha e) \cap mon-pl(map \alpha w2) = \{\};$
 $P w' \alpha w1' w2 \rrbracket \implies P (e\#w') \alpha (e\#w1') w2;$
 $\llbracket \alpha e w' w2' w1. \llbracket w' \in w1 \otimes_{\alpha} w2'; fst(\alpha e) \cap mon-pl(map \alpha w1) = \{\};$
 $P w' \alpha w1 w2' \rrbracket \implies P (e\#w') \alpha w1 (e\#w2')$
 $\rrbracket \implies P w \alpha w1 w2$
 ⟨*proof*⟩

lemma *cil-set-induct-fix* α [*induct set, case-names empty left right*]: $\llbracket w1 w2. \llbracket$
 $w \in w1 \otimes_{\alpha} w2;$
 $P \rrbracket \llbracket \alpha \rrbracket \llbracket;$
 $\llbracket e w' w1' w2. \llbracket w' \in w1' \otimes_{\alpha} w2; fst(\alpha e) \cap mon-pl(map \alpha w2) = \{\};$
 $P w' \alpha w1' w2 \rrbracket \implies P (e\#w') \alpha (e\#w1') w2;$
 $\llbracket e w' w2' w1. \llbracket w' \in w1 \otimes_{\alpha} w2'; fst(\alpha e) \cap mon-pl(map \alpha w1) = \{\};$
 $P w' \alpha w1 w2' \rrbracket \implies P (e\#w') \alpha w1 (e\#w2')$
 $\rrbracket \implies P w \alpha w1 w2$
 ⟨*proof*⟩

lemma *cil-cons1*: $\llbracket w \in wa \otimes_{\alpha} wb; fst(\alpha e) \cap mon-pl(map \alpha wb) = \{\} \rrbracket$
 $\implies e\#w \in e\#wa \otimes_{\alpha} wb$
 ⟨*proof*⟩

lemma *cil-cons2*: $\llbracket w \in wa \otimes_{\alpha} wb; fst(\alpha e) \cap mon-pl(map \alpha wa) = \{\} \rrbracket$
 $\implies e\#w \in wa \otimes_{\alpha} e\#wb$
 ⟨*proof*⟩

2.2 Properties of consistent interleaving

— Consistent interleaving is a restriction of interleaving

lemma *cil-subset-il*: $w \otimes_{\alpha} w' \subseteq w \otimes w'$
 ⟨*proof*⟩

lemma *cil-subset-il'*: $w \in w1 \otimes_{\alpha} w2 \implies w \in w1 \otimes w2$
 ⟨*proof*⟩

lemma *cil-set*: $w \in w1 \otimes_{\alpha} w2 \implies set w = set w1 \cup set w2$
 ⟨*proof*⟩

corollary *cil-mon-pl*: $w \in w1 \otimes_{\alpha} w2$
 $\implies mon-pl(map \alpha w) = mon-pl(map \alpha w1) \cup mon-pl(map \alpha w2)$
 ⟨*proof*⟩

lemma *cil-length*[*rule-format*]: $\forall w \in wa \otimes_{\alpha} wb. length w = length wa + length wb$
 ⟨*proof*⟩

lemma *cil-ileq*: $w \in w1 \otimes_{\alpha} w2 \implies w1 \preceq w \wedge w2 \preceq w$
 ⟨proof⟩

lemma *cil-commute*: $w \otimes_{\alpha} w' = w' \otimes_{\alpha} w$
 ⟨proof⟩

lemma *cil-assoc1*: $!!wl\ w1\ w2\ w3. \llbracket w \in w1 \otimes_{\alpha} w3; wl \in w1 \otimes_{\alpha} w2 \rrbracket$
 $\implies \exists wr. w \in w1 \otimes_{\alpha} wr \wedge wr \in w2 \otimes_{\alpha} w3$
 ⟨proof⟩

lemma *cil-assoc2*:
assumes $A: w \in w1 \otimes_{\alpha} wr$ **and** $B: wr \in w2 \otimes_{\alpha} w3$
shows $\exists wl. w \in wl \otimes_{\alpha} w3 \wedge wl \in w1 \otimes_{\alpha} w2$
 ⟨proof⟩

lemma *cil-map*: $w \in w1 \otimes_{(\alpha \circ f)} w2 \implies \text{map } f\ w \in \text{map } f\ w1 \otimes_{\alpha} \text{map } f\ w2$
 ⟨proof⟩

end

3 Acquisition Histories

theory *AcquisitionHistory*
imports *ConsInterleave*
begin

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

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

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

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

3.1 Definitions

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

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

lemma $ah\ cases[cases\ set]: \llbracket h \in ah; h\ m = \{\} \implies P ; m \in h\ m \implies P \rrbracket \implies P$
<proof>

3.2 Interleavability

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

definition

$ah\ il :: ('m \Rightarrow 'm\ set) \Rightarrow ('m \Rightarrow 'm\ set) \Rightarrow bool$ (**infix** $[*]$ 65)

where

$h1\ [*]\ h2 == \neg(\exists m1\ m2. m1 \in h2\ m2 \wedge m2 \in h1\ m1)$

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

lemma $ah\ il\ lockset\ disjoint:$

$\llbracket h1 \in ah; h2 \in ah; h1\ [*]\ h2 \rrbracket \implies h1\ m = \{\} \vee h2\ m = \{\}$

<proof>

Of course, acquisition history interleavability is commutative

lemma $ah\ il\ commute: h1\ [*]\ h2 \implies h2\ [*]\ h1$

<proof>

3.3 Used monitors

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

definition

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

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*: $\llbracket h1\ [*]\ h2; h1' \leq h1; h2' \leq h2 \rrbracket \Longrightarrow h1' [*]\ h2'$
<proof>

lemma *ah-leq-il-left*: $\llbracket h1\ [*]\ h2; h1' \leq h1 \rrbracket \Longrightarrow h1' [*]\ h2$ **and**
ah-leq-il-right: $\llbracket h1\ [*]\ h2; h2' \leq h2 \rrbracket \Longrightarrow h1\ [*]\ h2'$
<proof>

3.5 Acquisition histories of executions

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

consts $\alpha ah :: ('m\ set \times 'm\ set)\ list \Rightarrow 'm \Rightarrow 'm\ set$

primrec

$\alpha ah\ []\ m = \{ \}$
 $\alpha ah\ (e\#w)\ m = (if\ m \in fst\ e\ then\ fst\ e \cup snd\ e \cup mon-pl\ w\ else\ \alpha ah\ w\ m)$

— αah generates valid acquisition histories

lemma $\alpha ah-ah$: $\alpha ah\ w \in ah$

<proof>

lemma $\alpha ah-hd$: $\llbracket m \in fst\ e; x \in fst\ e \cup snd\ e \cup mon-pl\ w \rrbracket \Longrightarrow x \in \alpha ah\ (e\#w)\ m$
<proof>

lemma $\alpha ah-tl$: $\llbracket m \notin fst\ e; x \in \alpha ah\ w\ m \rrbracket \Longrightarrow x \in \alpha ah\ (e\#w)\ m$
<proof>

lemma $\alpha ah-cases$ [*cases set, case-names hd tl*]: \llbracket

$x \in \alpha ah\ w\ m;$
 $!!e\ w'. \llbracket w = e\#w'; m \in fst\ e; x \in fst\ e \cup snd\ e \cup mon-pl\ w' \rrbracket \Longrightarrow P;$
 $!!e\ w'. \llbracket w = e\#w'; m \notin fst\ e; x \in \alpha ah\ w'\ m \rrbracket \Longrightarrow P$

]] $\implies P$
 <proof>

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

lemma mon-ah-subset : $\text{mon-ah } (\alpha\text{ah } w) \subseteq \text{mon-pl } w$
 <proof>

lemma $\alpha\text{ah-ileq}$: $w1 \preceq w2 \implies \alpha\text{ah } w1 \leq \alpha\text{ah } w2$
 <proof>

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

lemma ah-interleavable1 :

$w \in w1 \otimes_{\alpha} w2 \implies \alpha\text{ah } (\text{map } \alpha \ w1) \ [*] \ \alpha\text{ah } (\text{map } \alpha \ w2)$
 — The lemma is shown by induction on the structure of the monitor consistent interleaving operator
 <proof>

lemma ah-interleavable2 :

assumes A : $\alpha\text{ah } (\text{map } \alpha \ w1) \ [*] \ \alpha\text{ah } (\text{map } \alpha \ w2)$
shows $w1 \otimes_{\alpha} w2 \neq \{\}$
 — This lemma is shown by induction on the sum of the word lengths
 <proof>

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

theorem ah-interleavable :

$(\alpha\text{ah } (\text{map } \alpha \ w1) \ [*] \ \alpha\text{ah } (\text{map } \alpha \ w2)) \longleftrightarrow (w1 \otimes_{\alpha} w2 \neq \{\})$
 <proof>

3.6 Acquisition history backward update

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

definition

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

Intuitively, $\text{ah-update } h \ (E, U) \ M \ m$ means to prepend a step (E, U) to the acquisition history h of a path that uses monitors M . Note that we need the

extra parameter M , since an acquisition history does not contain information about the monitors that are used on a path before the first monitor that will not be left has been entered.

lemma *ah-update-cons*: $\alpha ah (e\#w) = ah\text{-update} (\alpha ah w) e (mon\text{-pl } w)$
 ⟨proof⟩

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*: $\llbracket h \leq h'; F=F'; M \subseteq M' \rrbracket$
 $\implies ah\text{-update } h F M \leq ah\text{-update } h' F' M'$
 ⟨proof⟩

lemma *ah-update-mono2*: $\llbracket h \leq h'; U \subseteq U'; M \subseteq M' \rrbracket$
 $\implies ah\text{-update } h (E,U) M \leq ah\text{-update } h' (E,U') M'$
 ⟨proof⟩

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

types $('c, 'a) LTS = ('c \times 'a \times 'c) \text{ set}$

Transitive reflexive closure

inductive-set

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

for *t*

where

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

| *cons*[*simp*]: $\llbracket (c, a, c') \in t; (c', w, c'') \in trcl t \rrbracket \implies (c, a\#w, c'') \in trcl t$

4.2 Basic properties of transitive reflexive closure

lemma *trcl-empty-cons*: $(c, [], c') \in trcl t \implies (c=c')$
 ⟨proof⟩

lemma *trcl-empty-simp*[*simp*]: $(c, [], c') \in \text{trcl } t = (c = c')$
 ⟨*proof*⟩

lemma *trcl-single*[*simp*]: $((c, [a], c') \in \text{trcl } t) = ((c, a, c') \in t)$
 ⟨*proof*⟩

lemma *trcl-uncons*: $(c, a \# w, c') \in \text{trcl } t \implies \exists ch . (c, a, ch) \in t \wedge (ch, w, c') \in \text{trcl } t$
 ⟨*proof*⟩

lemma *trcl-uncons-cases*: \llbracket
 $(c, e \# w, c') \in \text{trcl } S;$
 $\text{!!} ch . \llbracket (c, e, ch) \in S; (ch, w, c') \in \text{trcl } S \rrbracket \implies P$
 $\rrbracket \implies P$
 ⟨*proof*⟩

lemma *trcl-one-elim*: $(c, e, c') \in t \implies (c, [e], c') \in \text{trcl } t$
 ⟨*proof*⟩

lemma *trcl-unconsE*[*cases set, case-names split*]: \llbracket
 $(c, e \# w, c') \in \text{trcl } S;$
 $\text{!!} ch . \llbracket (c, e, ch) \in S; (ch, w, c') \in \text{trcl } S \rrbracket \implies P$
 $\rrbracket \implies P$
 ⟨*proof*⟩

lemma *trcl-pair-unconsE*[*cases set, case-names split*]: \llbracket
 $((s, c), e \# w, (s', c')) \in \text{trcl } S;$
 $\text{!!} sh \text{ ch} . \llbracket ((s, c), e, (sh, ch)) \in S; ((sh, ch), w, (s', c')) \in \text{trcl } S \rrbracket \implies P$
 $\rrbracket \implies P$
 ⟨*proof*⟩

lemma *trcl-concat*: $\text{!!} c . \llbracket (c, w1, c') \in \text{trcl } t; (c', w2, c'') \in \text{trcl } t \rrbracket$
 $\implies (c, w1 @ w2, c'') \in \text{trcl } t$
 ⟨*proof*⟩

lemma *trcl-unconcat*: $\text{!!} c . (c, w1 @ w2, c') \in \text{trcl } t$
 $\implies \exists ch . (c, w1, ch) \in \text{trcl } t \wedge (ch, w2, c') \in \text{trcl } t$
 ⟨*proof*⟩

