Worklist Algorithms

Simon Wimmer, Peter Lammich

March 17, 2025

Abstract

This entry verifies a number of worklist algorithms for exploring sets of reachable sets of transition systems with *subsumption* relations. Informally speaking, a node a is subsumed by a node b if everything that is reachable from a is also reachable from b. Starting from a general abstract view of transition systems, we gradually add structure while refining our algorithms to more efficient versions. In the end, we obtain efficient imperative algorithms, which operate on a shared data structure to keep track of explored and yet-to-be-explored states, similar to the algorithms used in timed automata model checking [2, 1]. This entry forms part of the work described in a paper by the authors of this entry [4] and a PhD thesis [3].

Contents

1 Preliminaries

theory *Worklist-Locales*
imports *Refine-Imperative-HOL.Sepref Collections.HashCode Timed-Automata.Graphs*
begin

1.1 Search Spaces

A search space consists of a step relation, a start state, a final state predicate, and a subsumption preorder.

locale *Search-Space-Defs* =
fixes $E :: 'a \Rightarrow 'a \Rightarrow bool$ — Step relation
and $a_0 :: 'a$ — Start state
and $F :: 'a \Rightarrow bool$ — Final states
and *subsumes* $:: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\langle\preceq\rangle$ *50*) — Subsumption preorder
begin

sublocale *Graph-Start-Defs* E a_0 .

definition *subsumes-strictly* (**infix** $\langle\prec\rangle$ *50*) **where**
subsumes-strictly x $y = (x \preceq y \wedge \neg\ y \preceq x)$

no-notation *fun-rel-syn* (**infixr** $\langle\rightarrow\rangle$ *60*)

definition *F-reachable* $\equiv \exists\, a.$ *reachable* $a \wedge F\ a$

end

locale *Search-Space-Nodes-Defs* = *Search-Space-Defs* +
fixes $V :: 'a \Rightarrow bool$

locale *Search-Space-Defs-Empty* = *Search-Space-Defs* +
fixes *empty* $:: 'a \Rightarrow bool$

locale *Search-Space-Nodes-Empty-Defs* = *Search-Space-Nodes-Defs* + *Search-Space-Defs-Empty*

locale *Search-Space-Nodes* = *Search-Space-Nodes-Defs* +
assumes *refl*[*intro*!, *simp*]: $a \preceq a$
and *trans*[*trans*]: $a \preceq b \Longrightarrow b \preceq c \Longrightarrow a \preceq c$

assumes *mono*:
$a \preceq b \Longrightarrow E\ a\ a' \Longrightarrow V\ a \Longrightarrow V\ b \Longrightarrow \exists\ b'.\ V\ b' \wedge E\ b\ b' \wedge a' \preceq b'$
and *F-mono*: $a \preceq a' \Longrightarrow F\ a \Longrightarrow F\ a'$
begin

sublocale *preorder* $(\preceq)$ $(\prec)$
by *standard* (*auto simp*: *subsumes-strictly-def intro*: *trans*)

end

The set of reachable states must be finite, subsumption must be a preorder, and be compatible with steps and final states.

locale *Search-Space-Nodes-Empty* = *Search-Space-Nodes-Empty-Defs* +
assumes *refl*[*intro*!, *simp*]: $a \preceq a$
and *trans*[*trans*]: $a \preceq b \Longrightarrow b \preceq c \Longrightarrow a \preceq c$

assumes *mono*:
$a \preceq b \Longrightarrow E\ a\ a' \Longrightarrow V\ a \Longrightarrow V\ b \Longrightarrow \neg\ empty\ a \Longrightarrow \exists\ b'.\ V\ b' \wedge E\ b\ b' \wedge a' \preceq b'$
and *empty-subsumes*: $empty\ a \Longrightarrow a \preceq a'$
and *empty-mono*: $\neg\ empty\ a \Longrightarrow a \preceq b \Longrightarrow \neg\ empty\ b$
and *empty-E*: $V\ x \Longrightarrow empty\ x \Longrightarrow E\ x\ x' \Longrightarrow empty\ x'$
and *F-mono*: $a \preceq a' \Longrightarrow F\ a \Longrightarrow F\ a'$
begin

sublocale *preorder* $(\preceq)$ $(\prec)$
by *standard* (*auto simp*: *subsumes-strictly-def intro*: *trans*)

sublocale *search-space*:
Search-Space-Nodes $\lambda\ x\ y.\ E\ x\ y \wedge \neg\ empty\ y\ a_0\ F\ (\preceq)\ \lambda\ v.\ V\ v \wedge \neg\ empty\ v$
apply *standard*
apply *blast*
apply (*blast intro*: *trans*)
apply (*drule mono*; *blast dest*: *empty-mono*)
apply (*blast intro*: *F-mono*)
done

end

The set of reachable states must be finite, subsumption must be a preorder, and be compatible with steps and final states.

locale *Search-Space* = *Search-Space-Defs-Empty* +
assumes *refl*[*intro*!, *simp*]: $a \preceq a$
and *trans*[*trans*]: $a \preceq b \Longrightarrow b \preceq c \Longrightarrow a \preceq c$

assumes *mono*:
$a \preceq b \Longrightarrow E\ a\ a' \Longrightarrow reachable\ a \Longrightarrow reachable\ b \Longrightarrow \neg\ empty\ a \Longrightarrow$

$\exists\ b'.\ E\ b\ b' \wedge a' \preceq b'$
and *empty-subsumes*: $empty\ a \Longrightarrow a \preceq a'$
and *empty-mono*: $\neg\ empty\ a \Longrightarrow a \preceq b \Longrightarrow \neg\ empty\ b$
and *empty-E*: $reachable\ x \Longrightarrow empty\ x \Longrightarrow E\ x\ x' \Longrightarrow empty\ x'$
and *F-mono*: $a \preceq a' \Longrightarrow F\ a \Longrightarrow F\ a'$
begin

sublocale *preorder* $(\preceq)$ $(\prec)$
by *standard* (*auto simp*: *subsumes-strictly-def intro*: *trans*)

sublocale *Search-Space-Nodes-Empty E* a_0 *F* $(\preceq)$ *reachable empty*
including *graph-automation*
by *standard*
(*auto intro*: *trans empty-subsumes dest*: *empty-mono empty-E F-mono*, *auto 4 4 dest*: *mono*)

end

locale *Search-Space-finite* = *Search-Space* +
assumes *finite-reachable*: *finite* $\{a.\ reachable\ a \wedge \neg\ empty\ a\}$

locale *Search-Space-finite-strict* = *Search-Space* +
assumes *finite-reachable*: *finite* $\{a.\ reachable\ a\}$

sublocale *Search-Space-finite-strict* $\subseteq$ *Search-Space-finite*
by *standard* (*auto simp*: *finite-reachable*)

locale *Search-Space′* = *Search-Space* +
assumes *final-non-empty*: $F\ a \Longrightarrow \neg\ empty\ a$

locale *Search-Space′-finite* = *Search-Space′* + *Search-Space-finite*

locale *Search-Space″-Defs* = *Search-Space-Defs-Empty* +
fixes *subsumes′* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\langle\trianglelefteq\rangle$ *50*) — Subsumption preorder

locale *Search-Space″-pre* = *Search-Space″-Defs* +
assumes *empty-subsumes′*: $\neg\ empty\ a \Longrightarrow a \preceq b \longleftrightarrow a \trianglelefteq b$

locale *Search-Space″-start* = *Search-Space″-pre* +
assumes *start-non-empty* [*simp*]: $\neg\ empty\ a_0$

locale *Search-Space″* = *Search-Space″-pre* + *Search-Space′*

locale *Search-Space″-finite* = *Search-Space″* + *Search-Space-finite*

sublocale *Search-Space''-finite* $\subseteq$ *Search-Space'-finite* **..**

locale *Search-Space''-finite-strict* = *Search-Space''* + *Search-Space-finite-strict*

locale *Search-Space-Key-Defs* =
Search-Space''-Defs E **for** *E* :: $'v \Rightarrow 'v \Rightarrow$ *bool* +
fixes *key* :: $'v \Rightarrow 'k$

locale *Search-Space-Key* =
Search-Space-Key-Defs + *Search-Space''* +
assumes *subsumes-key*[*intro*, *simp*]: $a \trianglelefteq b \Longrightarrow$ *key a* = *key b*

locale *Worklist0-Defs* = *Search-Space-Defs* +
fixes *succs* :: $'a \Rightarrow 'a$ *list*

locale *Worklist0* = *Worklist0-Defs* + *Search-Space* +
assumes *succs-correct*: *reachable a* $\Longrightarrow$ *set* (*succs a*) = *Collect* (*E a*)

locale *Worklist1-Defs* = *Worklist0-Defs* + *Search-Space-Defs-Empty*

locale *Worklist1* = *Worklist1-Defs* + *Worklist0*

locale *Worklist2-Defs* = *Worklist1-Defs* + *Search-Space''-Defs*

locale *Worklist2* = *Worklist2-Defs* + *Worklist1* + *Search-Space''-pre* + *Search-Space*

locale *Worklist3-Defs* = *Worklist2-Defs* +
fixes *F'* :: $'a \Rightarrow$ *bool*

locale *Worklist3* = *Worklist3-Defs* + *Worklist2* +
assumes *F-split*: *F a* $\longleftrightarrow \neg$ *empty a* $\wedge$ *F' a*

locale *Worklist4* = *Worklist3* + *Search-Space''*

locale *Worklist-Map-Defs* = *Search-Space-Key-Defs* + *Worklist2-Defs*

locale *Worklist-Map* =
Worklist-Map-Defs + *Search-Space-Key* + *Worklist2*

locale *Worklist-Map2-Defs* = *Worklist-Map-Defs* + *Worklist3-Defs*

locale *Worklist-Map2* = *Worklist-Map2-Defs* + *Worklist-Map* + *Worklist3*

locale *Worklist-Map2-finite* = *Worklist-Map2* + *Search-Space-finite*

sublocale *Worklist-Map2-finite* ⊆ *Search-Space''-finite* **..**

locale *Worklist4-Impl-Defs* = *Worklist3-Defs* +
fixes *A* :: *'a* ⇒ *'ai* ⇒ *assn*
fixes *succsi* :: *'ai* ⇒ *'ai list Heap*
fixes $a_0 i$ :: *'ai Heap*
fixes *Fi* :: *'ai* ⇒ *bool Heap*
fixes *Lei* :: *'ai* ⇒ *'ai* ⇒ *bool Heap*
fixes *emptyi* :: *'ai* ⇒ *bool Heap*