4.2.1 Appending of elements to paths

lemma *trcl-rev-cons*: $\llbracket (c, w, ch) \in \text{trcl } T; (ch, e, c') \in T \rrbracket \implies (c, w @ [e], c') \in \text{trcl } T$
 ⟨*proof*⟩

lemma *trcl-rev-uncons*: $(c, w @ [e], c') \in \text{trcl } T$
 $\implies \exists ch . (c, w, ch) \in \text{trcl } T \wedge (ch, e, c') \in T$
 ⟨*proof*⟩

lemma *trcl-rev-induct*[*induct set, consumes 1, case-names empty snoc*]: $\text{!!} c' . \llbracket$
 $(c, w, c') \in \text{trcl } S;$
 $\text{!!} c . P \ c \ \llbracket c;$
 $\text{!!} c \ w \ c' \ e \ c'' . \llbracket (c, w, c') \in \text{trcl } S; (c', e, c'') \in S; P \ c \ w \ c' \rrbracket \implies P \ c \ (w @ [e]) \ c''$
 $\rrbracket \implies P \ c \ w \ c'$
 ⟨*proof*⟩

lemma *trcl-rev-cases*: $\text{!!} c \ c' . \llbracket$
 $(c, w, c') \in \text{trcl } S;$

```

[[w=[]; c=c']] ==> P;
!!ch e wh. [[w=wh@[e]; (c,wh,ch)∈trcl S; (ch,e,c')∈S ]] ==> P
]] ==> P
⟨proof⟩

```

```

lemma trcl-cons2: [[ (c,e,ch)∈T; (ch,f,c')∈T ]] ==> (c,[e,f],c')∈trcl T
⟨proof⟩

```

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 ⊆ B ==> trcl A ⊆ trcl B
⟨proof⟩

```

```

lemma trcl-inter-mono: x∈trcl (S∩R) ==> x∈trcl S x∈trcl (S∩R) ==> x∈trcl R
⟨proof⟩

```

4.2.4 Special lemmas for reasoning about states that are pairs

```

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

```

4.2.5 Invariants

```

lemma trcl-prop-trans[cases set, consumes 1, case-names empty steps]: [[
(c,w,c')∈trcl S;
[[c=c'; w=[]]] ==> P;
[[c∈Domain S; c'∈Range (Range S)]] ==> P
]] ==> P
⟨proof⟩

```

end

5 Thread Tracking

```

theory ThreadTracking
imports Main Multiset LTS Misc

```

begin

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

5.1 Semantic on multiset configuration

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

definition

$$gtr\ gtrs == \{ (\{ \#s\# \} + c, e, \{ \#s'\# \} + c') \mid s\ c\ e\ s'\ c' . ((s, c), e, (s', c')) \in gtrs \}$$

lemma *gtrI-s*: $((s, c), e, (s', c')) \in gtrs \implies (\{ \#s\# \} + c, e, \{ \#s'\# \} + c') \in gtr\ gtrs$
 $\langle proof \rangle$

lemma *gtrI*: $((s, c), w, (s', c')) \in trcl\ gtrs$
 $\implies (\{ \#s\# \} + c, w, \{ \#s'\# \} + c') \in trcl\ (gtr\ gtrs)$
 $\langle proof \rangle$

lemma *gtrE*: \llbracket
 $(c, e, c') \in gtr\ T;$
 $!!s\ ce\ s'\ ce' . \llbracket c = \{ \#s\# \} + ce; c' = \{ \#s'\# \} + ce'; ((s, ce), e, (s', ce')) \in T \rrbracket \implies P$
 $\rrbracket \implies P$
 $\langle proof \rangle$

lemma *gtr-empty-conf-s[simp]*:

$$\begin{aligned} & (\{ \# \}, w, c') \notin gtr\ S \\ & (c, w, \{ \# \}) \notin gtr\ S \end{aligned}$$

$$\langle proof \rangle$$

lemma *gtr-empty-conf1[simp]*: $((\{ \# \}, w, c') \in trcl\ (gtr\ S)) \iff (w = [] \wedge c' = \{ \# \})$
 $\langle proof \rangle$

lemma *gtr-empty-conf2[simp]*: $((c, w, \{ \# \}) \in trcl\ (gtr\ S)) \iff (w = [] \wedge c = \{ \# \})$
 $\langle proof \rangle$

lemma *gtr-find-thread*: \llbracket

$$\begin{aligned} & (c, e, c') \in gtr\ gtrs; \\ & !!s\ ce\ s'\ ce' . \llbracket c = \{ \#s\# \} + ce; c' = \{ \#s'\# \} + ce'; ((s, ce), e, (s', ce')) \in gtrs \rrbracket \implies P \end{aligned}$$

$$\rrbracket \implies P$$

$$\langle proof \rangle$$

lemma *gtr-step-cases[cases set, case-names loc other]*: \llbracket

$$\begin{aligned} & (\{ \#s\# \} + ce, e, c') \in gtr\ gtrs; \\ & !!s'\ ce' . \llbracket c' = \{ \#s'\# \} + ce'; ((s, ce), e, (s', ce')) \in gtrs \rrbracket \implies P; \\ & !!cc\ ss\ ss'\ ce' . \llbracket ce = \{ \#ss\# \} + cc; c' = \{ \#ss'\# \} + ce'; \\ & \quad ((ss, \{ \#s\# \} + cc), e, (ss', ce')) \in gtrs \rrbracket \implies P \end{aligned}$$

$$\rrbracket \implies P$$

$\langle proof \rangle$

lemma *gtr-rev-cases*[*cases set, case-names loc other*]: \llbracket
 $(c, e, \{\#s'\#\} + ce') \in gtr\ gtrs;$
 $!!s\ ce. \llbracket c = \{\#s\#\} + ce; ((s, ce), e, (s', ce')) \in gtrs \rrbracket \implies P;$
 $!!cc\ ss\ ss'\ ce. \llbracket c = \{\#ss\#\} + ce; ce' = \{\#ss'\#\} + cc;$
 $((ss, ce), e, (ss', \{\#s'\#\} + cc)) \in gtrs \rrbracket \implies P$
 $\rrbracket \implies P$
 $\langle proof \rangle$

5.2 Invariants

lemma *gtr-preserve-s*: \llbracket
 $(c, e, c') \in gtr\ T;$
 $P\ c;$
 $!!s\ c\ s'\ c'\ e. \llbracket P\ (\{\#s\#\} + c); ((s, c), e, (s', c')) \in T \rrbracket \implies P\ (\{\#s'\#\} + c')$
 $\rrbracket \implies P\ c'$
 $\langle proof \rangle$

lemma *gtr-preserve*: \llbracket
 $(c, w, c') \in trcl\ (gtr\ T);$
 $P\ c;$
 $!!s\ c\ s'\ c'\ e. \llbracket P\ (\{\#s\#\} + c); ((s, c), e, (s', c')) \in T \rrbracket \implies P\ (\{\#s'\#\} + c')$
 $\rrbracket \implies P\ c'$
 $\langle proof \rangle$

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

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

lemma (**in** *env-no-step*) *env-no-step*[*cases set, case-names csp*]: \llbracket
 $((s, c), w, (s', c')) \in trcl\ gtrs;$
 $!!\ csp. c' = csp + c \implies P$
 $\rrbracket \implies P$
 $\langle proof \rangle$

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

loc The thread made the step

spawn The thread was spawned by the step

env The thread was not involved in the step

lemma (in *env-no-step*) *rev-cases-p*[*cases set, case-names loc spawn env*]:

assumes *STEP*: $(c, e, \{s'\# \} + ce') \in gtr\ gtrs$ **and**

LOC: $!!s\ ce. \llbracket c = \{s\# \} + ce; ((s, ce), e, (s', ce')) \in gtrs \rrbracket \implies P$ **and**

SPAWN: $!!ss\ ss'\ ce\ csp.$

$\llbracket c = \{ss\# \} + ce; ce' = \{ss'\# \} + csp + ce;$
 $((ss, ce), e, (ss', \{s'\# \} + csp + ce)) \in gtrs \rrbracket$

$\implies P$ **and**

ENV: $!!ss\ ss'\ ce\ csp.$

$\llbracket c = \{ss\# \} + \{s'\# \} + ce; ce' = \{ss'\# \} + csp + ce;$
 $((ss, \{s'\# \} + ce), e, (ss', csp + (\{s'\# \} + ce))) \in gtrs \rrbracket$

$\implies P$

shows *P*

<proof>

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

$loc\ w == filter\ (\lambda e. case\ e\ of\ LOC\ a \Rightarrow True\ |\ ENV\ a \Rightarrow False)\ w$

definition

$env\ w == filter\ (\lambda e. case\ e\ of\ LOC\ a \Rightarrow False\ |\ ENV\ a \Rightarrow True)\ w$

definition

$le\text{-}rem\text{-}s\ e == case\ e\ of\ LOC\ a \Rightarrow a\ |\ ENV\ a \Rightarrow a$

Standard simplification lemmas

lemma $loc\text{-}env\text{-}simps[simp]$:

$loc\ [] = []$
 $env\ [] = []$
 $\langle proof \rangle$

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

$\langle proof \rangle$

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

$loc\ (a\#\#b) = (case\ a\ of\ LOC\ e \Rightarrow [a]\ |\ ENV\ e \Rightarrow []) @ loc\ b$
 $\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$env\ (a\#\#b) = (case\ a\ of\ LOC\ e \Rightarrow []\ |\ ENV\ e \Rightarrow [a]) @ env\ b$
 $\langle proof \rangle$

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

$\langle proof \rangle$

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

lemma $le\text{-}rem\text{-}simps\ [simp]$:

$le\text{-}rem\text{-}s\ (LOC\ a) = a$
 $le\text{-}rem\text{-}s\ (ENV\ a) = a$
 $\langle proof \rangle$

lemma $le\text{-}rem\text{-}id\text{-}simps[simp]$:

$le\text{-}rem\text{-}s \circ LOC = id$
 $le\text{-}rem\text{-}s \circ ENV = id$
 $\langle proof \rangle$

lemma $le\text{-}rem\text{-}id\text{-}map[simp]$:

$map\ le\text{-}rem\text{-}s\ (map\ LOC\ w) = w$
 $map\ le\text{-}rem\text{-}s\ (map\ ENV\ w) = w$
 $\langle proof \rangle$

lemma $env\text{-}map\text{-}env\ [simp]$: $env\ (map\ ENV\ w) = map\ ENV\ w$

$\langle proof \rangle$

lemma $env\text{-}map\text{-}loc\ [simp]$: $env\ (map\ LOC\ w) = []$

$\langle proof \rangle$
lemma *loc-map-env* [*simp*]: $loc (map ENV w) = []$
 $\langle proof \rangle$
lemma *loc-map-loc* [*simp*]: $loc (map LOC w) = map LOC w$
 $\langle proof \rangle$

5.4.2 Definition of the loc/env-semantics

types $'s\ el-conf = ('s \times 's\ multiset)$

inductive-set

$gtrp :: ('s\ el-conf, 'l)\ LTS \Rightarrow ('s\ el-conf, 'l\ el-step)\ LTS$

for S

where

$gtrp\text{-}loc: ((s,c), e, (s',c')) \in S \implies ((s,c), LOC\ e, (s',c')) \in gtrp\ S$

$| gtrp\text{-}env: ((s, \{ \#s\# \} + c), e, (s', \{ \#s\# \} + c')) \in S$

$\implies ((sl, \{ \#s\# \} + c), ENV\ e, (sl, \{ \#s\# \} + c')) \in gtrp\ S$

5.4.3 Relation between multiset- and loc/env-semantics

lemma *gtrp2gtr-s*:

$((s,c), e, (s',c')) \in gtrp\ T \implies (\{ \#s\# \} + c, le\text{-}rem\text{-}s\ e, \{ \#s'\# \} + c') \in gtr\ T$
 $\langle proof \rangle$

lemma *gtrp2gtr*:

$((s,c), w, (s',c')) \in trcl (gtrp\ T)$

$\implies (\{ \#s\# \} + c, map\ le\text{-}rem\text{-}s\ w, \{ \#s'\# \} + c') \in trcl (gtr\ T)$

$\langle proof \rangle$

lemma (*in env-no-step*) *gtr2gtrp-s*[*cases set, case-names gtrp*]:

assumes $A: (\{ \#s\# \} + c, e, c') \in gtr\ gtrs$

and CASE: $!!s'\ ce'\ ee. \llbracket c' = \{ \#s'\# \} + ce'; e = le\text{-}rem\text{-}s\ ee;$

$((s,c), ee, (s',ce')) \in gtrp\ gtrs \rrbracket$

$\implies P$

shows P

$\langle proof \rangle$

lemma (*in env-no-step*) *gtr2gtrp*[*cases set, case-names gtrp*]:

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

and CASE: $!!s'\ ce'\ ww. \llbracket c' = \{ \#s'\# \} + ce'; w = map\ le\text{-}rem\text{-}s\ ww;$

$((s,c), ww, (s',ce')) \in trcl (gtrp\ gtrs) \rrbracket$

$\implies P$

shows P

$\langle proof \rangle$

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^))∈T]]
                        ⇒ P ({#s'#}+c')
shows P ({#s'#}+c')
⟨proof⟩

```

```

lemma gtrp-preserve:
assumes A: ((s,c),w,(s',c^))∈trcl (gtrp T)
and INIT: P ({#s#}+c)
and PRES: !!s c s' c' e. [[P ({#s#}+c); ((s,c),e,(s',c^))∈T]]
                        ⇒ P ({#s'#}+c')
shows P ({#s'#}+c')
⟨proof⟩

```

end

6 Flowgraphs

```

theory Flowgraph
imports Main Misc
begin

```

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

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

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

6.1 Definitions

```

datatype ('p,'ba) edgeAnnot = Base 'ba | Call 'p | Spawn 'p
types ('n,'p,'ba) edge = ('n × ('p,'ba) edgeAnnot × 'n)

```

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

definition

initialproc fg p == $p = \text{main fg} \vee (\exists u v. (u, \text{Spawn } p, v) \in \text{edges fg})$

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

locale *flowgraph* =

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

— Edges are inside procedures only

assumes *edges-part*: $(u, a, v) \in \text{edges fg} \implies \text{proc-of fg } u = \text{proc-of fg } v$

— The entry point of a procedure must be in that procedure

assumes *entry-valid[simp]*: $\text{proc-of fg } (\text{entry fg } p) = p$

— The return point of a procedure must be in that procedure

assumes *return-valid[simp]*: $\text{proc-of fg } (\text{return fg } p) = p$

— Initial procedures do not synchronize on any monitors

assumes *initial-no-mon[simp]*: $\text{initialproc fg } p \implies \text{mon fg } p = \{\}$

6.2 Basic properties

lemma (**in** *flowgraph*) *spawn-no-mon[simp]*:
 $(u, \text{Spawn } p, v) \in \text{edges fg} \implies \text{mon fg } p = \{\}$
 $\langle \text{proof} \rangle$

lemma (**in** *flowgraph*) *main-no-mon[simp]*: $\text{mon fg } (\text{main fg}) = \{\}$
 $\langle \text{proof} \rangle$

lemma (**in** *flowgraph*) *entry-return-same-proc[simp]*:
 $\text{entry fg } p = \text{return fg } p' \implies p = p'$
 $\langle \text{proof} \rangle$

lemma (**in** *flowgraph*) *entry-entry-same-proc[simp]*:
 $\text{entry fg } p = \text{entry fg } p' \implies p = p'$
 $\langle \text{proof} \rangle$

lemma (**in** *flowgraph*) *return-return-same-proc[simp]*:
 $\text{return fg } p = \text{return fg } p' \implies p = p'$
 $\langle \text{proof} \rangle$

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

$$\begin{aligned} \textit{isolated-ret fg } p &== \\ &(\forall u l. \neg(u, l, \textit{return fg } p) \in \textit{edges fg}) \wedge \textit{entry fg } p \neq \textit{return fg } p \end{aligned}$$

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*: $\textit{initialproc fg } p \implies \textit{entry fg } p \neq \textit{return fg } p$

— The only outgoing edges of initial procedures' entry nodes are call edges to procedures with isolated return node

assumes *initial-call-no-ret*: $\llbracket \textit{initialproc fg } p; (\textit{entry fg } p, l, v) \in \textit{edges fg} \rrbracket \implies \exists p'. l = \textit{Call } p' \wedge \textit{isolated-ret fg } p'$

6.4 Example Flowgraph

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

definition

$$\begin{aligned} \textit{example-fg} &== \llbracket \\ &\textit{edges} = \{((0::\textit{nat}, 0::\textit{nat}), \textit{Call } 1, (0, 1)), ((1, 0), \textit{Spawn } 0, (1, 0)), \\ &\quad ((1, 0), \textit{Call } 0, (1, 0))\}, \\ &\textit{main} = 0, \\ &\textit{entry} = \lambda p. (p, 0), \\ &\textit{return} = \lambda p. (p, 1), \\ &\textit{mon} = \lambda p. \textit{if } p=1 \textit{ then } \{0\} \textit{ else } \{\}, \\ &\textit{proc-of} = \lambda (p, x). p \rrbracket \end{aligned}$$

lemma *exists-eflowgraph*: *eflowgraph example-fg*
<proof>

end

7 Operational Semantics

theory *Semantics*

imports *Main Flowgraph 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.

types

$'n \text{ conf} = ('n \text{ list}) \text{ multiset}$

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 $('p, 'ba) \text{ label} = LBase 'ba \mid LCall 'p \mid LRet \mid LSpawn 'p$

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

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

definition

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

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

definition

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

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

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

consts $mon-e :: ('b, 'c, 'd, 'a, 'e) \text{ flowgraph-rec-scheme} \Rightarrow ('c, 'f) \text{ label} \Rightarrow 'a \text{ set}$

primrec

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

lemma *mon-e-def*: $mon\text{-}e\ fg\ e == case\ e\ of\ (LCall\ p) \Rightarrow mon\ fg\ p \mid - \Rightarrow \{\}$
<proof>

definition

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

lemma *mon-s-alt*: $mon\text{-}s\ fg\ s == \bigcup\ mon\ fg\ \text{'proc-of fg ' set } s$
<proof>

lemma *mon-c-alt*: $mon\text{-}c\ fg\ c == \bigcup\ mon\text{-}s\ fg\ \text{'set-of } c$
<proof>

lemma *mon-w-alt*: $mon\text{-}w\ fg\ w == \bigcup\ mon\text{-}e\ fg\ \text{'set } w$
<proof>

lemma *mon-sI*: $\llbracket n \in set\ s; m \in mon\text{-}n\ fg\ n \rrbracket \Longrightarrow m \in mon\text{-}s\ fg\ s$
<proof>

lemma *mon-sD*: $m \in mon\text{-}s\ fg\ s \Longrightarrow \exists n \in set\ s. m \in mon\text{-}n\ fg\ n$
<proof>

lemma *mon-n-same-proc*:

$proc\text{-of}\ fg\ n = proc\text{-of}\ fg\ n' \Longrightarrow mon\text{-}n\ fg\ n = mon\text{-}n\ fg\ n'$
<proof>

lemma *mon-s-same-proc*:

$proc\text{-of}\ fg\ \text{'set } s = proc\text{-of}\ fg\ \text{'set } s' \Longrightarrow mon\text{-}s\ fg\ s = mon\text{-}s\ fg\ s'$
<proof>

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

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

lemma *mon-c-single[simp]*: $mon\text{-}c\ fg\ \{\#s\# \} = mon\text{-}s\ fg\ s$
<proof>

lemma *mon-s-single[simp]*: $mon\text{-}s\ fg\ [n] = mon\text{-}n\ fg\ n$
<proof>

lemma *mon-s-empty[simp]*: $mon\text{-}s\ fg\ [] = \{\}$
<proof>

lemma *mon-c-empty[simp]*: $mon\text{-}c\ fg\ \{\#\} = \{\}$
<proof>

lemma *mon-s-unconc*: $mon\text{-}s\ fg\ (a@b) = mon\text{-}s\ fg\ a \cup mon\text{-}s\ fg\ b$
<proof>

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

lemma *mon-c-unconc*: $\text{mon-c fg } (a+b) = \text{mon-c fg } a \cup \text{mon-c fg } b$
 ⟨proof⟩

lemma *mon-cI*: $\llbracket s:\#c; m \in \text{mon-s fg } s \rrbracket \implies m \in \text{mon-c fg } c$
 ⟨proof⟩

lemma *mon-cD*: $\llbracket m \in \text{mon-c fg } c \rrbracket \implies \exists s. s:\#c \wedge m \in \text{mon-s fg } s$
 ⟨proof⟩

lemma *mon-s-mono*: $\text{set } s \subseteq \text{set } s' \implies \text{mon-s fg } s \subseteq \text{mon-s fg } s'$
 ⟨proof⟩

lemma *mon-c-mono*: $c \leq \#c' \implies \text{mon-c fg } c \subseteq \text{mon-c fg } c'$
 ⟨proof⟩

lemma *mon-w-empty[simp]*: $\text{mon-w fg } [] = \{\}$
 ⟨proof⟩

lemma *mon-w-single[simp]*: $\text{mon-w fg } [e] = \text{mon-e fg } e$
 ⟨proof⟩

lemma *mon-w-unconc*: $\text{mon-w fg } (wa@wb) = \text{mon-w fg } wa \cup \text{mon-w fg } wb$
 ⟨proof⟩

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

lemma *mon-w-ileq*: $w \preceq w' \implies \text{mon-w fg } w \subseteq \text{mon-w fg } w'$
 ⟨proof⟩

7.3 Valid configurations

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

definition

$$\text{valid fg } c == \forall s s'. \{\#s\} + \{\#s'\} \leq \#c \longrightarrow \text{mon-s fg } s \cap \text{mon-s fg } s' = \{\}$$

lemma *valid-empty[simp, intro!]*: $\text{valid fg } \{\#\}$
 ⟨proof⟩

lemma *valid-single[simp, intro!]*: $\text{valid fg } \{\#s\}$
 ⟨proof⟩

lemma *valid-split1*:

$$\text{valid fg } (c+c') \implies \text{valid fg } c \wedge \text{valid fg } c' \wedge \text{mon-c fg } c \cap \text{mon-c fg } c' = \{\}$$
 ⟨proof⟩

lemma *valid-split2*:

$$\llbracket \text{valid fg } c; \text{valid fg } c'; \text{mon-c fg } c \cap \text{mon-c fg } c' = \{\} \rrbracket \implies \text{valid fg } (c+c')$$
 ⟨proof⟩

lemma *valid-unconc*:

$$\text{valid fg } (c+c') \iff (\text{valid fg } c \wedge \text{valid fg } c' \wedge \text{mon-c fg } c \cap \text{mon-c fg } c' = \{\})$$
 ⟨proof⟩

lemma *valid-no-mon*: $\text{mon-c fg } c = \{\} \implies \text{valid fg } c$

<proof>

7.4 Configurations at control points

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

consts *atU-s* :: 'n set \Rightarrow 'n list \Rightarrow bool

primrec

atU-s U [] = False
atU-s U (u#r) = ($u \in U$)

lemma *atU-s-decomp[simp]*: *atU-s U (s@s')* = (*atU-s U s* \vee ($s = [] \wedge$ *atU-s U s'*))

<proof>

definition

atU U c == $\exists s. s : \#c \wedge$ *atU-s U s*

lemma *atU-fmt*: \llbracket *atU U c*; $!!ui r. \llbracket ui \# r : \# c; ui \in U \rrbracket \implies P \rrbracket \implies P$

<proof>

lemma *atU-union-cases[case-names left right, consumes 1]*: \llbracket

atU U (c1+c2);
atU U c1 $\implies P$;
atU U c2 $\implies P$

$\rrbracket \implies P$

<proof>

lemma *atU-add*: *atU U c* \implies *atU U (c+ce)* \wedge *atU U (ce+c)*

<proof>

lemma *atU-union[simp]*: *atU U (c1+c2)* = (*atU U c1* \vee *atU U c2*)

<proof>

lemma *atU-empty[simp]*: \neg *atU U {#}*

<proof>

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

<proof>

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

<proof>

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

<proof>

definition

atUV U V c == $\exists su sv. \{ \#su\# \} + \{ \#sv\# \} \leq \# c \wedge$ *atU-s U su* \wedge *atU-s V sv*

lemma *atUV-empty[simp]*: \neg *atUV U V {#}*

<proof>

lemma *atUV-single[simp]*: \neg *atUV U V {#s#}*

<proof>

lemma *atUV-union[simp]*:

$$\begin{array}{l}
atUV \ U \ V \ (c1+c2) \longleftrightarrow \\
(\\
\quad (atUV \ U \ V \ c1) \vee \\
\quad (atUV \ U \ V \ c2) \vee \\
\quad (atU \ U \ c1 \wedge atU \ V \ c2) \vee \\
\quad (atU \ V \ c1 \wedge atU \ U \ c2) \\
) \\
\langle proof \rangle
\end{array}$$

lemma *atUV-union-cases*[*case-names left right lr rl, consumes 1*]: \llbracket

$$\begin{array}{l}
atUV \ U \ V \ (c1+c2); \\
atUV \ U \ V \ c1 \implies P; \\
atUV \ U \ V \ c2 \implies P; \\
\llbracket atU \ U \ c1; atU \ V \ c2 \rrbracket \implies P; \\
\llbracket atU \ V \ c1; atU \ U \ c2 \rrbracket \implies P \\
\rrbracket \implies P \\
\langle proof \rangle
\end{array}$$

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

$$\begin{array}{l}
refpoint :: ('n, 'p, 'ba, 'm, 'more) flowgraph-rec-scheme \implies \\
\quad ('n \ conf \times ('p, 'ba) \ label \times 'n \ conf) \ set
\end{array}$$

for *fg*

where

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

$$\begin{array}{l}
refpoint-base: \llbracket (u, Base \ a, v) \in edges \ fg; \ valid \ fg \ (\{ \#u\#r\# \} + c) \rrbracket \\
\implies (\{ \#u\#r\# \} + c, LBase \ a, \{ \#v\#r\# \} + c) \in refpoint \ fg \ |
\end{array}$$