locale *Worklist4-Impl* = *Worklist4-Impl-Defs* + *Worklist4* +
— This is the easy variant: Operations cannot depend on additional heap.
assumes [*sepref-fr-rules*]: (*uncurry0* $a_0 i$, *uncurry0* (*RETURN* (*PR-CONST* a_0))) ∈ *unit-assn*^{k} →$_a$ *A*
assumes [*sepref-fr-rules*]: (*Fi*,*RETURN o PR-CONST F'*) ∈ *A*^{k} →$_a$ *bool-assn*
assumes [*sepref-fr-rules*]: (*uncurry Lei*,*uncurry* (*RETURN oo PR-CONST* (⊴))) ∈ *A*^{k} ∗$_a$ *A*^{k} →$_a$ *bool-assn*
assumes [*sepref-fr-rules*]: (*succsi*,*RETURN o PR-CONST succs*) ∈ *A*^{k} →$_a$ *list-assn A*
assumes [*sepref-fr-rules*]: (*emptyi*,*RETURN o PR-CONST empty*) ∈ *A*^{k} →$_a$ *bool-assn*

locale *Worklist4-Impl-finite-strict* = *Worklist4-Impl* + *Search-Space-finite-strict*

sublocale *Worklist4-Impl-finite-strict* ⊆ *Search-Space''-finite-strict* **..**

locale *Worklist-Map2-Impl-Defs* =
Worklist4-Impl-Defs - - - - - - - - *A* + *Worklist-Map2-Defs* a_0 - - - - - *key*
for *A* :: *'a* ⇒ *'ai* :: {*heap*} ⇒ - **and** *key* :: *'a* ⇒ *'k* +
fixes *keyi* :: *'ai* ⇒ *'ki* :: {*hashable*, *heap*} *Heap*
fixes *copyi* :: *'ai* ⇒ *'ai Heap*
fixes *tracei* :: *string* ⇒ *'ai* ⇒ *unit Heap*

end
theory *Worklist-Common*
imports *Worklist-Locales*
begin

lemma *list-ex-foldli*:
list-ex P xs = *foldli xs Not* (λ *x y*. *P x* ∨ *y*) *False*

apply (*induction xs*)
subgoal
by *simp*
subgoal for *x xs*
by (*induction xs*) *auto*
done

lemma (**in** *Search-Space-finite*) *finitely-branching*:
assumes *reachable a*
shows *finite* ({a'. *E a a'* ∧ ¬ *empty a'*})
proof −
have {a'. *E a a'* ∧ ¬ *empty a'*} ⊆ {a'. *reachable a'* ∧ ¬ *empty a'*}
using *assms*(*1*) **by** (*auto intro*: *reachable-step*)
then show *?thesis* **using** *finite-reachable*
by (*rule finite-subset*)
qed

definition (**in** *Search-Space-Key-Defs*)
map-set-rel =
{(*m*, *s*).
⋃(*ran m*) = *s* ∧ (∀ *k*. ∀ *x*. *m k* = *Some x* ⟶ (∀ *v* ∈ *x*. *key v* = *k*))
∧
finite (*dom m*) ∧ (∀ *k S*. *m k* = *Some S* ⟶ *finite S*)
}

end

1.2 Miscellaneous

theory *Worklist-Algorithms-Misc*
imports *HOL−Library.Multiset*
begin

lemma *mset-eq-empty-iff*:
M = {#} ⟷ *set-mset M* = {}
by *auto*

lemma *filter-mset-eq-empty-iff*:
{#*x* ∈# *A*. *P x*#} = {#} ⟷ (∀ *x* ∈ *set-mset A*. ¬ *P x*)
by (*auto simp*: *mset-eq-empty-iff*)

lemma *mset-remove-member*:
x ∈# *A* − *B* **if** *x* ∈# *A x* ∉# *B*
using *that*

by (*metis count-greater-zero-iff in-diff-count not-in-iff*)

end
theory *Worklist-Algorithms-Tracing*
imports *Main Refine-Imperative-HOL.Sepref*
begin

datatype *message* = *ExploredState*

definition *write-msg* :: *message* ⇒ *unit* **where**
write-msg m = ()

code-printing code-module *Tracing* ⇀ (*SML*)
‹
structure Tracing : *sig*
val count-up : *unit* −> *unit*
val get-count : *unit* −> *int*
end = *struct*
val counter = *Unsynchronized.ref 0*;
fun count-up () = (*counter* := !*counter* + *1*);
fun get-count () = !*counter*;
end
› **and** (*OCaml*)
‹
module Tracing : *sig*
val count-up : *unit* −> *unit*
val get-count : *unit* −> *int*
end = *struct*
let counter = *ref 0*
let count-up () = (*counter* := !*counter* + *1*)
let get-count () = !*counter*
end
›

code-reserved (*SML*) *Tracing*

code-reserved (*OCaml*) *Tracing*

code-printing
constant *write-msg* ⇀ (*SML*) (*fn x* => *Tracing.count′-up* ()) -
and (*OCaml*) (*fun x* −> *Tracing.count′-up* ()) -

definition *trace* **where**
trace m x = (*let a* = *write-msg m in x*)

lemma *trace-alt-def*[*simp*]:
trace m x = (λ -. *x*) (*write-msg x*)
unfolding *trace-def* **by** *simp*

definition
test m = *trace ExploredState* ((*3* :: *int*) + *1*)

definition *TRACE m* = *RETURN* (*trace m* ())

lemma *TRACE-bind*[*simp*]:
do {*TRACE m*; *c*} = *c*
unfolding *TRACE-def* **by** *simp*

lemma [*sepref-import-param*]:
(*trace*, *trace*) ∈ ⟨*Id*,⟨*Id*,*Id*⟩*fun-rel*⟩*fun-rel*
by *simp*

sepref-definition *TRACE-impl* **is**
TRACE :: *id-assn*^{k} →$_a$ *unit-assn*
unfolding *TRACE-def* **by** *sepref*

lemmas [*sepref-fr-rules*] = *TRACE-impl.refine*

Somehow Sepref does not want to pick up TRACE as it is, so we use the following workaround:

definition *TRACE′* = *TRACE ExploredState*

definition *trace′* = *trace ExploredState*

lemma *TRACE′-alt-def*:
TRACE′ = *RETURN* (*trace′* ())
unfolding *TRACE-def TRACE′-def trace′-def* **..**

lemma [*sepref-import-param*]:
(*trace′*, *trace′*) ∈ ⟨*Id*,*Id*⟩*fun-rel*
by *simp*

sepref-definition *TRACE′-impl* **is**
uncurry0 TRACE′ :: *unit-assn*^{k} →$_a$ *unit-assn*
unfolding *TRACE′-alt-def*
by *sepref*

lemmas [*sepref-fr-rules*] = *TRACE′-impl.refine*

end

2 Subsumption Graphs

theory *Worklist-Algorithms-Subsumption-Graphs*
imports
Timed-Automata.Graphs
Timed-Automata.More-List
begin

2.1 Preliminaries

Transitive Closure **context**
fixes $R :: 'a \Rightarrow 'a \Rightarrow bool$
assumes *R-trans*[*intro*]: $\bigwedge x\ y\ z.\ R\ x\ y \Longrightarrow R\ y\ z \Longrightarrow R\ x\ z$
begin

lemma *rtranclp-transitive-compress1*: $R\ a\ c$ **if** $R\ a\ b\ R^{**}\ b\ c$
using *that*(*2*,*1*) **by** *induction auto*

lemma *rtranclp-transitive-compress2*: $R\ a\ c$ **if** $R^{**}\ a\ b\ R\ b\ c$
using *that* **by** *induction auto*

end

lemma *rtranclp-ev-induct*[*consumes 1*, *case-names irrefl trans step*]:
fixes $P :: 'a \Rightarrow bool$ **and** $R :: 'a \Rightarrow 'a \Rightarrow bool$
assumes *reachable-finite*: *finite* $\{x.\ R^{**}\ a\ x\}$
assumes *R-irrefl*: $\bigwedge x.\ \neg\ R\ x\ x$ **and** *R-trans*[*intro*]: $\bigwedge x\ y\ z.\ R\ x\ y \Longrightarrow R\ y\ z \Longrightarrow R\ x\ z$
assumes *step*: $\bigwedge x.\ R^{**}\ a\ x \Longrightarrow P\ x \vee (\exists\ y.\ R\ x\ y)$
shows $\exists\ x.\ P\ x \wedge R^{**}\ a\ x$
proof −
let *?S* = $\{y.\ R^{**}\ a\ y\}$
from *reachable-finite* **have** *finite ?S*
by *auto*
then have $\exists\ x \in ?S.\ P\ x$
using *step*
proof (*induction ?S arbitrary*: *a rule*: *finite-psubset-induct*)
case *psubset*
let *?S* = $\{y.\ R^{**}\ a\ y\}$
from *psubset* **have** *finite ?S* **by** *auto*
show *?case*

```
    proof (cases ?S = {})
      case True
      then show ?thesis by auto
    next
      case False
      then obtain y where R** a y
        by auto
      from psubset(3)[OF this] show ?thesis
      proof
        assume P y
        with ‹R** a y› show ?thesis by auto
      next
        assume ∃ z. R y z
        then obtain z where R y z by safe
        let ?T = {y. R** z y}
        from ‹R y z› ‹R** a y› have ¬ R** z a
          by (auto simp: R-irrefl dest!: rtranclp-transitive-compress2[of R,
rotated])
        then have a ∉ ?T by auto
        moreover have ?T ⊆ ?S
          using ‹R** a y› ‹R y z› by auto
        ultimately have ?T ⊂ ?S
          by auto
        have P x ∨ Ex (R x) if R** z x for x
          using that ‹R y z› ‹R** a y› by (auto intro!: psubset.prems)
         from psubset.hyps(2)[OF ‹?T ⊂ ?S› this] psubset.prems ‹R y z›
‹R** a y› obtain w
          where R** z w P w by auto
        with ‹R** a y› ‹R y z› have R** a w by auto
        with ‹P w› show ?thesis by auto
      qed
    qed
  qed
  then show ?thesis by auto
qed

lemma rtranclp-ev-induct2[consumes 2, case-names irrefl trans step]:
  fixes P Q :: 'a ⇒ bool
  assumes Q-finite: finite {x. Q x} and Q-witness: Q a
  assumes R-irrefl: ⋀ x. ¬ R x x and R-trans[intro]: ⋀ x y z. R x y ⟹
R y z ⟹ R x z
  assumes step: ⋀ x. Q x ⟹ P x ∨ (∃ y. R x y ∧ Q y)
  shows ∃ x. P x ∧ Q x ∧ R** a x
proof −
```

let *?R* = λ *x y*. *R x y* ∧ *Q x* ∧ *Q y*
have [*intro*]: *R*** *a x* **if** *?R*** *a x* **for** *x*
using *that* **by** *induction auto*
have [*intro*]: *Q x* **if** *?R*** *a x* **for** *x*
using *that* ‹*Q a*› **by** (*auto elim*: *rtranclp.cases*)
have {*x*. *?R*** *a x*} ⊆ {*x*. *Q x*} **by** *auto*
with ‹*finite* -› **have** *finite* {*x*. *?R*** *a x*} **by** − (*rule finite-subset*)
then have ∃ *x*. *P x* ∧ *?R*** *a x*
proof (*induction rule*: *rtranclp-ev-induct*)
case *prems*: (*step x*)
with *step*[*of x*] **show** *?case* **by** *auto*
qed (*auto simp*: *R-irrefl*)
then show *?thesis* **by** *auto*
qed

2.2 Definitions

locale *Subsumption-Graph-Pre-Defs* =
ord less-eq less **for** *less-eq* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ‹⪯› *50*) **and** *less* (**infix** ‹≺› *50*) +
fixes *E* :: *'a* ⇒ *'a* ⇒ *bool* — The full edge set
begin

sublocale *Graph-Defs E* **.**

end

locale *Subsumption-Graph-Pre-Nodes-Defs* = *Subsumption-Graph-Pre-Defs* +
fixes *V* :: *'a* ⇒ *bool*
begin

sublocale *Subgraph-Node-Defs-Notation* **.**

end

locale *Subsumption-Graph-Defs* = *Subsumption-Graph-Pre-Defs* +
fixes s_0 :: *'a* — Start state
fixes *RE* :: *'a* ⇒ *'a* ⇒ *bool* — Subgraph of the graph given by the full edge set
begin

sublocale *Graph-Start-Defs E* s_0 **.**

sublocale *G*: *Graph-Start-Defs RE* s_0 **.**

sublocale *G'*: *Graph-Start-Defs* λ *x y. RE x y* $\vee$ $(x \prec y \wedge$ *G.reachable y*$)$ s_0 **.**

abbreviation *G'-E* $(\langle\text{-} \rightarrow_{G'} \text{-}\rangle\ [100, 100]\ 40)$ **where**
G'-E x y $\equiv$ *RE x y* $\vee$ $(x \prec y \wedge$ *G.reachable y*$)$

notation *RE* $(\langle\text{-} \rightarrow_{G} \text{-}\rangle\ [100, 100]\ 40)$

notation *G.reaches* $(\langle\text{-} \rightarrow_{G}* \text{-}\rangle\ [100, 100]\ 40)$

notation *G.reaches1* $(\langle\text{-} \rightarrow_{G}{}^{+} \text{-}\rangle\ [100, 100]\ 40)$

notation *G'.reaches* $(\langle\text{-} \rightarrow_{G}*'' \text{-}\rangle\ [100, 100]\ 40)$

notation *G'.reaches1* $(\langle\text{-} \rightarrow_{G}{}^{+}{}'' \text{-}\rangle\ [100, 100]\ 40)$

end

locale *Subsumption-Graph-Pre* = *Subsumption-Graph-Defs* + *preorder less-eq less* +
assumes *mono*:
$a \preceq b \Longrightarrow E\ a\ a' \Longrightarrow$ *reachable a* $\Longrightarrow$ *reachable b* $\Longrightarrow \exists\ b'.\ E\ b\ b' \wedge a' \preceq b'$
begin

lemmas *preorder-intros* = *order-trans less-trans less-imp-le*

end

locale *Subsumption-Graph-Pre-Nodes* = *Subsumption-Graph-Pre-Nodes-Defs* + *preorder less-eq less* +
assumes *mono*:
$a \preceq b \Longrightarrow a \rightarrow a' \Longrightarrow V\ a \Longrightarrow V\ b \Longrightarrow \exists\ b'.\ b \rightarrow b' \wedge a' \preceq b'$
begin

lemmas *preorder-intros* = *order-trans less-trans less-imp-le*

end

This is sufficient to show that if $\rightarrow_G$ cannot reach an accepting state, then $\rightarrow$ cannot either.

locale *Reachability-Compatible-Subsumption-Graph-Pre* =
Subsumption-Graph-Defs + *preorder less-eq less* +
assumes *mono*:
$a \preceq b \Longrightarrow E\ a\ a' \Longrightarrow$ *reachable* $a \vee$ *G.reachable* $a \Longrightarrow$ *reachable* $b \vee$ *G.reachable* b
$\Longrightarrow \exists\ b'.\ E\ b\ b' \wedge a' \preceq b'$
assumes *reachability-compatible*:
$\forall\ s.$ *G.reachable* $s \longrightarrow (\forall\ s'.\ E\ s\ s' \longrightarrow RE\ s\ s') \vee (\exists\ t.\ s \prec t \wedge$ *G.reachable* $t)$
assumes *finite-reachable*: *finite* $\{a.$ *G.reachable* $a\}$

locale *Reachability-Compatible-Subsumption-Graph* =
Subsumption-Graph-Defs + *Subsumption-Graph-Pre* +
assumes *reachability-compatible*:
$\forall\ s.$ *G.reachable* $s \longrightarrow (\forall\ s'.\ E\ s\ s' \longrightarrow RE\ s\ s') \vee (\exists\ t.\ s \prec t \wedge$ *G.reachable* $t)$
assumes *subgraph*: $\forall\ s\ s'.\ RE\ s\ s' \longrightarrow E\ s\ s'$
assumes *finite-reachable*: *finite* $\{a.$ *G.reachable* $a\}$

locale *Subsumption-Graph-View-Defs* = *Subsumption-Graph-Defs* +
fixes *SE* :: $'a \Rightarrow 'a \Rightarrow$ *bool* — Subsumption edges
and *covered* :: $'a \Rightarrow$ *bool*

locale *Reachability-Compatible-Subsumption-Graph-View* =
Subsumption-Graph-View-Defs + *Subsumption-Graph-Pre* +
assumes *reachability-compatible*:
$\forall\ s.$ *G.reachable* $s \longrightarrow$
(*if covered* s *then* $(\exists\ t.\ SE\ s\ t \wedge$ *G.reachable* $t)$ *else* $(\forall\ s'.\ E\ s\ s' \longrightarrow RE\ s\ s')$)
assumes *subsumption*: $\forall\ s'.\ SE\ s\ s' \longrightarrow s \prec s'$
assumes *subgraph*: $\forall\ s\ s'.\ RE\ s\ s' \longrightarrow E\ s\ s'$
assumes *finite-reachable*: *finite* $\{a.$ *G.reachable* $a\}$
begin

sublocale *Reachability-Compatible-Subsumption-Graph* $(\preceq)$ $(\prec)$ $E\ s_0\ RE$
proof *unfold-locales*
have $(\forall s'.\ E\ s\ s' \longrightarrow RE\ s\ s') \vee (\exists t.\ s \prec t \wedge$ *G.reachable* $t)$ **if** *G.reachable* s **for** s
using *that reachability-compatible subsumption* **by** (*cases covered s*; *fastforce*)
then show $\forall s.$ *G.reachable* $s \longrightarrow (\forall s'.\ E\ s\ s' \longrightarrow RE\ s\ s') \vee (\exists t.\ s \prec t$

$\wedge$ *G.reachable t*)
by *auto*
qed (*use subgraph* **in** ‹*auto intro*: *finite-reachable mono*›)

end

locale *Subsumption-Graph-Closure-View-Defs* =
ord less-eq less **for** *less-eq* :: $'b \Rightarrow 'b \Rightarrow$ *bool* (**infix** ‹$\preceq$› *50*) **and** *less* (**infix** ‹$\prec$› *50*) +
fixes *E* :: $'a \Rightarrow 'a \Rightarrow$ *bool* — The full edge set
and s_0 :: $'a$ — Start state
fixes *RE* :: $'a \Rightarrow 'a \Rightarrow$ *bool* — Subgraph of the graph given by the full edge set
fixes *SE* :: $'a \Rightarrow 'a \Rightarrow$ *bool* — Subsumption edges
and *covered* :: $'a \Rightarrow$ *bool*
fixes *closure* :: $'a \Rightarrow 'b$
fixes *P* :: $'a \Rightarrow$ *bool*
fixes *Q* :: $'a \Rightarrow$ *bool*
begin

sublocale *Graph-Start-Defs E* s_0 **.**

sublocale *G*: *Graph-Start-Defs RE* s_0 **.**

end

locale *Reachability-Compatible-Subsumption-Graph-Closure-View* =
Subsumption-Graph-Closure-View-Defs +
preorder less-eq less +
assumes *mono*:
closure a $\preceq$ *closure b* $\Longrightarrow$ *E a a'* $\Longrightarrow$ *P a* $\Longrightarrow$ *P b* $\Longrightarrow$ $\exists$ *b'*. *E b b'* $\wedge$ *closure a'* $\preceq$ *closure b'*
assumes *closure-eq*:
closure a = *closure b* $\Longrightarrow$ *E a a'* $\Longrightarrow$ *P a* $\Longrightarrow$ *P b* $\Longrightarrow$ $\exists$ *b'*. *E b b'* $\wedge$ *closure a'* = *closure b'*
assumes *reachability-compatible*:
$\forall$ *s*. *Q s* $\longrightarrow$ (*if covered s then* ($\exists$ *t*. *SE s t* $\wedge$ *G.reachable t*) *else* ($\forall$ *s'*. *E s s'* $\longrightarrow$ *RE s s'*))
assumes *subsumption*: $\forall$ *s'*. *SE s s'* $\longrightarrow$ *closure s* $\prec$ *closure s'*
assumes *subgraph*: $\forall$ *s s'*. *RE s s'* $\longrightarrow$ *E s s'*
assumes *finite-closure*: *finite* (*closure* ‘ *UNIV*)
assumes *P-post*: *a* $\rightarrow$ *b* $\Longrightarrow$ *P b*
assumes *P-pre*: *a* $\rightarrow$ *b* $\Longrightarrow$ *P a*
assumes *P-*s_0: *P* s_0

assumes *Q-post*: $RE\ a\ b \Longrightarrow Q\ b$
assumes *Q-*s_0: $Q\ s_0$
begin

definition *close* **where** $close\ e\ a\ b = (\exists\ x\ y.\ e\ x\ y \wedge a = closure\ x \wedge b = closure\ y)$

lemma *Simulation-close*:
Simulation A (*close A*) ($\lambda\ a\ b.\ b = closure\ a$)
unfolding *close-def* **by** *standard auto*