— 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.

$$\begin{array}{l}
refpoint-call: \llbracket (u, Call \ p, v) \in edges \ fg; \ valid \ fg \ (\{ \#u\#r\# \} + c); \\
\quad mon \ fg \ p \cap mon-c \ fg \ c = \{ \} \rrbracket \\
\implies (\{ \#u\#r\# \} + c, LCall \ p, \{ \#entry \ fg \ p\#v\#r\# \} + c) \in refpoint \ fg \ |
\end{array}$$

— 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.

$$\begin{array}{l}
refpoint-ret: \llbracket \ valid \ fg \ (\{ \#return \ fg \ p\#r\# \} + c) \rrbracket \\
\implies (\{ \#return \ fg \ p\#r\# \} + c, LRet, (\{ \#r\# \} + c)) \in refpoint \ fg \ |
\end{array}$$

— 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.

$$\begin{aligned} \text{refpoint-spawn: } & \llbracket (u, \text{Spawn } p, v) \in \text{edges } fg; \text{ valid } fg \ (\{\#u\#r\# \} + c) \rrbracket \\ & \implies (\{\#u\#r\# \} + c, \text{LSpawn } p, \{\#v\#r\# \} + \#[\text{entry } fg \ p] \# + c) \in \text{refpoint } fg \end{aligned}$$

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

$$\begin{aligned} \text{trss} :: & ('n, 'p, 'ba, 'm, 'more) \text{ flowgraph-rec-scheme} \implies \\ & (('n \text{ list} * 'n \text{ conf}) * ('p, 'ba) \text{ label} * ('n \text{ list} * 'n \text{ conf})) \text{ set} \end{aligned}$$

for fg

where

$$\begin{aligned} \text{trss-base: } & \llbracket (u, \text{Base } a, v) \in \text{edges } fg \rrbracket \implies \\ & ((u\#r, c), \text{LBase } a, (v\#r, c)) \in \text{trss } fg \\ | \text{trss-call: } & \llbracket (u, \text{Call } p, v) \in \text{edges } fg; \text{ mon } fg \ p \cap \text{mon-c } fg \ c = \{\} \rrbracket \implies \\ & ((u\#r, c), \text{LCall } p, ((\text{entry } fg \ p)\#v\#r, c)) \in \text{trss } fg \\ | \text{trss-ret: } & \llbracket ((\text{return } fg \ p)\#r), c, \text{LRet}, (r, c) \rrbracket \in \text{trss } fg \\ | \text{trss-spawn: } & \llbracket (u, \text{Spawn } p, v) \in \text{edges } fg \rrbracket \implies \\ & ((u\#r, c), \text{LSpawn } p, (v\#r, \#[\text{entry } fg \ p] \# + c)) \in \text{trss } fg \end{aligned}$$

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

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

— We also generate the loc/env-semantics

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

7.6 Basic properties

7.6.1 Validity

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

$$\llbracket \text{valid } fg \ (\{\#s\# \} + c); ((s, c), e, (s', c')) \in \text{trss } fg \rrbracket \implies \text{valid } fg \ (\{\#s'\# \} + c')$$

<proof>

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

$$\llbracket ((s, c), w, (s', c')) \in \text{trcl} \ (\text{trss } fg); \text{valid } fg \ (\{\#s\# \} + c) \rrbracket \implies \text{valid } fg \ (\{\#s'\# \} + c')$$

<proof>

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

$$\llbracket (c, e, c') \in \text{tr } fg; \text{valid } fg \ c \rrbracket \implies \text{valid } fg \ c'$$

<proof>

lemma (in *flowgraph*) *tr-valid-preserve*:

$\llbracket (c,w,c') \in \text{trcl } (tr \text{ fg}); \text{ valid fg } c \rrbracket \implies \text{ valid fg } c'$
 $\langle \text{proof} \rangle$

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

$\llbracket ((s,c),e,(s',c')) \in \text{trp fg}; \text{ valid fg } (\{ \#s\# \} + c) \rrbracket \implies \text{ valid fg } (\{ \#s'\# \} + c')$
 $\langle \text{proof} \rangle$

lemma (in *flowgraph*) *trp-valid-preserve*:

$\llbracket ((s,c),w,(s',c')) \in \text{trcl } (trp \text{ fg}); \text{ valid fg } (\{ \#s\# \} + c) \rrbracket \implies \text{ valid fg } (\{ \#s'\# \} + c')$
 $\langle \text{proof} \rangle$

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*: $\text{ valid fg } c \implies ((c,e,c') \in \text{refpoint fg}) \longleftrightarrow ((c,e,c') \in \text{tr fg})$
 $\langle \text{proof} \rangle$

lemma (in *flowgraph*) *refpoint-eq*:

$\text{ valid fg } c \implies ((c,w,c') \in \text{trcl } (\text{refpoint fg})) \longleftrightarrow ((c,w,c') \in \text{trcl } (tr \text{ fg}))$
 $\langle \text{proof} \rangle$

7.6.3 Case distinctions

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

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

$\rrbracket \implies P$

$\langle \text{proof} \rangle$

lemma *trss-c-fmt-s*: $\llbracket ((s,c),e,(s',c')) \in \text{trss fg} \rrbracket$

$\implies \exists \text{ csp}. c' = \text{csp} + c \wedge$
 $(\text{csp} = \{ \# \} \vee (\exists p. e = \text{LSpawn } p \wedge \text{csp} = \{ \#[\text{entry fg } p] \# \}))$

$\langle \text{proof} \rangle$

lemma (in *flowgraph*) *trss-c'-split-s*: \llbracket

$((s,c),e,(s',c')) \in \text{trss fg};$
 $!!\text{csp}. \llbracket c' = \text{csp} + c; \text{mon-c fg } \text{csp} = \{ \} \rrbracket \implies P$

$\rrbracket \implies P$

$\langle \text{proof} \rangle$

lemma *trss-c-cases*[*cases set, case-names c-case*]: $!!s \ c. \llbracket$

$((s,c),w,(s',c')) \in \text{trcl } (trss \text{ fg});$
 $!!\text{csp}. \llbracket c' = \text{csp} + c; !!s. s : \# \text{csp} \implies \exists p \ u \ v. s = [\text{entry fg } p] \wedge$
 $\quad (u, \text{Spawn } p, v) \in \text{edges fg} \wedge$
 $\quad \text{initialproc fg } p \rrbracket$

$\implies P$

$\rrbracket \implies P$

$\langle \text{proof} \rangle$

lemma (in *flowgraph*) *c-of-initial-no-mon*:
assumes $A: \forall s. s:\#csp \implies \exists p. s=[\text{entry } fg \ p] \wedge \text{initialproc } fg \ p$
shows $\text{mon-c } fg \ csp = \{\}$
 $\langle \text{proof} \rangle$

lemma (in *flowgraph*) *trss-c-no-mon-s*:
assumes $A: ((s,c),e,(s',c')) \in \text{trss } fg$
shows $\text{mon-c } fg \ c' = \text{mon-c } fg \ c$
 $\langle \text{proof} \rangle$

corollary (in *flowgraph*) *trss-c-no-mon*:
 $((s,c),w,(s',c')) \in \text{trcl } (\text{trss } fg) \implies \text{mon-c } fg \ c' = \text{mon-c } fg \ c$
 $\langle \text{proof} \rangle$

lemma (in *flowgraph*) *trss-spawn-no-mon-step[simp]*:
 $((s,c),L\text{spawn } p, (s',c')) \in \text{trss } fg \implies \text{mon } fg \ p = \{\}$
 $\langle \text{proof} \rangle$

lemma *trss-no-empty-s[simp]*: $(([],c),e,(s',c')) \in \text{trss } fg = \text{False}$
 $\langle \text{proof} \rangle$

lemma *trss-no-empty[simp]*:
assumes $A: (([],c),w,(s',c')) \in \text{trcl } (\text{trss } fg)$
shows $w=[] \wedge s'=[] \wedge c=c'$
 $\langle \text{proof} \rangle$

lemma *trs-step-cases[cases set, case-names NO-SPAWN SPAWN]*:
assumes $A: (c,e,c') \in \text{tr } fg$
assumes $A\text{-NO-SPAWN}: \forall s \ ce \ s' \ csp. \llbracket$
 $((s,ce),e,(s',ce)) \in \text{trss } fg;$
 $c=\{\#s\# \}+ce; \ c'=\{\#s'\# \}+ce$
 $\rrbracket \implies P$
assumes $A\text{-SPAWN}: \forall s \ ce \ s' \ p. \llbracket$
 $((s,ce),L\text{spawn } p,(s',\{\#[\text{entry } fg \ p]\# \}+ce)) \in \text{trss } fg;$
 $c=\{\#s\# \}+ce;$
 $c'=\{\#s'\# \}+\{\#[\text{entry } fg \ p]\# \}+ce;$
 $e=L\text{spawn } p$
 $\rrbracket \implies P$
shows P
 $\langle \text{proof} \rangle$

7.7 Advanced properties

7.7.1 Stack composition / decomposition

lemma *trss-stack-comp-s*:

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

<proof>

lemma *trss-stack-comp*:

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

<proof>

lemma *trss-stack-decomp-s*: $\llbracket ((s@r,c),e,(s',c')) \in \text{trss fg}; s \neq [] \rrbracket$

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

<proof>

lemma *trss-find-return*: \llbracket

$$\begin{aligned} & ((s@r,c),w,(r,c')) \in \text{trcl} (\text{trss fg}); \\ & !!wa \text{ } wb \text{ } ch. \llbracket w = wa@wb; ((s,c),wa,([],ch)) \in \text{trcl} (\text{trss fg}); \\ & \quad ((r,ch),wb,(r,c')) \in \text{trcl} (\text{trss fg}) \rrbracket \implies P \end{aligned}$$

$$\rrbracket \implies P$$

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

<proof>

lemma *trss-return-cases*[*cases set*]: $!!u \text{ } r \text{ } c. \llbracket$

$$\begin{aligned} & ((u\#r,c),w,(r',c')) \in \text{trcl} (\text{trss fg}); \\ & !! s' \text{ } u'. \llbracket r' = s'@u'\#r; (([u],c),w,(s'@[u],c')) \in \text{trcl} (\text{trss fg}) \rrbracket \implies P; \\ & !! wa \text{ } wb \text{ } ch. \llbracket w = wa@wb; (([u],c),wa,([],ch)) \in \text{trcl} (\text{trss fg}); \\ & \quad ((r,ch),wb,(r',c')) \in \text{trcl} (\text{trss fg}) \rrbracket \implies P \end{aligned}$$

$$\rrbracket \implies P$$

<proof>

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

$$!!v \text{ } r' \text{ } c'. \llbracket (([sp],c),w,(v\#r',c')) \in \text{trcl} (\text{trss fg}); r' \neq [] \rrbracket$$

$$\implies \exists rh \text{ } ch \text{ } p \text{ } wa \text{ } wb.$$

$$w = wa@(LCall \text{ } p)\#wb \wedge$$

$$\text{proc-of fg } v = p \wedge$$

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

$$((rh,ch),LCall \text{ } p,((\text{entry fg } p)\#r',ch)) \in \text{trss fg} \wedge$$

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

<proof>

lemma (in *flowgraph*) *trss-find-call'*:

$$\text{assumes } A: (([sp],c),w,(\text{return fg } p\#[u],c')) \in \text{trcl} (\text{trss fg})$$

$$\text{and } EX: !!uh \text{ } ch \text{ } wa \text{ } wb. \llbracket$$

$$w = wa@(LCall \text{ } p)\#wb;$$

$$(([sp],c),wa,([uh],ch)) \in \text{trcl} (\text{trss fg});$$

$$(([uh],ch),LCall \text{ } p,((\text{entry fg } p)\#[u],ch)) \in \text{trss fg};$$

$$(uh, Call \text{ } p, u') \in \text{edges fg};$$

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

$\square \implies P$
shows P
 <proof>

lemma (in *flowgraph*) *trss-bot-proc-const*:
 $!!s' u' c'. ((s@[u],c),w,(s'@[u'],c')) \in \text{trcl } (trss \text{ fg})$
 $\implies \text{proc-of fg } u = \text{proc-of fg } u'$
 <proof>

lemma (in *flowgraph*) *trss-er-path-proc-const*:
 $(([\text{entry fg } p],c),w,([\text{return fg } q],c')) \in \text{trcl } (trss \text{ fg}) \implies p=q$
 <proof>

lemma *trss-2empty-to-2return*: $\llbracket ((s,c),w,([\],c')) \in \text{trcl } (trss \text{ fg}); s \neq [\] \rrbracket \implies$
 $\exists w' p. w=w'@[LRet] \wedge ((s,c),w',([\text{return fg } p],c')) \in \text{trcl } (trss \text{ fg})$
 <proof>

lemma *trss-2return-to-2empty*: $\llbracket ((s,c),w,([\text{return fg } p],c')) \in \text{trcl } (trss \text{ fg}) \rrbracket$
 $\implies ((s,c),w@[LRet],([\],c')) \in \text{trcl } (trss \text{ fg})$
 <proof>