sublocale *view*: *Reachability-Compatible-Subsumption-Graph*
($\preceq$) ($\prec$) *close E closure* s_0 *close RE*
supply [*simp*] = *close-def*
supply [*intro*] = *P-pre P-post Q-post*
proof (*standard*, *goal-cases*)
case *prems*: (*1 a b a'*)
then obtain *x y* **where** [*simp*]: $x \to y\ a = closure\ x\ a' = closure\ y$
by *auto*
then have *P x P y*
by *blast+*
from *prems*(*4*) *P-*s_0 **obtain** x' **where** [*simp*]: $b = closure\ x'\ P\ x'$
unfolding *Graph-Start-Defs.reachable-def* **by** *cases auto*
from *mono*[*OF* ‹- $\preceq$ -›[*simplified*] ‹$x \to y$› ‹*P x*› ‹*P x'*›] **obtain** b' **where**
$x' \to b'$ *closure y* $\preceq$ *closure b'*
by *auto*
then show *?case*
by *auto*
next
case *2*
interpret *Simulation RE close RE* $\lambda\ a\ b.\ b = closure\ a$
by (*rule Simulation-close*)
{ fix *x* **assume** *Graph-Start-Defs.reachable* (*close RE*) (*closure* s_0) *x*
then obtain x' **where** [*simp*]: $x = closure\ x'\ Q\ x'\ P\ x'$
using *Q-*s_0 *P-*s_0 *subgraph* **unfolding** *Graph-Start-Defs.reachable-def*
by *cases auto*
have ($\forall\ s'.\ close\ E\ x\ s' \longrightarrow close\ RE\ x\ s'$)
$\vee$ ($\exists\ t.\ x \prec t \wedge$ *Graph-Start-Defs.reachable* (*close RE*) (*closure* s_0) *t*)
proof (*cases covered x'*)
case *True*
with *reachability-compatible* ‹*Q x'*› **obtain** *t* **where** *SE x' t G.reachable*
t
by *fastforce*
then show *?thesis*

```
      using subsumption
    by − (rule disjI2, auto dest: simulation-reaches simp: Graph-Start-Defs.reachable-def)
  next
    case False
    with reachability-compatible ‹Q x'› have ∀ s'. x' → s' ⟶ RE x' s'
      by auto
    then show ?thesis
      unfolding close-def using closure-eq[OF - - - ‹P x'›] by − (rule
disjI1, force)
  qed
 }
 then show ?case
  by (intro allI impI)
next
 case 3
 then show ?case
  using subgraph by auto
next
 case 4
 have {a. Graph-Start-Defs.reachable (close RE) (closure s0) a} ⊆ closure
‘ UNIV
   by (smt Graph-Start-Defs.reachable-induct close-def full-SetCompr-eq
mem-Collect-eq subsetI)
 also have finite ...
  by (rule finite-closure)
 finally show ?case .
qed

end

locale Reachability-Compatible-Subsumption-Graph-Final = Reachability-Compatible-Subsumption-G
+
 fixes F :: 'a ⇒ bool — Final states
 assumes F-mono[intro]: F a ⟹ a ≼ b ⟹ F b

locale Liveness-Compatible-Subsumption-Graph = Reachability-Compatible-Subsumption-Graph-Fina
+
 assumes no-subsumption-cycle:
  G'.reachable x ⟹ x →G^+' x ⟹ x →G^+ x
```

2.3 Reachability

```
context Subsumption-Graph-Defs
begin
```

Setup for automation

context
  includes *graph-automation*
begin

lemma *G′-reachable-G-reachable*[*intro*]:
  G.reachable a **if** *G′.reachable a*
  using *that* **by** (*induction*; *blast*)

lemma *G-reachable-G′-reachable*[*intro*]:
  G′.reachable a **if** *G.reachable a*
  using *that* **by** (*induction*; *blast*)

lemma *G-G′-reachable-iff*:
  G.reachable a $\longleftrightarrow$ *G′.reachable a*
  by *blast*

end

end

context *Reachability-Compatible-Subsumption-Graph-Pre*
begin

lemmas *preorder-intros* = *order-trans less-trans less-imp-le*

lemma *G′-finite-reachable*: *finite* {*a*. *G′.reachable a*}
  by (*blast intro*: *finite-subset*[*OF* - *finite-reachable*])

lemma *G-reachable-has-surrogate*:
  $\exists$ *t*. *G.reachable t* $\wedge$ *s* $\preceq$ *t* $\wedge$ ($\forall$ *s′*. *E t s′* $\longrightarrow$ *RE t s′*) **if** *G.reachable s*
proof −
  note [*intro*] = *preorder-intros*
  from *finite-reachable* ‹*G.reachable s*› **obtain** *x* **where**
    $\forall$ *s′*. *E x s′* $\longrightarrow$ *RE x s′ G.reachable x* ((≺)**) *s x*
    apply *atomize-elim*
    apply (*induction rule*: *rtranclp-ev-induct2*)
    using *reachability-compatible* **by** *auto*
  moreover from ‹((≺)**) *s x*› **have** *s* ≺ *x* $\vee$ *s* = *x*
    by *induction auto*
  ultimately show *?thesis*
    by *auto*

qed

lemma *reachable-has-surrogate*:
  $\exists\ t.\ G.reachable\ t \wedge s \preceq t \wedge (\forall\ s'.\ E\ t\ s' \longrightarrow RE\ t\ s')$ **if** *reachable s*
  using *that*
proof *induction*
  case *start*
  have $G.reachable\ s_0$
    by *auto*
  then show *?case*
    by (*rule G-reachable-has-surrogate*)
next
  case (*step a b*)
  then obtain *t* **where** $*$: $G.reachable\ t\ a \preceq t\ (\forall\, s'.\ t \to s' \longrightarrow t \to_G s')$
    by *auto*
  from *mono*[*OF* ‹$a \preceq t$› ‹$a \to b$›] ‹*reachable a*› ‹*G.reachable t*› **obtain** b'
where
    $t \to b'\ b \preceq b'$
    by *auto*
  with *G-reachable-has-surrogate*[*of b'*] $*$ **show** *?case*
    by (*auto intro*: *preorder-intros G.reachable-step*)
qed

context
  fixes $F :: 'a \Rightarrow bool$ — Final states
  assumes *F-mono*[*intro*]: $F\ a \Longrightarrow a \preceq b \Longrightarrow F\ b$
begin

corollary *reachability-correct*:
  $\nexists\ s'.\ reachable\ s' \wedge F\ s'$ **if** $\nexists\ s'.\ G.reachable\ s' \wedge F\ s'$
  using *that* **by** (*auto dest*!: *reachable-has-surrogate*)

end

end

context *Reachability-Compatible-Subsumption-Graph*
begin

Setup for automation

context
  includes *graph-automation*
begin

lemma *subgraph'*[*intro*]:
  $E\ s\ s'$ **if** $RE\ s\ s'$
  using *that subgraph* **by** *blast*

lemma *G-reachability-sound*[*intro*]:
  reachable a **if** *G.reachable a*
  using *that* **by** (*induction*; *blast*)

lemma *G-steps-sound*[*intro*]:
  steps xs **if** *G.steps xs*
  using *that* **by** (*induction*; *blast*)

lemma *G-run-sound*[*intro*]:
  run xs **if** *G.run xs*
  using *that* **by** (*coinduction arbitrary*: *xs*) (*auto 4 3 elim*: *G.run.cases*)

lemma *G'-reachability-sound*[*intro*]:
  reachable a **if** *G'.reachable a*
  using *that* **by** (*induction*; *blast*)

lemma *G'-finite-reachable*: *finite* {*a*. *G'.reachable a*}
  by (*blast intro*: *finite-subset*[*OF - finite-reachable*])

lemma *G-steps-G'-steps*[*intro*]:
  G'.steps as **if** *G.steps as*
  using *that* **by** *induction auto*

lemma *reachable-has-surrogate*:
  $\exists\ t.\ G.reachable\ t \wedge s \preceq t \wedge (\forall\ s'.\ E\ t\ s' \longrightarrow RE\ t\ s')$ **if** *G.reachable s*
proof −
  note [*intro*] = *preorder-intros*
  from *finite-reachable* ‹*G.reachable s*› **obtain** *x* **where**
    $\forall\, s'.\ E\ x\ s' \longrightarrow RE\ x\ s'\ G.reachable\ x\ ((\prec)^{**})\ s\ x$
    apply *atomize-elim*
    apply (*induction rule*: *rtranclp-ev-induct2*)
    using *reachability-compatible* **by** *auto*
  moreover from ‹$((\prec)^{**})\ s\ x$› **have** $s \prec x \vee s = x$
    by *induction auto*
  ultimately show *?thesis*
    by *auto*
qed

lemma *reachable-has-surrogate'*:

$\exists\ t.\ s \preceq t \wedge s \rightarrow_G{*}'\ t \wedge (\forall\ s'.\ E\ t\ s' \longrightarrow RE\ t\ s')$ **if** *G.reachable s*
proof −
note [*intro*] = *preorder-intros*
from ‹*G.reachable s*› **have** ‹*G.reachable s*› **by** *auto*
from *finite-reachable this* **obtain** *x* **where**
real-edges: $\forall\ s'.\ E\ x\ s' \longrightarrow RE\ x\ s'$ **and** *G.reachable x* $((\prec)^{**})$ *s x*
apply *atomize-elim*
apply (*induction rule*: *rtranclp-ev-induct2*)
using *reachability-compatible* **by** *auto*
from ‹$((\prec)^{**})$ *s x*› **have** $s \prec x \vee s = x$
by *induction auto*
then show *?thesis*
proof
assume $s \prec x$
with *real-edges* ‹*G.reachable x*› **show** *?thesis*
by (*inst-existentials x*) *auto*
next
assume $s = x$
with *real-edges* **show** *?thesis*
by (*inst-existentials s*) *auto*
qed
qed

lemma *subsumption-step*:
$\exists\ a''\ b'.\ a' \preceq a'' \wedge b \preceq b' \wedge a'' \rightarrow_G b' \wedge$ *G.reachable a''* **if**
reachable a E a b G.reachable a' a $\preceq a'$
proof −
note [*intro*] = *preorder-intros*
from *mono*[*OF* ‹$a \preceq a'$› ‹*E a b*› ‹*reachable a*›] ‹*G.reachable a'*› **obtain** b' **where** *E a' b' b* $\preceq b'$
by *auto*
from *reachable-has-surrogate*[*OF* ‹*G.reachable a'*›] **obtain** a''
where $a' \preceq a''$ *G.reachable a''* **and** $*$: $\forall\ s'.\ E\ a''\ s' \longrightarrow RE\ a''\ s'$
by *auto*
from *mono*[*OF* ‹$a' \preceq a''$› ‹*E a' b'*›] ‹*G.reachable a'*› ‹*G.reachable a''*› **obtain** b'' **where**
E a'' b'' b' $\preceq b''$
by *auto*
with $*$ ‹$a' \preceq a''$› ‹$b \preceq b'$› ‹*G.reachable a''*› **show** *?thesis*
by *auto*
qed

lemma *subsumption-step′*:
$\exists\ b'.\ b \preceq b' \wedge a' \rightarrow_G{}^{+}{}'\ b'$ **if** *reachable a a* $\rightarrow$ *b G′.reachable a′ a* $\preceq a'$