7.7.2 Adding threads

lemma *trss-env-increasing-s*: $((s,c),e,(s',c')) \in trss \text{ fg} \implies c \leq \#c'$
 <proof>

lemma *trss-env-increasing*: $((s,c),w,(s',c')) \in \text{trcl } (trss \text{ fg}) \implies c \leq \#c'$
 <proof>

7.7.3 Conversion between environment and monitor restrictions

lemma *trss-mon-e-no-ctx*:
 $((s,c),e,(s',c')) \in trss \text{ fg} \implies \text{mon-e fg } e \cap \text{mon-c fg } c = \{\}$
 <proof>

lemma (in *flowgraph*) *trss-mon-w-no-ctx*:
 $((s,c),w,(s',c')) \in \text{trcl } (trss \text{ fg}) \implies \text{mon-w fg } w \cap \text{mon-c fg } c = \{\}$
 <proof>

lemma (in *flowgraph*) *trss-modify-context-s*:
 $!!cn. \llbracket ((s,c),e,(s',c')) \in trss \text{ fg}; \text{mon-e fg } e \cap \text{mon-c fg } cn = \{\} \rrbracket$
 $\implies \exists csp. c'=csp+c \wedge \text{mon-c fg } csp = \{\} \wedge ((s,cn),e,(s',csp+cn)) \in trss \text{ fg}$
 <proof>

lemma (in *flowgraph*) *trss-modify-context[rule-format]*:
 $\llbracket ((s,c),w,(s',c')) \in \text{trcl } (trss \text{ fg}) \rrbracket$
 $\implies \forall cn. \text{mon-w fg } w \cap \text{mon-c fg } cn = \{\}$
 $\longrightarrow (\exists csp. c'=csp+c \wedge \text{mon-c fg } csp = \{\} \wedge$
 $\quad ((s,cn),w,(s',csp+cn)) \in \text{trcl } (trss \text{ fg}))$
 <proof>

lemma *trss-add-context-s*:
 $\llbracket ((s,c),e,(s',c')) \in trss \text{ fg}; \text{mon-e fg } e \cap \text{mon-c fg } ce = \{\} \rrbracket$

$\implies ((s, c+ce), e, (s', c'+ce)) \in trss\ fg$
 ⟨proof⟩

lemma *trss-add-context*:

$\llbracket ((s, c), w, (s', c')) \in trcl\ (trss\ fg); mon-w\ fg\ w \cap mon-c\ fg\ ce = \{\} \rrbracket$
 $\implies ((s, c+ce), w, (s', c'+ce)) \in trcl\ (trss\ fg)$
 ⟨proof⟩

lemma *trss-drop-context-s*: $\llbracket ((s, c+ce), e, (s', c'+ce)) \in trss\ fg \rrbracket$

$\implies ((s, c), e, (s', c')) \in trss\ fg \wedge mon-e\ fg\ e \cap mon-c\ fg\ ce = \{\}$
 ⟨proof⟩

lemma *trss-drop-context*: $\forall s\ c. \llbracket ((s, c+ce), w, (s', c'+ce)) \in trcl\ (trss\ fg) \rrbracket$

$\implies ((s, c), w, (s', c')) \in trcl\ (trss\ fg) \wedge mon-w\ fg\ w \cap mon-c\ fg\ ce = \{\}$
 ⟨proof⟩

lemma *trss-xchange-context-s*:

assumes $A: ((s, c), e, (s', csp+c)) \in trss\ fg$
and $M: mon-c\ fg\ cn \subseteq mon-c\ fg\ c$
shows $((s, cn), e, (s', csp+cn)) \in trss\ fg$
 ⟨proof⟩

lemma *trss-xchange-context*:

assumes $A: ((s, c), w, (s', csp+c)) \in trcl\ (trss\ fg)$
and $M: mon-c\ fg\ cn \subseteq mon-c\ fg\ c$
shows $((s, cn), w, (s', csp+cn)) \in trcl\ (trss\ fg)$
 ⟨proof⟩

lemma *trss-drop-all-context-s*[cases set, case-names dropped]:

assumes $A: ((s, c), e, (s', c')) \in trss\ fg$
and $C: \forall csp. \llbracket c' = csp+c; ((s, \{\#\}), e, (s', csp)) \in trss\ fg \rrbracket \implies P$
shows P
 ⟨proof⟩

lemma *trss-drop-all-context*[cases set, case-names dropped]:

assumes $A: ((s, c), w, (s', c')) \in trcl\ (trss\ fg)$
and $C: \forall csp. \llbracket c' = csp+c; ((s, \{\#\}), w, (s', csp)) \in trcl\ (trss\ fg) \rrbracket \implies P$
shows P
 ⟨proof⟩

lemma *tr-add-context-s*:

$\llbracket (c, e, c') \in tr\ fg; mon-e\ fg\ e \cap mon-c\ fg\ ce = \{\} \rrbracket \implies (c+ce, e, c'+ce) \in tr\ fg$
 ⟨proof⟩

lemma *tr-add-context*:

$\llbracket (c, w, c') \in trcl\ (tr\ fg); mon-w\ fg\ w \cap mon-c\ fg\ ce = \{\} \rrbracket$
 $\implies (c+ce, w, c'+ce) \in trcl\ (tr\ fg)$
 ⟨proof⟩

end

8 Normalized Paths

theory *Normalization*

imports *Main ThreadTracking Semantics ConsInterleave*

begin

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

8.1 Semantic properties of restricted flowgraphs

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

— Procedures with isolated return nodes will never return

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

isolated-ret fg p;

proc-of fg u = p;

$u \neq \text{return fg } p$;

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

] \Rightarrow *False*

<proof>

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

[$(([\text{entry fg } p],c),e,(s',c')) \in \text{trss fg}$; *initialproc fg p*]

$\Rightarrow \exists p'. e = \text{LCall } p' \wedge \text{isolated-ret fg } p'$

<proof>

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

assumes $A: w \neq []$ *initialproc fg p*

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

shows *False*

<proof>

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

assumes $A: w \neq []$

initialproc fg p

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

$\text{length } r' \leq 1$

shows *False*

<proof>

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 path² 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).

inductive-set

$$ntrs :: ('n, 'p, 'ba, 'm, 'more) \textit{flowgraph-rec-scheme} \Rightarrow \\ (('n \textit{ list} \times 'n \textit{ conf}) \times ('p, 'ba) \textit{ label list} \times ('n \textit{ list} \times 'n \textit{ conf})) \textit{ set}$$

for fg

where

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

$$ntrs\text{-}step: \begin{aligned} & \llbracket ((u\#r, ce), LCall\ p, (entry\ fg\ p\ \# \ u' \ \# \ r, ce)) \in trss\ fg; \\ & \quad (([entry\ fg\ p], ce), w, ([v], ce')) \in trcl\ (trss\ fg) \rrbracket \Longrightarrow \\ & ((u\#r, ce), LCall\ p\ \# \ w, (v\ \# \ u'\ \# \ r, ce')) \in ntrs\ fg \end{aligned}$$

abbreviation ntr where $ntr\ fg == gtr\ (ntrs\ fg)$

abbreviation $ntrp$ where $ntrp\ fg == gtrp\ (ntrs\ fg)$

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

(proof)

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

²Same-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.

<proof>

lemma *ntrs-is-trss*: $((s,c),w,(s',c')) \in \text{trcl } (ntrs \text{ fg})$
 $\implies ((s,c),\text{foldl } (op \ @) \ [] \ w,(s',c')) \in \text{trcl } (trss \text{ fg})$
<proof>

lemma *ntr-is-tr-s*: $(c,w,c') \in \text{ntr fg} \implies (c,w,c') \in \text{trcl } (tr \text{ fg})$
<proof>

lemma *ntr-is-tr*: $(c,ww,c') \in \text{trcl } (ntr \text{ fg}) \implies (c,\text{foldl } (op \ @) \ [] \ ww,c') \in \text{trcl } (tr \text{ fg})$
<proof>

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

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

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

lemma (in *eflowgraph*) *ntr-sl-move-left*: $!!ce \ u \ r \ w \ r' \ ce'$.
 $\llbracket (\{\#[\text{entry fg } p]\#\},ww,\{\# \ u\#r \ \#\}+ce) \in \text{trcl } (ntr \text{ fg});$
 $(([u],ce),w,(r',ce')) \in \text{trcl } (trss \ \text{fg});$
initialproc fg p;
length r' \leq 1; $w \neq []$
 $\rrbracket \implies \exists ww'. (\{\#[\text{entry fg } p]\#\}, ww',\{\# \ r'@r \ \#\}+ce') \in \text{trcl } (ntr \ \text{fg})$
<proof>

Finally we can prove: *Any reachable configuration can also be reached by a normalized path.* With *eflowgraph.ntr-sl-move-left* we can easily show this

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

lemma (in *eflowgraph*) *normalize*: \llbracket
 $(cstart, w, c') \in trcl (tr fg);$
 $cstart = \{\# [entry fg p] \#\};$
 $initialproc fg p \rrbracket$
 $\implies \exists w'. (\{\# [entry fg p] \#\}, w', c') \in trcl (ntr fg)$
— The lemma is shown by induction on the reaching path
 $\langle proof \rangle$

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

theorem (in *eflowgraph*) *ntr-repr*:
 $(\exists w. (\{\#[entry fg (main fg)]\#\}, w, c) \in trcl (tr fg))$
 $\iff (\exists w. (\{\#[entry fg (main fg)]\#\}, w, c) \in trcl (ntr fg))$
 $\langle proof \rangle$

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*]: \llbracket
 $((s, c), w, (s', c')) \in ntrs fg;$
 $!!csp. \llbracket c' = csp + c; !!s. s : \#csp \implies \exists p u v. s = [entry fg p] \wedge$
 $(u, Spawn p, v) \in edges fg \wedge$
 $initialproc fg p$
 $\rrbracket \implies P$
 $\rrbracket \implies P$
 $\langle proof \rangle$

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

8.4.1 Validity

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

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

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

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

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

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

assumes *A*: $((s,c),e,(s',c')) \in \text{ntrp } fg$

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

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

<proof>

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

assumes *A*: $((s,c),e,(s',c')) \in \text{trcl } (\text{ntrp } fg)$

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

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

<proof>

8.4.2 Monitors

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

definition

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

definition

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

definition

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

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

<proof>

lemma *mon-ww-uncons[simp]*:

mon-ww fg (*ee* $\#$ *ww*) = *mon-w fg ee* \cup *mon-ww fg ww*

<proof>

lemma *mon-ww-unconc*:

mon-ww fg (*ww1* $\@$ *ww2*) = *mon-ww fg ww1* \cup *mon-ww fg ww2*

<proof>

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

<proof>

lemma *mon-env-single[simp]*:

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

<proof>

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

mon-env fg (*e* $\#$ *w*)

$= (\text{case } e \text{ of } LOC \ a \Rightarrow \{\} \mid ENV \ a \Rightarrow \text{mon-}w \text{ fg } a) \cup \text{mon-env fg } w$
 $\langle \text{proof} \rangle$

lemma *mon-env-unconc*:

$\text{mon-env fg } (w1 @ w2) = \text{mon-env fg } w1 \cup \text{mon-env fg } w2$
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *mon-loc-single[simp]*:

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

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

$\text{mon-loc fg } (e \# w)$
 $= (\text{case } e \text{ of } ENV \ a \Rightarrow \{\} \mid LOC \ a \Rightarrow \text{mon-}w \text{ fg } a) \cup \text{mon-loc fg } w$
 $\langle \text{proof} \rangle$

lemma *mon-loc-unconc*:

$\text{mon-loc fg } (w1 @ w2) = \text{mon-loc fg } w1 \cup \text{mon-loc fg } w2$
 $\langle \text{proof} \rangle$

lemma *mon-ww-of-foldl[simp]*: $\text{mon-}w \text{ fg } (\text{foldl } (op \ @) \ [] \ ww) = \text{mon-ww fg } ww$

$\langle \text{proof} \rangle$

lemma *mon-ww-ileq*: $w \preceq w' \implies \text{mon-ww fg } w \subseteq \text{mon-ww fg } w'$

$\langle \text{proof} \rangle$

lemma *mon-ww-cil*:

$w \in w1 \otimes_{\alpha} w2 \implies \text{mon-ww fg } w = \text{mon-ww fg } w1 \cup \text{mon-ww fg } w2$
 $\langle \text{proof} \rangle$

lemma *mon-loc-cil*:

$w \in w1 \otimes_{\alpha} w2 \implies \text{mon-loc fg } w = \text{mon-loc fg } w1 \cup \text{mon-loc fg } w2$
 $\langle \text{proof} \rangle$

lemma *mon-env-cil*:

$w \in w1 \otimes_{\alpha} w2 \implies \text{mon-env fg } w = \text{mon-env fg } w1 \cup \text{mon-env fg } w2$
 $\langle \text{proof} \rangle$

lemma *mon-ww-of-le-rem*:

$\text{mon-ww fg } (\text{map } le\text{-rem-s } w) = \text{mon-loc fg } w \cup \text{mon-env fg } w$
 $\langle \text{proof} \rangle$

lemma *mon-env-ileq*: $w \preceq w' \implies \text{mon-env fg } w \subseteq \text{mon-env fg } w'$

$\langle \text{proof} \rangle$

lemma *mon-loc-ileq*: $w \preceq w' \implies \text{mon-loc fg } w \subseteq \text{mon-loc fg } w'$

$\langle \text{proof} \rangle$

lemma *mon-loc-map-loc[simp]*: $\text{mon-loc fg } (\text{map } LOC \ w) = \text{mon-ww fg } w$

$\langle \text{proof} \rangle$

lemma *mon-env-map-env[simp]*: $\text{mon-env fg } (\text{map } ENV \ w) = \text{mon-ww fg } w$

$\langle \text{proof} \rangle$
lemma *mon-loc-map-env[simp]*: $\text{mon-loc fg (map ENV w) = \{\}}$
 $\langle \text{proof} \rangle$
lemma *mon-env-map-loc[simp]*: $\text{mon-env fg (map LOC w) = \{\}}$
 $\langle \text{proof} \rangle$
lemma (in *flowgraph*) *ntrs-mon-increasing-s*: $((s,c),e,(s',c')) \in \text{ntrs fg}$
 $\implies \text{mon-s fg s} \subseteq \text{mon-s fg s'} \wedge \text{mon-c fg c} = \text{mon-c fg c'}$
 $\langle \text{proof} \rangle$

lemma (in *flowgraph*) *ntr-mon-increasing-s*:
 $(c,ee,c') \in \text{ntr fg} \implies \text{mon-c fg c} \subseteq \text{mon-c fg c'}$
 $\langle \text{proof} \rangle$

lemma (in *flowgraph*) *ntrp-mon-increasing-s*: $((s,c),e,(s',c')) \in \text{ntrp fg}$
 $\implies \text{mon-s fg s} \subseteq \text{mon-s fg s'} \wedge \text{mon-c fg c} \subseteq \text{mon-c fg c'}$
 $\langle \text{proof} \rangle$

lemma (in *flowgraph*) *ntrp-mon-increasing*: $((s,c),e,(s',c')) \in \text{trcl (ntrp fg)}$
 $\implies \text{mon-s fg s} \subseteq \text{mon-s fg s'} \wedge \text{mon-c fg c} \subseteq \text{mon-c fg c'}$
 $\langle \text{proof} \rangle$

8.4.3 Modifying the context

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

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

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

lemma (in *flowgraph*) *ntrp-mon-env-e-no-ctx*:
 $((s,c),ENV e,(s',c')) \in \text{ntrp fg} \implies \text{mon-w fg e} \cap \text{mon-s fg s} = \{\}$
 $\langle \text{proof} \rangle$

lemma (in *flowgraph*) *ntrp-mon-loc-e-no-ctx*:
 $((s,c),LOC e,(s',c')) \in \text{ntrp fg} \implies \text{mon-w fg e} \cap \text{mon-c fg c} = \{\}$
 $\langle \text{proof} \rangle$

lemma (in *flowgraph*) *ntrp-mon-env-w-no-ctx*:
 $((s,c),w,(s',c')) \in \text{trcl (ntrp fg)} \implies \text{mon-env fg w} \cap \text{mon-s fg s} = \{\}$
 $\langle \text{proof} \rangle$

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

$\langle proof \rangle$

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

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

assumes $A: ((s,c),ee,(s',c')) \in ntrs\ fg$

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

shows $\exists csp. c' = csp + c \wedge mon-c\ fg\ csp = \{\} \wedge ((s,cn),ee,(s',csp+cn)) \in ntrs\ fg$

$\langle proof \rangle$

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

$\llbracket ((s,c),w,(s',c')) \in trcl\ (ntrs\ fg) \rrbracket$

$\implies \forall cn. mon-ww\ fg\ w \cap mon-c\ fg\ cn = \{\}$

$\longrightarrow (\exists csp. c' = csp + c \wedge mon-c\ fg\ csp = \{\} \wedge$
 $((s,cn),w,(s',csp+cn)) \in trcl\ (ntrs\ fg))$

$\langle proof \rangle$

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 $((s,cn),ee,(s',csp+cn)) \in ntrs\ fg$

$\langle proof \rangle$

lemma *ntrs-replace-context-s*:

assumes $A: ((s,c+cr),ee,(s',c'+cr)) \in ntrs\ fg$

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

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

$\langle proof \rangle$

lemma (in *flowgraph*) *ntrs-xchange-context*: $!!s\ c\ c'\ cn. \llbracket$

$((s,c),ww,(s',c')) \in trcl\ (ntrs\ fg);$

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

$\rrbracket \implies \exists csp.$

$c' = csp + c \wedge ((s,cn),ww,(s',csp+cn)) \in trcl\ (ntrs\ fg)$

$\langle proof \rangle$

lemma (in *flowgraph*) *ntrs-replace-context*:

assumes $A: ((s,c+cr),ww,(s',c'+cr)) \in trcl\ (ntrs\ fg)$

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

shows $((s,c+crn),ww,(s',c'+crn)) \in trcl\ (ntrs\ fg)$

$\langle proof \rangle$

lemma (in *flowgraph*) *ntr-add-context-s*:

assumes $A: (c,e,c') \in ntr\ fg$

and $B: mon-w\ fg\ e \cap mon-c\ fg\ cn = \{\}$

shows $(c+cn,e,c'+cn) \in ntr\ fg$

$\langle proof \rangle$

lemma (in *flowgraph*) *ntr-add-context*:
 $\llbracket (c,w,c') \in \text{trcl } (ntr \text{ fg}); \text{ mon-ww fg } w \cap \text{ mon-c fg } cn = \{\} \rrbracket$
 $\implies (c+cn,w,c'+cn) \in \text{trcl } (ntr \text{ fg})$
 $\langle \text{proof} \rangle$

lemma (in *flowgraph*) *ntrs-add-context-s*:
assumes $A: ((s,c),e,(s',c')) \in ntrs \text{ fg}$
and $B: \text{ mon-w fg } e \cap \text{ mon-c fg } cn = \{\}$
shows $((s,c+cn),e,(s',c'+cn)) \in ntrs \text{ fg}$
 $\langle \text{proof} \rangle$

lemma (in *flowgraph*) *ntrp-add-context-s*:
 $\llbracket ((s,c),e,(s',c')) \in ntrp \text{ fg}; \text{ mon-w fg } (le\text{-rem-}s \ e) \cap \text{ mon-c fg } cn = \{\} \rrbracket$
 $\implies ((s,c+cn),e,(s',c'+cn)) \in ntrp \text{ fg}$
 $\langle \text{proof} \rangle$

lemma (in *flowgraph*) *ntrp-add-context*: \llbracket
 $((s,c),w,(s',c')) \in \text{trcl } (ntrp \text{ fg});$
 $\text{ mon-ww fg } (\text{map } le\text{-rem-}s \ w) \cap \text{ mon-c fg } cn = \{\}$
 $\rrbracket \implies ((s,c+cn),w,(s',c'+cn)) \in \text{trcl } (ntrp \text{ fg})$
 $\langle \text{proof} \rangle$

8.4.4 Altering the local stack

lemma *ntrs-stack-comp-s*:
assumes $A: ((s,c),ee,(s',c')) \in ntrs \text{ fg}$
shows $((s@r,c),ee,(s'@r,c')) \in ntrs \text{ fg}$
 $\langle \text{proof} \rangle$

lemma *ntrs-stack-comp*: $((s,c),ww,(s',c')) \in \text{trcl } (ntrs \text{ fg})$
 $\implies ((s@r,c),ww,(s'@r,c')) \in \text{trcl } (ntrs \text{ fg})$
 $\langle \text{proof} \rangle$

lemma (in *flowgraph*) *ntrp-stack-comp-s*:
assumes $A: ((s,c),ee,(s',c')) \in ntrp \text{ fg}$
and $B: \text{ mon-s fg } r \cap \text{ mon-env fg } [ee] = \{\}$
shows $((s@r,c),ee,(s'@r,c')) \in ntrp \text{ fg}$
 $\langle \text{proof} \rangle$

lemma (in *flowgraph*) *ntrp-stack-comp*:
 $\llbracket ((s,c),ww,(s',c')) \in \text{trcl } (ntrp \text{ fg}); \text{ mon-s fg } r \cap \text{ mon-env fg } ww = \{\} \rrbracket$
 $\implies ((s@r,c),ww,(s'@r,c')) \in \text{trcl } (ntrp \text{ fg})$
 $\langle \text{proof} \rangle$

lemma *ntrs-stack-top-decomp-s*:
assumes $A: ((u\#r,c),ee,(s',c')) \in ntrs \text{ fg}$
and $EX: !!v \ u' \ p. \llbracket$
 $s'=v\#u'\#r;$

$$\begin{aligned}
& (([u],c),ee,([v,u'],c')) \in ntrs\ fg; \\
& (u,Call\ p,u') \in edges\ fg \\
& \quad] \implies P \\
& \text{shows } P \\
& \langle proof \rangle
\end{aligned}$$

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

lemma *ntrs-stack-decomp*: $!!u\ s\ r\ c\ P. [$
 $((u\#s@r,c),ww,(s',c')) \in trcl\ (ntrs\ fg);$
 $!!v\ rr. [s'=v\#rr@r; ((u\#s,c),ww,(v\#rr,c')) \in trcl\ (ntrs\ fg)] \implies P$
 $]$ $\implies P$
 $\langle proof \rangle$

lemma *ntrp-stack-decomp-s*:
assumes $A: ((u\#s@r,c),ee,(s',c')) \in ntrp\ fg$
and $EX: !!v\ rr. [s'=v\#rr@r; ((u\#s,c),ee,(v\#rr,c')) \in ntrp\ fg] \implies P$
shows P
 $\langle proof \rangle$

lemma *ntrp-stack-decomp*: $!!u\ s\ r\ c\ P. [$
 $((u\#s@r,c),ww,(s',c')) \in trcl\ (ntrp\ fg);$
 $!!v\ rr. [s'=v\#rr@r; ((u\#s,c),ww,(v\#rr,c')) \in trcl\ (ntrp\ fg)] \implies P$
 $]$ $\implies P$
 $\langle proof \rangle$

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_α -operator:

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

definition

$\alpha n\ fg\ e ==$ if $e = []$ then $(\{\},\{\})$ else $(mon-e\ fg\ (hd\ e),\ mon-w\ fg\ (tl\ e))$

lemma $\alpha n\text{-simps}[simp]$:

$$\alpha n\text{ fg } [] = (\{\}, \{\})$$

$$\alpha n\text{ fg } (e\#w) = (\text{mon-}e\text{ fg } e, \text{mon-}w\text{ fg } w)$$

$\langle\text{proof}\rangle$

definition

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

lemma $\alpha nl\text{-def}'$: $\alpha nl\text{ fg } == \alpha n\text{ fg } \circ \text{le-rem-}s$

$\langle\text{proof}\rangle$

lemma $\alpha nl\text{-simps}[simp]$:

$$\alpha nl\text{ fg } (ENV\ x) = \alpha n\text{ fg } x$$

$$\alpha nl\text{ fg } (LOC\ x) = \alpha n\text{ fg } x$$

$\langle\text{proof}\rangle$

lemma $\alpha nl\text{-simps1}[simp]$:

$$(\alpha nl\text{ fg }) \circ ENV = \alpha n\text{ fg }$$

$$(\alpha nl\text{ fg }) \circ LOC = \alpha n\text{ fg }$$

$\langle\text{proof}\rangle$

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

$\langle\text{proof}\rangle$

lemma $\alpha n\text{-fst-snd}[simp]$: $\text{fst } (\alpha n\text{ fg } w) \cup \text{snd } (\alpha n\text{ fg } w) = \text{mon-}w\text{ fg } w$

$\langle\text{proof}\rangle$

lemma $\text{mon-pl-of-}\alpha nl$: $\text{mon-pl } (\text{map } (\alpha nl\text{ fg }) w) = \text{mon-loc fg } w \cup \text{mon-env fg } w$

$\langle\text{proof}\rangle$

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

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

$\langle\text{proof}\rangle$

lemma $\text{cil-}\alpha nl\text{-cons-helper}$:

$$\text{mon-pl } (\text{map } (\alpha nl\text{ fg }) wb) = \text{mon-}ww\text{ fg } (\text{map } \text{le-rem-}s\ wb)$$