proof −
note [*intro*] = *preorder-intros*
from *mono*[*OF* ‹$a \preceq a'$› ‹*E a b*› ‹*reachable a*›] ‹G'.*reachable* a'› **obtain** b' **where**
$b \preceq b'\ a' \to b'$
by *auto*
from *reachable-has-surrogate'*[*of* a'] ‹G'.*reachable* a'› **obtain** a'' **where** *:
$a' \preceq a''\ a' \to_G{*}'\ a''\ \forall\, s'.\ a'' \to s' \longrightarrow a'' \to_G s'$
by *auto*
with ‹G'.*reachable* a'› **have** G'.*reachable* a''
by *blast*
with *mono*[*OF* ‹$a' \preceq a''$› ‹*E a' b'*›] ‹G'.*reachable* a'› **obtain** b'' **where**
$b' \preceq b''\ a'' \to b''$
by *auto*
with $*$ ‹$b \preceq b'$› ‹$b' \preceq b''$› ‹G'.*reachable* a''› **show** *?thesis*
by (*auto simp*: G'.*reaches1-reaches-iff2*)
qed

theorem *reachability-complete'*:
$\exists\ s'.\ s \preceq s' \wedge$ *G.reachable* s' **if** $a \to{*}\ s$ *G.reachable a*
using *that*
proof (*induction*)
case *base*
then show *?case* **by** *auto*
next
case (*step s t*)
then obtain s' **where** $s \preceq s'$ *G.reachable* s'
by *auto*
with *step*(*4*) **have** *reachable a G.reachable* s'
by *auto*
with *step*(*1*) **have** *reachable s*
by *auto*
from *subsumption-step*[*OF* ‹*reachable s*› ‹*E s t*› ‹*G.reachable* s'› ‹$s \preceq s'$›]
obtain $s''\ t'$ **where**
$s' \preceq s''\ t \preceq t'\ s'' \to_G t'$ *G.reachable* s''
by *atomize-elim*
with ‹*G.reachable* s'› **show** *?case*
by *auto*
qed

corollary *reachability-complete*:
$\exists\ s'.\ s \preceq s' \wedge$ *G.reachable* s' **if** *reachable s*
using *reachability-complete'*[*of* s_0 *s*] *that* **unfolding** *reachable-def* **by** *auto*

corollary *reachability-correct*:
(∃ s'. $s \preceq s' \wedge$ *reachable* s') $\longleftrightarrow$ (∃ s'. $s \preceq s' \wedge$ *G.reachable* s')
by (*blast dest*: *reachability-complete intro*: *preorder-intros*)

lemma *steps-G'-steps*:
∃ *ys ns*. *list-all2* (⪯) *xs* (*nths ys ns*) ∧ *G'.steps* (*b* # *ys*) **if**
steps (*a* # *xs*) *reachable a a* ⪯ *b G'.reachable b*
using *that*
proof (*induction a* # *xs arbitrary*: *a b xs*)
case (*Single*)
then show *?case* **by** *force*
next
case (*Cons x y xs*)
from *subsumption-step'*[*OF* ‹*reachable x*› ‹*E x y*› - ‹*x* ⪯ *b*›] ‹*G'.reachable b*› **obtain** *b'* **where**
y ⪯ *b' b* $\rightarrow_G{}^{+\prime}$ *b'*
by *auto*
with ‹*reachable x*› *Cons.hyps*(*1*) *Cons.prems*(*3*) **obtain** *ys ns* **where**
list-all2 (⪯) *xs* (*nths ys ns*) *G'.steps* (*b'* # *ys*)
by *atomize-elim* (*blast intro*: *Cons.hyps*(*3*)[*OF* - ‹*y* ⪯ *b'*›] *intro*: *graphI-aggressive*)
from ‹*b* $\rightarrow_G{}^{+\prime}$ *b'*› *this*(*2*) **obtain** *as* **where**
G'.steps (*b* # *as* @ *b'* # *ys*)
by (*fastforce intro*: *G'.graphI-aggressive1*)
with ‹*y* ⪯ *b'*› **show** *?case*
apply (*inst-existentials as* @ *b'* # *ys* {*length as*} ∪ {*n* + *length as* + *1* | *n*. *n* ∈ *ns*})
subgoal
apply (*subst nths-split*, *force*)
apply (*subst nths-nth*, (*simp*; *fail*))
apply *simp*
apply (*subst nths-shift*, *force*)
subgoal premises *prems*
proof −
have
{*x* − *length as* |*x*. *x* ∈ {*Suc* (*n* + *length as*) |*n*. *n* ∈ *ns*}} = {*n* + *1* | *n*. *n* ∈ *ns*}
by *force*
with ‹*list-all2* - - -› **show** *?thesis*
by (*simp add*: *nths-Cons*)
qed
done
by *assumption*
qed

lemma *cycle-G′-cycle″*:
assumes *steps* (s_0 # *ws* @ *x* # *xs* @ [*x*])
shows ∃ *x′ xs′ ys′*. *x* ⪯ *x′* ∧ *G′.steps* (s_0 # *xs′* @ *x′* # *ys′* @ [*x′*])
proof −
let *?n* = *card* {*x*. *G′.reachable x*} + *1*
let *?xs* = *x* # *concat* (*replicate ?n* (*xs* @ [*x*]))
from *assms*(*1*) **have** *steps* (*x* # *xs* @ [*x*])
by (*auto intro*: *graphI-aggressive2*)
with *steps-replicate*[*of x* # *xs* @ [*x*] *?n*] **have** *steps ?xs*
by *auto*
have *steps* (s_0 # *ws* @ *?xs*)
proof −
from *assms* **have** *steps* (s_0 # *ws* @ [*x*])
by (*auto intro*: *graphI-aggressive2*)
with ‹*steps ?xs*› **show** *?thesis*
by (*fastforce intro*: *graphI-aggressive1*)
qed
from *steps-G′-steps*[*OF this*, *of* s_0] **obtain** *ys ns* **where** *ys*:
list-all2 (⪯) (*ws* @ *x* # *concat* (*replicate ?n* (*xs* @ [*x*]))) (*nths ys ns*)
G′.steps (s_0 # *ys*)
by *auto*
then obtain *x′ ys′ ns′ ws′* **where** *ys′*:
G′.steps (*x′* # *ys′*) *G′.steps* (s_0 # *ws′* @ [*x′*])
list-all2 (⪯) (*concat* (*replicate ?n* (*xs* @ [*x*]))) (*nths ys′ ns′*)
apply *atomize-elim*
apply *clarsimp*
apply (*subst* (*asm*) *list-all2-append1*)
apply *safe*
apply (*subst* (*asm*) *list-all2-Cons1*)
apply *safe*
apply (*drule nths-eq-appendD*)
apply *safe*
apply (*drule nths-eq-ConsD*)
apply *safe*
subgoal for *ys1 ys2 z ys3 ys4 ys5 ys6 ys7 i*
apply (*inst-existentials z ys7*)
subgoal premises *prems*
using *prems*(*1*) **by** (*auto intro*: *G′.graphI-aggressive2*)
subgoal premises *prems*
proof −
from *prems* **have** *G′.steps* ((s_0 # *ys4* @ *ys6* @ [*z*]) @ *ys7*)
by *auto*
moreover then have *G′.steps* (s_0 # *ys4* @ *ys6* @ [*z*])

```
        by (auto intro: G'.graphI-aggressive2)
      ultimately show ?thesis
        by (inst-existentials ys4 @ ys6) auto
    qed
    by force
  done
let ?ys = filter ((≼) x) ys'
have length ?ys ≥ ?n
  using list-all2-replicate-elem-filter[OF ys'(3), of x]
  using filter-nths-length[of ((≼) x) ys' ns']
  by auto
from ‹G'.steps (s₀ # ws' @ [x'])› have G'.reachable x'
  by − (rule G'.reachable-reaches, auto)
have set ?ys ⊆ set ys'
  by auto
also have ... ⊆ {x. G'.reachable x}
  using ‹G'.steps (x' # -)› ‹G'.reachable x'›
  by clarsimp (rule G'.reachable-steps-elem[rotated], assumption, auto)
finally have ¬ distinct ?ys
  using distinct-card[of ?ys] ‹- >= ?n›
  by − (rule ccontr; drule distinct-length-le[OF G'-finite-reachable]; simp)
from not-distinct-decomp[OF this] obtain as y bs cs where ?ys = as @
[y] @ bs @ [y] @ cs
  by auto
then obtain as' bs' cs' where
  ys' = as' @ [y] @ bs' @ [y] @ cs'
  apply atomize-elim
  apply simp
 apply (drule filter-eq-appendD filter-eq-ConsD filter-eq-appendD[OF sym],
clarify)+
  apply clarsimp
  subgoal for as1 as2 bs1 bs2 cs'
    by (inst-existentials as1 @ as2 bs1 @ bs2) simp
  done
have G'.steps (y # bs' @ [y])
proof −
  from ‹G'.steps (x' # -)› ‹ys' = -› show ?thesis
    by (force intro: G'.graphI-aggressive2)
qed
moreover have G'.steps (s₀ # ws' @ x' # as' @ [y])
proof −
  from ‹G'.steps (x' # ys')› ‹ys' = -› have G'.steps (x' # as' @ [y])
    by (force intro: G'.graphI-aggressive2)
  with ‹G'.steps (s₀ # ws' @ [x'])› show ?thesis
```

```
    by (fastforce intro: G'.graphI-aggressive1)
  qed
  moreover from ‹?ys = -›  have x ⪯ y
  proof −
    from ‹?ys = -› have y ∈ set ?ys by auto
    then show ?thesis by auto
  qed
  ultimately show ?thesis
   by (inst-existentials y ws' @ x' # as' bs'; fastforce intro: G'.graphI-aggressive1)
qed

lemma cycle-G'-cycle':
  assumes steps (s₀ # ws @ x # xs @ [x])
  shows ∃ y ys. x ⪯ y ∧ G'.steps (y # ys @ [y]) ∧ G'.reachable y
proof −
  from cycle-G'-cycle''[OF assms] obtain x' xs' ys' where
    x ⪯ x' G'.steps (s₀ # xs' @ x' # ys' @ [x'])
    by auto
  then show ?thesis
    apply (inst-existentials x' ys')
    subgoal by assumption
    subgoal by (auto intro: G'.graphI-aggressive2)
    by (rule G'.reachable-reaches, auto intro: G'.graphI-aggressive2)
qed

lemma cycle-G'-cycle:
  assumes reachable x x →⁺ x
  shows ∃ y ys. x ⪯ y ∧ G'.reachable y ∧ y →_G⁺' y
proof −
  from assms(2) obtain xs where ∗: steps (x # xs @ x # xs @ [x])
    by (fastforce intro: graphI-aggressive1)
  from reachable-steps[of x] assms(1) obtain ws where steps ws hd ws =
s₀ last ws = x
    by auto
  with ∗ obtain us where steps (s₀ # (us @ xs) @ x # xs @ [x])
    by (cases ws; force intro: graphI-aggressive1)
  from cycle-G'-cycle'[OF this] show ?thesis
    by (auto intro: G'.graphI-aggressive2)
qed

corollary G'-reachability-complete:
  ∃ s'. s ⪯ s' ∧ G.reachable s' if G'.reachable s
  using reachability-complete that by auto
```

end

end

corollary (**in** *Reachability-Compatible-Subsumption-Graph-Final*) *reachability-correct*:
(∃ *s'*. *reachable s'* ∧ *F s'*) ⟷ (∃ *s'*. *G.reachable s'* ∧ *F s'*)
using *reachability-complete* **by** *blast*

2.4 Liveness

theorem (**in** *Liveness-Compatible-Subsumption-Graph*) *cycle-iff*:
(∃ *x*. *x* →⁺ *x* ∧ *reachable x* ∧ *F x*) ⟷ (∃ *x*. *x* →$_G$⁺ *x* ∧ *G.reachable x* ∧ *F x*)
by (*auto 4 4 intro*: *no-subsumption-cycle steps-reaches1 dest*: *cycle-G'-cycle G.graphD*)

2.5 Appendix

context *Subsumption-Graph-Pre-Nodes*
begin

Setup for automation

context
includes *graph-automation*
begin

lemma *steps-mono*:
assumes *G'.steps* (*x* # *xs*) *x* ⪯ *y V x V y*
shows ∃ *ys*. *G'.steps* (*y* # *ys*) ∧ *list-all2* (⪯) *xs ys*
using *assms* **including** *subgraph-automation*
proof (*induction x* # *xs arbitrary*: *x y xs*)
case (*Single x*)
then show *?case* **by** *auto*
next
case (*Cons x y xs x'*)
from *mono*[*OF* ‹*x* ⪯ *x'*›] ‹*x* → *y*› *Cons.prems* **obtain** *y'* **where** *x'* → *y'* *y* ⪯ *y'*
by *auto*
with *Cons.hyps*(*3*)[*OF* ‹*y* ⪯ *y'*›] ‹*x* → *y*› *Cons.prems* **obtain** *ys* **where** *G'.steps* (*y'* # *ys*) *list-all2* (⪯) *xs ys*
by *auto*
with ‹*x'* → *y'*› ‹*y* ⪯ *y'*› **show** *?case*
by *auto*

qed

lemma *steps-append-subsumption*:
assumes $G'.steps\ (x \# xs)\ G'.steps\ (y \# ys)\ y \preceq last\ (x \# xs)\ V\ x\ V\ y$
shows $\exists\ ys'.\ G'.steps\ (x \# xs\ @\ ys')\ \wedge\ list\text{-}all2\ (\preceq)\ ys\ ys'$
proof −
from *assms* **have** $V\ (last\ (x \# xs))$
by − (*rule G′-steps-V-last*, *auto*)
from *steps-mono*[*OF* ‹$G'.steps\ (y \# ys)$› ‹$y \preceq$ -› ‹$V\ y$› *this*] **obtain** ys'
where
$G'.steps\ (last\ (x \# xs) \# ys')\ list\text{-}all2\ (\preceq)\ ys\ ys'$
by *auto*
with *G′.steps-append*[*OF* ‹$G'.steps\ (x \# xs)$› *this*(*1*)] **show** *?thesis*
by *auto*
qed

lemma *steps-replicate-subsumption*:
assumes $x \preceq last\ (x \# xs)\ G'.steps\ (x \# xs)\ n > 0\ V\ x$
notes [*intro*] = *preorder-intros*
shows $\exists\ ys.\ G'.steps\ (x \# ys) \wedge list\text{-}all2\ (\preceq)\ (concat\ (replicate\ n\ xs))\ ys$
using *assms*
proof (*induction n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
show *?case*
proof (*cases n*)
case *0*
with *Suc.prems* **show** *?thesis*
by (*inst-existentials xs*) (*auto intro*: *list-all2-refl*)
next
case *prems*: (*Suc n′*)
with *Suc* ‹n = -› **obtain** *ys* **where** *ys*:
$list\text{-}all2\ (\preceq)\ (concat\ (replicate\ n\ xs))\ ys\ G'.steps\ (x \# ys)$
by *auto*
with ‹n = -› **have** $list\text{-}all2\ (\preceq)\ (concat\ (replicate\ n'\ xs)\ @\ xs)\ ys$
by (*metis append-Nil2 concat.simps*(*1*,*2*) *concat-append replicate-Suc replicate-append-same*)
with ‹$x \preceq$ -› **have** $x \preceq last\ (x \# ys)$
by (*cases xs*; *auto 4 3 dest*: *list-all2-last split*: *if-split-asm*)
from *steps-append-subsumption*[*OF* ‹$G'.steps\ (x \# ys)$› ‹$G'.steps\ (x \# xs)$› *this*] ‹$V\ x$› **obtain**
ys' **where** $G'.steps\ (x \# ys\ @\ ys')\ list\text{-}all2\ (\preceq)\ xs\ ys'$

```
    by auto
  with ys(1) ‹n = -› show ?thesis
    apply (inst-existentials ys @ ys')
    by auto
      (metis
          append-Nil2 concat.simps(1,2) concat-append list-all2-appendI
replicate-Suc
        replicate-append-same
      )
 qed
qed

context
 assumes finite-V: finite {x. V x}
begin

lemma wf-less-on-reachable-set:
 assumes antisym: ⋀ x y. x ⪯ y ⟹ y ⪯ x ⟹ x = y
 shows wf {(x, y). y ≺ x ∧ V x ∧ V y} (is wf ?S)
proof (rule finite-acyclic-wf)
 have ?S ⊆ {(x, y). V x ∧ V y}
   by auto
 also have finite ...
   using finite-V by auto
 finally show finite ?S .
next
 interpret order by unfold-locales (rule antisym)
 show acyclicP (λx y. y ≺ x ∧ V x ∧ V y)
   by (rule acyclicI-order[where f = id]) auto
qed
```

This shows that looking for cycles and pre-cycles is equivalent in monotone subsumption graphs.

```
lemma pre-cycle-cycle':
 assumes A: x ⪯ x' G'.steps (x # xs @ [x']) V x
 shows ∃ x'' ys. x' ⪯ x'' ∧ G'.steps (x'' # ys @ [x'']) ∧ V x''
proof −
 let ?n  = card {x. V x} + 1
 let ?xs = concat (replicate ?n (xs @ [x']))
 from steps-replicate-subsumption[OF - ‹G'.steps -›, of ?n] ‹V x› ‹x ⪯ x'›
obtain ys where
   G'.steps (x # ys) list-all2 (⪯) ?xs ys
   by auto
 let ?ys = filter ((⪯) x') ys
```

```
  have length ?ys ≥ ?n
    using list-all2-replicate-elem-filter[OF ‹list-all2 (⪯) ?xs ys›, of x']
    by auto
  have set ?ys ⊆ set ys
    by auto
  also have ... ⊆ {x. V x}
   using G'-steps-V-all[OF ‹G'.steps (x # ys)›] ‹V x› unfolding list-all-iff
by auto
  finally have ¬ distinct ?ys
    using distinct-card[of ?ys] ‹- >= ?n›
    by − (rule ccontr, drule distinct-length-le[OF finite-V], auto)
  from not-distinct-decomp[OF this] obtain as y bs cs where ?ys = as @
[y] @ bs @ [y] @ cs
    by auto
  then obtain as' bs' cs' where
    ys = as' @ [y] @ bs' @ [y] @ cs'
    apply atomize-elim
    apply simp
   apply (drule filter-eq-appendD filter-eq-ConsD filter-eq-appendD[OF sym],
clarify)+
    apply clarsimp
    subgoal for as1 as2 bs1 bs2 cs'
      by (inst-existentials as1 @ as2 bs1 @ bs2) simp
    done
  have G'.steps (y # bs' @ [y])
  proof −
    from ‹G'.steps (x # ys)› ‹ys = -› have G'.steps (x # as' @ (y # bs' @
[y]) @ cs')
      by auto
    then show ?thesis
      by − ((simp; fail) | drule G'.stepsD)+
  qed
  moreover have V y
  proof −
    from ‹G'.steps (x # ys)› ‹ys = -› have G'.steps ((x # as' @ [y]) @ (bs'
@ y # cs'))
      by simp
    then have G'.steps (x # as' @ [y])
      by (blast dest: G'.stepsD)
    with ‹V x› show ?thesis
      by (auto dest: G'-steps-V-last)
  qed
  moreover from ‹?ys = -› have x' ⪯ y
  proof −
```

from ‹*?ys* = -› **have** *y* ∈ *set ?ys* **by** *auto*
then show *?thesis* **by** *auto*
qed
ultimately show *?thesis*
by *auto*
qed

lemma *pre-cycle-cycle*:
(∃ *x x'*. *V x* ∧ *x* →$^+$ *x'* ∧ *x* ≼ *x'*) ⟷ (∃ *x*. *V x* ∧ *x* →$^+$ *x*)
including *reaches-steps-iff* **by** (*force dest*: *pre-cycle-cycle'*)

lemma *pre-cycle-cycle-reachable*:
(∃ *x x'*. a_0 →* *x* ∧ *V x* ∧ *x* →$^+$ *x'* ∧ *x* ≼ *x'*) ⟷ (∃ *x*. a_0 →* *x* ∧ *V x* ∧ *x* →$^+$ *x*)
proof −
interpret *interp*: *Subsumption-Graph-Pre-Nodes* - - *E* λ *x*. a_0 →* *x* ∧ *V x*
including *graph-automation-aggressive*
by *standard* (*drule mono*, *auto 4 3 simp*: *Subgraph-Node-Defs.E'-def E'-def*)
interpret *start*: *Graph-Start-Defs E'* a_0 **.**
have ∗: *start.reachable-subgraph.E'* = *interp.E'*
unfolding *interp.E'-def start.reachable-subgraph.E'-def*
unfolding *start.reachable-def E'-def*
by *auto*
have ∗: *start.reachable-subgraph.G'.reaches1* = *interp.G'.reaches1*
unfolding *tranclp-def* ∗ **..**
have ∗: *interp.G'.reaches1 x y* ⟷ *x* →$^+$ *y* **if** a_0 →* *x* **for** *x y*
using *start.reachable-reaches1-equiv*[*of x y*] *that* **unfolding** ∗ **by** (*simp add*: *start.reachable-def*)
from *interp.pre-cycle-cycle finite-V* **show** *?thesis*
by (*auto simp*: ∗)
qed