$\langle\text{proof}\rangle$

lemma $\text{cil-}\alpha n\text{-cons1}$: $\llbracket w \in wa \otimes_{\alpha n\text{ fg }} wb; \text{fst } (\alpha n\text{ fg } e) \cap \text{mon-}ww\text{ fg } wb = \{\} \rrbracket$

$$\implies e\#w \in e\#wa \otimes_{\alpha n\text{ fg }} wb$$

$\langle\text{proof}\rangle$

lemma $\text{cil-}\alpha n\text{-cons2}$: $\llbracket w \in wa \otimes_{\alpha n\text{ fg }} wb; \text{fst } (\alpha n\text{ fg } e) \cap \text{mon-}ww\text{ fg } wa = \{\} \rrbracket$

$$\implies e\#w \in wa \otimes_{\alpha n\text{ fg }} e\#wb$$

$\langle\text{proof}\rangle$

8.5.2 Monitors

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

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

shows $\text{mon-}s\text{ fg } s' = \text{mon-}s\text{ fg } s \cup \text{fst } (\alpha n\text{ fg } e)$

$\langle\text{proof}\rangle$

corollary (in *flowgraph*) *ntrs-called-mon*:

assumes $A: ((s,c),e,(s',c')) \in ntrs\ fg$

shows $fst(\alpha n\ fg\ e) \subseteq mon-s\ fg\ s'$

<proof>

lemma (in *flowgraph*) *ntr-mon-s*:

$(c,e,c') \in ntr\ fg \implies mon-c\ fg\ c' = mon-c\ fg\ c \cup fst(\alpha n\ fg\ e)$

<proof>

lemma (in *flowgraph*) *ntrp-mon-s*:

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

shows $mon-c\ fg\ (\{s\}+c') = mon-c\ fg\ (\{s\}+c) \cup fst(\alpha nl\ fg\ e)$

<proof>

8.5.3 Interleaving theorem

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

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

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

$!!ca\ cb. \llbracket (ca+cb, w, c') \in trcl(ntr\ fg); valid\ fg\ (ca+cb) \rrbracket \implies$

$\exists ca'\ cb'\ wa\ wb.$

$c' = ca' + cb' \wedge$

$w \in (wa \otimes_{\alpha n} fg\ wb) \wedge$

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

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

$(ca, wa, ca') \in trcl(ntr\ fg) \wedge (cb, wb, cb') \in trcl(ntr\ fg)$

<proof>

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

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

$!!s\ c1\ c2\ s'\ c'.$

$\llbracket ((s, c1+c2), w, (s', c')) \in trcl(ntrp\ fg); valid\ fg\ (\{s\}+c1+c2) \rrbracket$

$\implies \exists w1\ w2\ c1'\ c2'.$

$w \in w1 \otimes_{\alpha nl} fg\ (map\ ENV\ w2) \wedge$

$c' = c1' + c2' \wedge$

$((s, c1), w1, (s', c1')) \in trcl(ntrp\ fg) \wedge$

$(c2, w2, c2') \in trcl(ntr\ fg) \wedge$

$$\begin{aligned} & \text{mon-ww fg } (\text{map le-rem-s } w1) \cap \text{mon-c fg } c2 = \{\} \wedge \\ & \text{mon-ww fg } w2 \cap \text{mon-c fg } (\{\#s\# \} + c1) = \{\} \end{aligned}$$

$\langle \text{proof} \rangle$

lemma (in *flowgraph*) *nt-split'*:

assumes $A: (ca+cb, w, c') \in \text{trcl } (\text{nt fg})$

and *VALID*: *valid fg* $(ca+cb)$

shows $\exists ca' cb' wa wb.$

$$c' = ca' + cb' \wedge$$

$$w \in (wa \otimes_{\alpha n} fg wb) \wedge$$

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

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

$$(ca, wa, ca') \in \text{trcl } (\text{nt fg}) \wedge$$

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

$\langle \text{proof} \rangle$

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*) *nt-unsplit*:

assumes $A: w \in wa \otimes_{\alpha n} fg wb$ **and**

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

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

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

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

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

$\langle \text{proof} \rangle$

lemma (in *flowgraph*) *nt-unsplit*:

assumes $A: w \in wa \otimes_{\alpha nl} fg (\text{map ENV } wb)$ **and**

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

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

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

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

shows $((s, ca+cb), w, (s', ca'+cb')) \in \text{trcl } (\text{nt rp fg})$

$\langle \text{proof} \rangle$

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*) *nt-interleave*: *valid fg* $(ca+cb) \implies$

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

$(\exists ca' cb' wa wb.$

$$c' = ca' + cb' \wedge$$

$$w \in (wa \otimes_{\alpha n} fg wb) \wedge$$

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

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

$(ca, wa, ca') \in \text{trcl}(\text{ntr fg}) \wedge (cb, wb, cb') \in \text{trcl}(\text{ntr fg})$
 <proof>

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

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

$w \in w1 \otimes_{\alpha n l} \text{fg}(\text{map ENV } w2) \wedge$
 $c' = c1' + c2' \wedge$
 $((s, c1), w1, (s', c1')) \in \text{trcl}(\text{ntrp fg}) \wedge$
 $(c2, w2, c2') \in \text{trcl}(\text{ntr fg}) \wedge$
 $\text{mon-ww fg}(\text{map le-rem-s } w1) \cap$
 $\text{mon-c fg } c2 = \{\}$ \wedge
 $\text{mon-ww fg } w2 \cap \text{mon-c fg } (\{s\} + c1) = \{\}$

<proof>

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

corollary (in *flowgraph*) *ntr2ntrp*: \llbracket

$(c, w, c') \in \text{trcl}(\text{ntr fg});$
 $\text{mon-c fg } (\{s\} + cl) \cap (\text{mon-c fg } c \cup \text{mon-ww fg } w) = \{\}$
 $\rrbracket \implies ((s, cl + c), \text{map ENV } w, (s, cl + c')) \in \text{trcl}(\text{ntrp fg})$

<proof>

8.5.4 Reverse splitting

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

lemma (in *flowgraph*) *ntr-reverse-split*: $\llbracket !w s' ce'. \llbracket$

$(c, w, \{s\} + ce') \in \text{trcl}(\text{ntr fg});$
 $\text{valid fg } c \rrbracket \implies$
 $\exists s ce w1 w2 ce1' ce2'.$
 $c = \{s\} + ce \wedge$
 $ce' = ce1' + ce2' \wedge$
 $w \in w1 \otimes_{\alpha n} \text{fg } w2 \wedge$
 $\text{mon-s fg } s \cap (\text{mon-c fg } ce \cup \text{mon-ww fg } w2) = \{\} \wedge$
 $\text{mon-c fg } ce \cap (\text{mon-s fg } s \cup \text{mon-ww fg } w1) = \{\} \wedge$
 $(\{s\}, w1, \{s'\} + ce1') \in \text{trcl}(\text{ntr fg}) \wedge$
 $(ce, w2, ce2') \in \text{trcl}(\text{ntr fg})$

— The proof works by induction on the initial configuration. Note that configurations consist of finitely many threads only

— FIXME: An induction over the size (rather than over the adding of some fixed element) may lead to a smoother proof here

<proof>

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. Their least solution describes the desired analysis information. In its generic form, a constraint system R is a set of inequations over a complete lattice (L, \sqsubseteq) and a set of variables V . An inequation has the form $R[v] \sqsubseteq \text{rhs}$, where $R[v] \in V$ and 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:

```
for  $(u, \text{Call } q, v) \in E$ :
 $S^k[v] \sqsupseteq \{(\text{mon}(q) \cup M \cup M', \tilde{P}) \mid (M, P) \in S^k[u] \wedge (M', P') \in S^k[r_q]$ 
 $\wedge \tilde{P} \leq P \uplus P' \wedge |\tilde{P}| \leq 2\}$ 
```

For some parameter k and a flowgraph with nodes N and edges E , this generates a constraint system over the variables $\{S^k[v] \mid v \in N\}$. One constraint is generated for each call edge. While we use a powerset lattice here, we can in general use any complete lattice. However, all the constraint systems needed for our conflict analysis are defined over powerset lattices $(\mathcal{P}(a), \subseteq)$ for some type a . This admits a convenient formalization in Isabelle/HOL using inductively defined sets. We inductively define a relation between variables³ and the elements of their values in the least solution, i.e. the set $\{(v, x) \mid x \in R[v]\}$. For example, the constraint generator pattern from above would become the following introduction rule in the inductive definition of the set *S-cs fg k*:

```
 $\llbracket (u, \text{Call } q, v) \in \text{edges fg}; (u, M, P) \in \text{S-cs fg } k;$ 
 $(\text{return fg } q, Ms, Ps) \in \text{S-cs fg } k; P' \leq \#P + Ps; \text{size } P' \leq k \rrbracket$ 
 $\implies (v, \text{mon fg } q \cup M \cup Ms, P') \in \text{S-cs fg } k$ 
```

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 .

³Variables are identified by control nodes here

9.1 Same-level paths

9.1.1 Definition

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

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

inductive-set

$S\text{-cs} :: ('n, 'p, 'ba, 'm, 'more) \text{ flowgraph-rec-scheme} \Rightarrow \text{nat} \Rightarrow$
 $('n \times 'm \text{ set} \times 'p \text{ multiset}) \text{ set}$

for $fg k$

where

$S\text{-init}: (entry\ fg\ p, \{\}, \{\#\}) \in S\text{-cs fg } k$
 $| S\text{-base}: \llbracket (u, Base\ a, v) \in edges\ fg; (u, M, P) \in S\text{-cs fg } k \rrbracket \implies (v, M, P) \in S\text{-cs fg } k$
 $| S\text{-call}: \llbracket (u, Call\ q, v) \in edges\ fg; (u, M, P) \in S\text{-cs fg } k;$
 $\quad (return\ fg\ q, Ms, Ps) \in S\text{-cs fg } k; P' \leq \#P + Ps; size\ P' \leq k \rrbracket$
 $\implies (v, mon\ fg\ q \cup M \cup Ms, P') \in S\text{-cs fg } k$
 $| S\text{-spawn}: \llbracket (u, Spawn\ q, v) \in edges\ fg; (u, M, P) \in S\text{-cs fg } k;$
 $\quad P' \leq \#\{\#q\} + P; size\ P' \leq k \rrbracket$
 $\implies (v, M, P') \in S\text{-cs fg } k$

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

9.1.2 Soundness and Precision

Soundness of the constraint system $S\text{-cs}$ 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 $S\text{-init}$ 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 $\text{flowgraph.trss-find-call}'$). With the induction hypothesis, we get the soundness for the (shorter) prefix of the path, and depending on the last step we can choose a constraint that implies soundness for the whole path.

lemma (in flowgraph) $S\text{-sound}$: $!!p \ v \ c' \ P.$

$$\begin{aligned} & \llbracket ([\text{entry } fg \ p], \{\#\}), w, ([v], c') \in \text{trcl } (\text{trss } fg); \\ & \quad \text{size } P \leq k; (\lambda p. [\text{entry } fg \ p]) \text{ '# } P \leq \# \ c' \rrbracket \\ & \implies (v, \text{mon-}w \ fg \ w, P) \in S\text{-cs } fg \ k \end{aligned}$$

$\langle \text{proof} \rangle$

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

$$\begin{aligned} & \implies \exists p \ c' \ w. \\ & \quad ([[\text{entry } fg \ p], \{\#\}), w, ([v], c') \in \text{trcl } (\text{trss } fg) \wedge \\ & \quad \text{size } P \leq k \wedge \\ & \quad (\lambda p. [\text{entry } fg \ p]) \text{ '# } P \leq \# \ c' \wedge \\ & \quad M = \text{mon-}w \ fg \ w \end{aligned}$$

$\langle \text{proof} \rangle$

theorem (in flowgraph) $S\text{-sound-precise}$:

$$\begin{aligned} & (v, M, P) \in S\text{-cs } fg \ k \iff \\ & (\exists p \ c' \ w. ([[\text{entry } fg \ p], \{\#\}), w, ([v], c') \in \text{trcl } (\text{trss } fg) \wedge \\ & \quad \text{size } P \leq k \wedge (\lambda p. [\text{entry } fg \ p]) \text{ '# } P \leq \# \ c' \wedge M = \text{mon-}w \ fg \ w) \end{aligned}$$

$\langle \text{proof} \rangle$

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

lemma (in flowgraph) $S\text{-sound-ntrop}$:

$$\begin{aligned} & \text{assumes } A: ([u], \{\#\}), \text{eel}, (\text{sh}, \text{ch}) \in \text{ntrop } fg \text{ and} \\ & \text{CASE: } !!p \ u' \ v \ w. \llbracket \end{aligned}$$

$$\begin{aligned}
& eel = LOC (LCall\ p\ \#w); \\
& (u, Call\ p, u') \in edges\ fg; \\
& sh = [v, u']; \\
& proc\text{-}of\ fg\ v = p; \\
& mon\text{-}c\ fg\ ch = \{\}; \\
& !!s. s:\#\ ch \implies \exists p\ u\ v. s = [entry\ fg\ p] \wedge \\
& \quad (u, Spawn\ p, v) \in edges\ fg \wedge \\
& \quad initialproc\ fg\ p; \\
& !!P. (\lambda p. [entry\ fg\ p])\ '\#\ P \leq\#\ ch \implies \\
& \quad (v, mon\text{-}w\ fg\ w, P) \in S\text{-}cs\ fg\ (size\ P) \\
& \quad \implies Q \\
& \text{shows } Q \\
& \langle proof \rangle
\end{aligned}$$

lemma (in *flowgraph*) *S*-precise-*ntrp*:

assumes *ENTRY*: $(v, M, P) \in S\text{-}cs\ fg\ k$ **and**

P: *proc*-of *fg* *v* = *p* **and**

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

shows $\exists w\ ch.$

$(([u], \{\#\}), LOC (LCall\ p\ \#w), ([v, u'], ch)) \in ntrp\ fg \wedge$

$size\ P \leq k \wedge$

$M = mon\text{-}w\ fg\ w \wedge$

$mon\text{-}n\ fg\ v = mon\ fg\ p \wedge$

$(\lambda p. [entry\ fg\ p])\ '\#\ P \leq\#\ ch \wedge$

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

$\langle proof \rangle$

9.2 Single reaching path

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

9.2.1 Constraint system

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

Here, the correspondence between paths and entries included into the inductively defined set is not perfect but strong enough for our purposes: While each constraint system entry corresponds to a path, not each path corresponds to a constraint system entry. But for each path reaching a

configuration at U , we find an entry with less or equal monitors and an acquisition history less or equal to the acquisition history of the path.

inductive-set

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

for $fg\ U$

where

$RU\text{-init}: u \in U \Longrightarrow (u, \{\}, \{\}, \lambda x. \{\}) \in RU\text{-cs}\ fg\ U$

| $RU\text{-call}: \llbracket (u, Call\ p, u') \in edges\ fg; \text{proc-of}\ fg\ v = p; (v, M, P) \in S\text{-cs}\ fg\ 0; (v, Ml, Me, h) \in RU\text{-cs}\ fg\ U; \text{mon-n}\ fg\ u \cap Me = \{\} \rrbracket$

$\Longrightarrow (u, \text{mon}\ fg\ p \cup M \cup Ml, Me, \text{ah-update}\ h\ (\text{mon}\ fg\ p, M)\ (Ml \cup Me)) \in RU\text{-cs}\ fg\ U$

| $RU\text{-spawn}: \llbracket (u, Call\ p, u') \in edges\ fg; \text{proc-of}\ fg\ v = p; (v, M, P) \in S\text{-cs}\ fg\ 1; q: \#P; (\text{entry}\ fg\ q, Ml, Me, h) \in RU\text{-cs}\ fg\ U; (\text{mon-n}\ fg\ u \cup \text{mon}\ fg\ p) \cap (Ml \cup Me) = \{\} \rrbracket$

$\Longrightarrow (u, \text{mon}\ fg\ p \cup M, Ml \cup Me, \text{ah-update}\ h\ (\text{mon}\ fg\ p, M)\ (Ml \cup Me)) \in RU\text{-cs}\ fg\ U$

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

The $RU\text{-init}$ -constraint reflects that we can reach a control node from itself with the empty path. The $RU\text{-call}$ -constraint describes the case that U is reached from the initial thread, and the $RU\text{-spawn}$ -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 ($\text{mon-n}\ fg\ u \cap Me = \{\}$). 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 ($(\text{mon-n}\ fg\ u \cup \text{mon}\ fg\ p) \cap (Ml \cup Me) = \{\}$). Note that in the call case, we do not need to explicitly check that the monitors used by the environment are disjoint from the monitors acquired by the called procedure because this already follows from the existence of a reaching path, as the starting point of this path already holds all these monitors.

However, in the spawn case, we have to check for both the monitors of the start node and of the called procedure to be compatible with the already

known reaching path from the entry node of the spawned thread.

9.2.2 Soundness and precision

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

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

lemma (in *flowgraph*) *ntr-reverse-split-atU*:

assumes V : valid fg c **and**

A : atU U c' **and**

B : $(c, w, c') \in \text{trcl}(\text{ntr fg})$

shows $\exists s w' c1'$.

$s \# c \wedge w' \leq w \wedge c1' \leq \# c' \wedge$

atU U $c1' \wedge (\{ \# s \# \}, w', c1') \in \text{trcl}(\text{ntr fg})$

<proof>

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

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

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

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

lemma (in *flowgraph*) *RU-sound*:

!! $u s' c'$. $\llbracket ([u], \{ \# \}), w, (s', c') \in \text{trcl}(\text{ntrp fg}); \text{atU } U (\{ \# s' \# \} + c') \rrbracket$

$\implies \exists Ml Me h$.

$(u, Ml, Me, h) \in \text{RU-cs fg } U \wedge$

$Ml \subseteq \text{mon-loc fg } w \wedge$

$Me \subseteq \text{mon-env fg } w \wedge$

$h \leq \alpha h (\text{map } (\alpha \text{n}l \text{ fg}) w)$

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

<proof>

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

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

lemma (in flowgraph) RU -precise: $(u, Ml, Me, h) \in RU\text{-cs } fg \ U$
 $\implies \exists w \ s' \ c'.$
 $(([u], \{\#\}), w, (s', c')) \in trcl \ (ntrp \ fg) \wedge$
 $atU \ U \ (\{\#s'\#\} + c') \wedge$
 $mon\text{-}loc \ fg \ w = Ml \wedge$
 $mon\text{-}env \ fg \ w = Me \wedge$
 $\alpha ah \ (map \ (\alpha nl \ fg) \ w) = h$
<proof>

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

inductive-set

$RUV\text{-cs} :: ('n, 'p, 'ba, 'm, 'more) \text{ flowgraph-rec-scheme} \Rightarrow$
 $'n \text{ set} \Rightarrow 'n \text{ set} \Rightarrow ('n \times 'm \text{ set} \times 'm \text{ set}) \text{ set}$

for $fg \ U \ V$

where

$RUV\text{-call}:$

$\llbracket (u, Call \ p, u') \in edges \ fg; \text{proc-of } fg \ v = p; (v, M, P) \in S\text{-cs } fg \ 0;$
 $(v, Ml, Me) \in RUV\text{-cs } fg \ U \ V; \text{mon-n } fg \ u \cap Me = \{\} \rrbracket$
 $\implies (u, mon \ fg \ p \cup M \cup Ml, Me) \in RUV\text{-cs } fg \ U \ V$

| $RUV\text{-spawn}:$

$\llbracket (u, Call \ p, u') \in edges \ fg; \text{proc-of } fg \ v = p; (v, M, P) \in S\text{-cs } fg \ 1; q : \# \ P;$
 $(entry \ fg \ q, Ml, Me) \in RUV\text{-cs } fg \ U \ V;$
 $(mon\text{-}n \ fg \ u \cup mon \ fg \ p) \cap (Ml \cup Me) = \{\} \rrbracket$
 $\implies (u, mon \ fg \ p \cup M, Ml \cup Me) \in RUV\text{-cs } fg \ U \ V$

| $RUV\text{-split-le}:$

$$\begin{aligned}
& \llbracket (u, \text{Call } p, u') \in \text{edges } fg; \text{proc-of } fg \ v = p; (v, M, P) \in S\text{-cs } fg \ 1; q : \# P; \\
& \quad (v, Ml, Me, h) \in RU\text{-cs } fg \ U; (\text{entry } fg \ q, Ml', Me', h') \in RU\text{-cs } fg \ V; \\
& \quad (\text{mon-n } fg \ u \cup \text{mon } fg \ p) \cap (Me \cup Ml' \cup Me') = \{\}; h \ [*] \ h' \rrbracket \\
& \implies (u, \text{mon } fg \ p \cup M \cup Ml, Me \cup Ml' \cup Me') \in RUV\text{-cs } fg \ U \ V \\
& | \text{RUV-split-el:} \\
& \llbracket (u, \text{Call } p, u') \in \text{edges } fg; \text{proc-of } fg \ v = p; (v, M, P) \in S\text{-cs } fg \ 1; q : \# P; \\
& \quad (v, Ml, Me, h) \in RU\text{-cs } fg \ V; (\text{entry } fg \ q, Ml', Me', h') \in RU\text{-cs } fg \ U; \\
& \quad (\text{mon-n } fg \ u \cup \text{mon } fg \ p) \cap (Me \cup Ml' \cup Me') = \{\}; h \ [*] \ h' \rrbracket \\
& \implies (u, \text{mon } fg \ p \cup M \cup Ml, Me \cup Ml' \cup Me') \in RUV\text{-cs } fg \ U \ V \\
& | \text{RUV-split-ee:} \\
& \llbracket (u, \text{Call } p, u') \in \text{edges } fg; \text{proc-of } fg \ v = p; (v, M, P) \in S\text{-cs } fg \ 2; \\
& \quad \{\#q\} + \{\#q'\} \leq \# P; \\
& \quad (\text{entry } fg \ q, Ml, Me, h) \in RU\text{-cs } fg \ U; (\text{entry } fg \ q', Ml', Me', h') \in RU\text{-cs } fg \ V; \\
& \quad (\text{mon-n } fg \ u \cup \text{mon } fg \ p) \cap (Ml \cup Me \cup Ml' \cup Me') = \{\}; h \ [*] \ h' \rrbracket \\
& \implies (u, \text{mon } fg \ p \cup M, Ml \cup Me \cup Ml' \cup Me') \in RUV\text{-cs } fg \ U \ V
\end{aligned}$$

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

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

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

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

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

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

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

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

9.3.2 Soundness and precision

lemma (in flowgraph) RUV-sound: $\llbracket u \ s' \ c' \rrbracket$

$$\llbracket (([u], \{\#\}), w, (s', c')) \in \text{trcl} (\text{ntrp } fg); \text{atUV } U \ V \ (\{\#s'\} + c') \rrbracket$$

$\implies \exists Ml Me.$
 $(u, Ml, Me) \in RUV\text{-cs } fg \ U \ V \ \wedge$
 $Ml \subseteq mon\text{-loc } fg \ w \ \wedge$
 $Me \subseteq mon\text{-env } fg \ w$
 — The soundness proof is done by induction over the length of the reaching path
 $\langle proof \rangle$

lemma (in *flowgraph*) *RUV-precise*: $(u, Ml, Me) \in RUV\text{-cs } fg \ U \ V$
 $\implies \exists w \ s' \ c'.$
 $(([u], \{\#\}), w, (s', c')) \in trcl \ (ntrp \ fg) \ \wedge$
 $atUV \ U \ V \ (\{\#s'\#\} + c') \ \wedge$
 $mon\text{-loc } fg \ w = Ml \ \wedge$
 $mon\text{-env } fg \ w = Me$
 $\langle proof \rangle$

end

10 Main Result

theory *MainResult*
imports *ConstraintSystems*
begin

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

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

- The flowgraph and the assumptions made on it in the *flowgraph*- and *eflowgraph*-locales. Note that we show in Section 6.4 that there is at least one non-trivial model of *eflowgraph*.
- The reference point semantics (*refpoint*) and the transitive closure operator (*trcl*).
- The definition of *atUV*.
- All dependencies of the above definitions in the Isabelle standard libraries.

theorem (in *eflowgraph*) *RUV-is-sim-reach*:
 $(\exists w \ c'. (\#[entry \ fg \ (main \ fg)]\#), w, c') \in trcl \ (refpoint \ fg) \ \wedge \ atUV \ U \ V \ c'$
 $\iff (\exists Ml \ Me. (entry \ fg \ (main \ fg), Ml, Me) \in RUV\text{-cs } fg \ U \ V)$

— The proof uses the soundness and precision theorems wrt. to normalized paths (*flowgraph.RUV-sound*, *flowgraph.RUV-precise*) as well as the normalization result, i.e. that every reachable configuration is also reachable using a normalized path

(*eflowgraph.normalize*) and, vice versa, that every normalized path is also a usual path (*ntr-is-tr*). Finally the conversion between our working semantics and the semantic reference point is exploited (*flowgraph.refpoint-eq*).
<proof>

end

11 Conclusion

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

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

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

Acknowledgement We thank Dejvuth Suwimonteerabuth for an interesting discussion about static analysis of programs with locks. We also

thank the people on the Isabelle mailing list for quick and useful responses.

References

- [1] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. of CONCUR'05*. Springer, 2005.
- [2] J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Proc. of FoSSaCS'99*, pages 14–30. Springer, 1999.
- [3] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *Proc. of POPL'00*, pages 1–11. Springer, 2000.
- [4] V. Kahlon and A. Gupta. An automata-theoretic approach for model checking threads for LTL properties. In *Proc. of LICS 2006*, pages 101–110. IEEE Computer Society, 2006.
- [5] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *Proc. of CAV 2005*, pages 505–518. Springer, 2005.
- [6] P. Lammich and M. Müller-Olm. Precise fixpoint-based analysis of programs with thread-creation. In *Proc. of CONCUR 2007*, pages 287–302. Springer, 2007.
- [7] H. Seidl and B. Steffen. Constraint-based inter-procedural analysis of parallel programs. *Nordic Journal of Computing (NJC)*, 7(4):375–400, 2000.