end

end

end

context *Subsumption-Graph-Pre*
begin

Setup for automation

context
includes *graph-automation*
begin

interpretation *Subsumption-Graph-Pre-Nodes - - E reachable*
apply *standard*
apply (*drule mono*)
apply (*simp-all add*: *Subgraph-Node-Defs.E'-def*)
apply *force*
by *auto*

lemma *steps-mono*:
assumes *steps* (*x* # *xs*) *x* ⪯ *y reachable x reachable y*
shows ∃ *ys*. *steps* (*y* # *ys*) ∧ *list-all2* (⪯) *xs ys*
using *assms steps-mono* **by** (*simp add*: *reachable-steps-equiv*)

lemma *steps-append-subsumption*:
assumes *steps* (*x* # *xs*) *steps* (*y* # *ys*) *y* ⪯ *last* (*x* # *xs*) *reachable x reachable y*
shows ∃ *ys'*. *steps* (*x* # *xs* @ *ys'*) ∧ *list-all2* (⪯) *ys ys'*
using *assms steps-append-subsumption* **by** (*simp add*: *reachable-steps-equiv*)

lemma *steps-replicate-subsumption*:
assumes *x* ⪯ *last* (*x* # *xs*) *steps* (*x* # *xs*) *n* > *0 reachable x*
notes [*intro*] = *preorder-intros*
shows ∃ *ys*. *steps* (*x* # *ys*) ∧ *list-all2* (⪯) (*concat* (*replicate n xs*)) *ys*
using *assms steps-replicate-subsumption* **by** (*simp add*: *reachable-steps-equiv*)

context
assumes *finite-reachable*: *finite* {*x*. *reachable x*}
begin

lemma *wf-less-on-reachable-set*:
assumes *antisym*: ⋀ *x y*. *x* ⪯ *y* ⟹ *y* ⪯ *x* ⟹ *x* = *y*
shows *wf* {(*x*, *y*). *y* ≺ *x* ∧ *reachable x* ∧ *reachable y*} (**is** *wf ?S*)
proof (*rule finite-acyclic-wf*)
have *?S* ⊆ {(*x*, *y*). *reachable x* ∧ *reachable y*}
by *auto*
also have *finite* ...
using *finite-reachable* **by** *auto*
finally show *finite ?S* **.**
next

interpret *order* **by** *standard* (*rule antisym*)
show *acyclicP* ($\lambda x\ y.\ y \prec x \wedge$ *reachable* $x \wedge$ *reachable* y)
by (*rule acyclicI-order*[**where** $f = id$]) *auto*
qed

This shows that looking for cycles and pre-cycles is equivalent in monotone subsumption graphs.

lemma *pre-cycle-cycle'*:
assumes *A*: $x \preceq x'$ *steps* ($x \# xs$ @ $[x']$) *reachable* x
shows $\exists\ x''\ ys.\ x' \preceq x'' \wedge$ *steps* ($x'' \# ys$ @ $[x'']$) $\wedge$ *reachable* x''
using *assms pre-cycle-cycle'*[*OF finite-reachable*] *reachable-steps-equiv* **by** *meson*

lemma *pre-cycle-cycle*:
($\exists\ x\ x'.$ *reachable* $x \wedge$ *reaches* $x\ x' \wedge x \preceq x'$) $\longleftrightarrow$ ($\exists\ x.$ *reachable* $x \wedge$ *reaches* $x\ x$)
including *reaches-steps-iff* **by** (*force dest*: *pre-cycle-cycle'*)

end

end

end

context *Subsumption-Graph-Defs*
begin

sublocale *G''*: *Graph-Start-Defs* $\lambda\ x\ y.\ \exists\ z.$ *G.reachable* $z \wedge x \preceq z \wedge$ *RE* $z\ y\ s_0$.

lemma *G''-reachable-G'*[*intro*]:
G'.reachable x **if** *G''.reachable* x
using *that*
unfolding *G'.reachable-def G''.reachable-def G-G'-reachable-iff Graph-Start-Defs.reachable-def*
proof (*induction*)
case *base*
then show *?case*
by *blast*
next
case (*step* $y\ z$)
then obtain z' **where**
$RE^{**}\ s_0\ z'\ y \preceq z'\ RE\ z'\ z$
by *auto*

from *this*(*1*) **have** $(\lambda x\ y.\ RE\ x\ y \vee x \prec y \wedge RE^{**}\ s_0\ y)^{**}\ s_0\ z'$
by (*induction*; *blast intro*: *rtranclp.intros*(*2*))
with ‹*RE z' z*› **show** *?case*
by (*blast intro*: *rtranclp.intros*(*2*))
qed

end

locale *Reachability-Compatible-Subsumption-Graph-Total* = *Reachability-Compatible-Subsumption-G*
+
assumes *total*: *reachable a* $\Longrightarrow$ *reachable b* $\Longrightarrow$ $a \preceq b \vee b \preceq a$
begin

sublocale *G''-pre*: *Subsumption-Graph-Pre* $(\preceq)$ $(\prec)$ $\lambda\ x\ y.\ \exists\ z.$ *G.reachable*
$z \wedge x \preceq z \wedge RE\ z\ y$
proof (*standard*, *safe*, *goal-cases*)
case *prems*: (*1 a b a' z*)
show *?case*
proof (*cases* $b \preceq z$)
case *True*
with *prems* **show** *?thesis*
by *auto*
next
case *False*
with *total*[*of b z*] *prems* **have** $z \preceq b$
by *auto*
with *subsumption-step*[*of z a' b*] *prems* **obtain** $a''\ b'$ **where**
$b \preceq a''\ a' \preceq b'$ *RE a'' b' G.reachable a''*
by *auto*
then show *?thesis*
by (*inst-existentials b' a''*) *auto*
qed
qed

end

2.6 Old Material

locale *Reachability-Compatible-Subsumption-Graph'* = *Subsumption-Graph-Defs*
+ *order* $(\preceq)$ $(\prec)$ +
assumes *reachability-compatible*:
$\forall\ s.$ *G.reachable s* $\longrightarrow$ $(\forall\ s'.\ E\ s\ s' \longrightarrow RE\ s\ s') \vee (\exists\ t.\ s \prec t \wedge$
G.reachable t)
assumes *subgraph*: $\forall\ s\ s'.\ RE\ s\ s' \longrightarrow E\ s\ s'$

assumes *finite-reachable*: *finite* $\{a.\ G.reachable\ a\}$
assumes *mono*:
$a \preceq b \Longrightarrow E\ a\ a' \Longrightarrow reachable\ a \Longrightarrow G.reachable\ b \Longrightarrow \exists\ b'.\ E\ b\ b' \wedge a' \preceq b'$
begin

Setup for automation

context
includes *graph-automation*
notes [*intro*] = *order.trans*
begin

lemma *subgraph'*[*intro*]:
$E\ s\ s'$ **if** $RE\ s\ s'$
using *that subgraph* **by** *blast*

lemma *G-reachability-sound*[*intro*]:
reachable a **if** *G.reachable a*
using *that* **unfolding** *reachable-def G.reachable-def* **by** (*induction*; *blast intro*: *rtranclp.intros*(*2*))

lemma *G-steps-sound*[*intro*]:
steps xs **if** *G.steps xs*
using *that* **by** *induction auto*

lemma *G-run-sound*[*intro*]:
run xs **if** *G.run xs*
using *that* **by** (*coinduction arbitrary*: *xs*) (*auto 4 3 elim*: *G.run.cases*)

lemma *reachable-has-surrogate*:
$\exists\ t.\ G.reachable\ t \wedge s \preceq t \wedge (\forall\ s'.\ E\ t\ s' \longrightarrow RE\ t\ s')$ **if** *G.reachable s*
using *that*
proof −
from *finite-reachable* ‹*G.reachable s*› **obtain** *x* **where**
$\forall\, s'.\ E\ x\ s' \longrightarrow RE\ x\ s'\ G.reachable\ x\ ((\prec)^{**})\ s\ x$
apply *atomize-elim*
apply (*induction rule*: *rtranclp-ev-induct2*)
using *reachability-compatible* **by** *auto*
moreover from ‹$((\prec)^{**})\ s\ x$› **have** $s \prec x \vee s = x$
by *induction auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *subsumption-step*:

$\exists\ a''\ b'.\ a' \preceq a'' \wedge b \preceq b' \wedge RE\ a''\ b' \wedge G.reachable\ a''$ **if**
reachable a E a b G.reachable a' a $\preceq a'$
proof −
from *mono*[*OF* ‹$a \preceq a'$› ‹*E a b*› ‹*reachable a*› ‹*G.reachable a'*›] **obtain** b' **where** *E a' b' b* $\preceq b'$
by *auto*
from *reachable-has-surrogate*[*OF* ‹*G.reachable a'*›] **obtain** a''
where $a' \preceq a''$ *G.reachable a''* **and** $*$: $\forall\ s'.\ E\ a''\ s' \longrightarrow RE\ a''\ s'$
by *auto*
from *mono*[*OF* ‹$a' \preceq a''$› ‹*E a' b'*›] ‹*G.reachable a'*› ‹*G.reachable a''*› **obtain** b'' **where**
E a'' b'' b' $\preceq b''$
by *auto*
with $*$ ‹$a' \preceq a''$› ‹$b \preceq b'$› ‹*G.reachable a''*› **show** *?thesis* **by** *auto*
qed

theorem *reachability-complete'*:
$\exists\ s'.\ s \preceq s' \wedge G.reachable\ s'$ **if** $E^{**}\ a\ s$ *G.reachable a*
using *that*
proof (*induction*)
case *base*
then show *?case* **by** *auto*
next
case (*step s t*)
then obtain s' **where** $s \preceq s'$ *G.reachable s'*
by *auto*
with *step*(*4*) **have** *reachable a G.reachable s'*
by *auto*
with *step*(*1*) **have** *reachable s*
by (*auto simp*: *reachable-def*)
from *subsumption-step*[*OF* ‹*reachable s*› ‹*E s t*› ‹*G.reachable s'*› ‹$s \preceq s'$›]
obtain $s''\ t'$ **where**
$s' \preceq s''\ t \preceq t'\ s'' \rightarrow_G t'$ *G.reachable s''*
by *atomize-elim*
with ‹*G.reachable s'*› **show** *?case*
by (*auto simp*: *reachable-def*)
qed

corollary *reachability-complete*:
$\exists\ s'.\ s \preceq s' \wedge G.reachable\ s'$ **if** *reachable s*
using *reachability-complete'*[*of* s_0 *s*] *that* **unfolding** *reachable-def* **by** *auto*

corollary *reachability-correct*:
$(\exists\ s'.\ s \preceq s' \wedge reachable\ s') \longleftrightarrow (\exists\ s'.\ s \preceq s' \wedge G.reachable\ s')$

using *reachability-complete* **by** *blast*

lemma *G′-reachability-sound*[*intro*]:
reachable a **if** *G′.reachable a*
using *that* **by** *induction auto*

corollary *G′-reachability-complete*:
$\exists\ s'.\ s \preceq s' \wedge$ *G.reachable* s' **if** *G′.reachable s*
using *reachability-complete that* **by** *auto*

end

end

end

3 Unified Passed-Waiting-List

theory *Unified-PW*
imports *Refine-Imperative-HOL.Sepref Worklist-Common Worklist-Algorithms-Subsumption-Graph*
begin

hide-const *wait*

3.1 Utilities

definition *take-from-set* **where**
take-from-set s = *ASSERT* $(s \neq \{\}) \gg$ *SPEC* $(\lambda\ (x,\ s').\ x \in s \wedge s' = s - \{x\})$

lemma *take-from-set-correct*:
assumes $s \neq \{\}$
shows *take-from-set* $s \leq$ *SPEC* $(\lambda\ (x,\ s').\ x \in s \wedge s' = s - \{x\})$
using *assms* **unfolding** *take-from-set-def* **by** *simp*

lemmas [*refine-vcg*] = *take-from-set-correct*[*THEN order.trans*]

definition *take-from-mset* **where**
take-from-mset s = *ASSERT* $(s \neq \{\#\}) \gg$ *SPEC* $(\lambda\ (x,\ s').\ x \in\# s \wedge s' = s - \{\#x\#\})$

lemma *take-from-mset-correct*:

assumes $s \neq \{\#\}$
shows *take-from-mset* $s \leq$ *SPEC* $(\lambda\ (x,\ s').\ x \in\# s \wedge s' = s - \{\#x\#\})$
using *assms* **unfolding** *take-from-mset-def* **by** *simp*

lemmas [*refine-vcg*] = *take-from-mset-correct*[*THEN order.trans*]

lemma *set-mset-mp*: *set-mset* $m \subseteq s \Longrightarrow n <$ *count* $m\ x \Longrightarrow x \in s$
by (*meson count-greater-zero-iff le-less-trans subsetCE zero-le*)

lemma *pred-not-lt-is-zero*: $(\neg\ n - Suc\ 0 < n) \longleftrightarrow n{=}0$ **by** *auto*

3.2 Generalized Worklist Algorithm

context *Search-Space-Defs-Empty*
begin
definition *reachable-subsumed* $S = \{x' \mid x\ x'.$ *reachable* $x' \wedge \neg$ *empty* $x' \wedge x' \preceq x \wedge x \in S\}$

definition
pw-var =
inv-image (
$\{(b,\ b').\ b \wedge \neg\ b'\}$
$<*lex*>$
$\{(passed',\ passed).$
$passed' \subseteq \{a.$ *reachable* $a \wedge \neg$ *empty* $a\} \wedge$ *passed* $\subseteq \{a.$ *reachable* $a \wedge \neg$ *empty* $a\} \wedge$
reachable-subsumed passed $\subset$ *reachable-subsumed passed'*$\}$
$<*lex*>$
measure size
)
$(\lambda\ (a,\ b,\ c).\ (c,\ a,\ b))$

definition *pw-inv-frontier passed wait* =
$(\forall\ a \in passed.\ (\exists\ a' \in$ *set-mset wait*. $a \preceq a') \vee$
$(\forall\ a'.\ E\ a\ a' \wedge \neg$ *empty* $a' \longrightarrow (\exists\ b' \in passed \cup$ *set-mset wait*. $a' \preceq b'))$

definition *start-subsumed passed wait* $= (\neg$ *empty* $a_0 \longrightarrow (\exists\ a \in passed \cup$ *set-mset wait*. $a_0 \preceq a))$

definition *pw-inv* $\equiv \lambda\ (passed,\ wait,\ brk).$
$(brk \longrightarrow (\exists\ f.$ *reachable* $f \wedge F\ f)) \wedge$

$(\neg\ brk \longrightarrow$
$passed \subseteq \{a.\ reachable\ a \wedge \neg\ empty\ a\}$
$\wedge$ *pw-inv-frontier passed wait*
$\wedge\ (\forall\ a \in passed \cup set\text{-}mset\ wait.\ \neg\ F\ a)$
$\wedge$ *start-subsumed passed wait*
$\wedge$ *set-mset wait* $\subseteq$ *Collect reachable*)

definition *add-pw-spec passed wait a* $\equiv$ *SPEC* (λ(*passed'*,*wait'*,*brk*).
if $\exists\, a'.\ E\ a\ a' \wedge F\ a'$ *then*
brk
else
$\neg brk \wedge set\text{-}mset\ wait' \subseteq set\text{-}mset\ wait \cup \{a'\ .\ E\ a\ a'\} \wedge$
$(\forall\ s \in set\text{-}mset\ wait.\ \exists\ s' \in set\text{-}mset\ wait'.\ s \preceq s') \wedge$
$(\forall\ s \in \{a'\ .\ E\ a\ a' \wedge \neg\ empty\ a'\}.\ \exists\ s' \in set\text{-}mset\ wait' \cup passed.\ s \preceq s') \wedge$
$(\forall\ s \in passed \cup \{a\}.\ \exists\ s' \in passed'.\ s \preceq s') \wedge$
$(passed' \subseteq passed \cup \{a\} \cup \{a'\ .\ E\ a\ a' \wedge \neg\ empty\ a'\} \wedge$
$((\exists\ x \in passed'.\ \neg\ (\exists\ x' \in passed.\ x \preceq x')) \vee wait' \subseteq\# wait \wedge passed = passed')$
)
)

definition
init-pw-spec $\equiv$
SPEC (λ (*passed*, *wait*).
if empty a_0 *then passed* $= \{\} \wedge wait \subseteq\# \{\#a_0\#\}$ *else passed* $\subseteq \{a_0\} \wedge wait = \{\#a_0\#\}$)

abbreviation *subsumed-elem* :: *'a* $\Rightarrow$ *'a set* $\Rightarrow$ *bool*
where *subsumed-elem a M* $\equiv \exists\ a'.\ a' \in M \wedge a \preceq a'$

notation
subsumed-elem ($\langle$(-/ $\in''$ -)$\rangle$ [*51*, *51*] *50*)

definition *pw-inv-frontier' passed wait* =
$(\forall\ a.\ a \in passed \longrightarrow$
$(a \in' set\text{-}mset\ wait)$
$\vee\ (\forall\ a'.\ E\ a\ a' \wedge \neg\ empty\ a' \longrightarrow (a' \in' passed \cup set\text{-}mset\ wait)))$

lemma *pw-inv-frontier-frontier'*:
pw-inv-frontier' passed wait **if**
pw-inv-frontier passed wait passed $\subseteq$ *Collect reachable*
using *that* **unfolding** *pw-inv-frontier'-def pw-inv-frontier-def* **by** (*blast*

intro: *trans*)

lemma
pw-inv-frontier passed wait **if** *pw-inv-frontier′ passed wait*
using *that* **unfolding** *pw-inv-frontier-def pw-inv-frontier′-def* **by** *blast*

definition *pw-algo* **where**
pw-algo = *do*
{
if F a_0 *then RETURN* (*True*, {})
else if empty a_0 *then RETURN* (*False*, {})
else do {
(*passed*, *wait*) ← *init-pw-spec*;
(*passed*, *wait*, *brk*) ← *WHILEIT pw-inv* (λ (*passed*, *wait*, *brk*). ¬ *brk* ∧ *wait* ≠ {#})
(λ (*passed*, *wait*, *brk*). *do*
{
(*a*, *wait*) ← *take-from-mset wait*;
ASSERT (*reachable a*);
if empty a then RETURN (*passed*, *wait*, *brk*) *else add-pw-spec passed wait a*
}
)
(*passed*, *wait*, *False*);
RETURN (*brk*, *passed*)
}
}

end

Correctness Proof **instance** *nat* :: *preorder* **..**

context *Search-Space-finite* **begin**

lemma *wf-worklist-var-aux*:
wf {(*passed′*, *passed*).
passed′ ⊆ {*a*. *reachable a* ∧ ¬ *empty a*} ∧ *passed* ⊆ {*a*. *reachable a* ∧ ¬ *empty a*} ∧
reachable-subsumed passed ⊂ *reachable-subsumed passed′*}
proof (*rule finite-acyclic-wf*, *goal-cases*)
case *1*
have {(*passed′*, *passed*).

```
      passed' ⊆ {a. reachable a ∧ ¬ empty a} ∧ passed ⊆ {a. reachable a
∧ ¬ empty a} ∧
      reachable-subsumed passed ⊂ reachable-subsumed passed'}
  ⊆ {(passed', passed).
      passed ⊆ {a. reachable a ∧ ¬ empty a} ∧ passed' ⊆ {a. reachable a
∧ ¬ empty a}}
    unfolding reachable-subsumed-def by auto
   moreover have finite ... using finite-reachable using [[simproc add:
finite-Collect]] by simp
   ultimately show ?case by (rule finite-subset)
 next
   case 2
   have *: class.preorder (≤) ((<) :: nat ⇒ nat ⇒ bool)
     by (rule preorder-class.axioms)
   show ?case
  proof (rule preorder.acyclicI-order[where f = λ a. card (reachable-subsumed
a)],
         rule preorder-class.axioms, rule psubset-card-mono)
     fix a
     have reachable-subsumed a ⊆ {a. reachable a ∧ ¬ empty a}
       unfolding reachable-subsumed-def by blast
      then show finite (reachable-subsumed a) using finite-reachable by
(rule finite-subset)
   qed auto
 qed

 lemma wf-worklist-var:
   wf pw-var
   unfolding pw-var-def
  by (auto 4 3 intro: wf-worklist-var-aux finite-acyclic-wf preorder.acyclicI-order[where
f = id]
        preorder-class.axioms)

 context
 begin

 private lemma aux5:
   assumes
     a' ∈ passed'
     a ∈# wait
     a ≼ a'
     start-subsumed passed wait
     ∀ s∈passed. ∃ x∈passed'. s ≼ x
     ∀ s∈#wait − {#a#}. Multiset.Bex wait' ((≼) s)
```

shows *start-subsumed passed′ wait′*
using *assms* **unfolding** *start-subsumed-def* **apply** *clarsimp*
by (*metis Un-iff insert-DiffM2 local.trans mset-right-cancel-elem*)

private lemma *aux11*:
assumes
empty a
start-subsumed passed wait
shows *start-subsumed passed* (*wait* − {#*a*#})
using *assms* **unfolding** *start-subsumed-def*
by *auto* (*metis UnI2 diff-single-trivial empty-mono insert-DiffM insert-noteq-member*)

lemma *aux3-aux*:
assumes *pw-inv-frontier′ passed wait*
¬ *b* ∈′ *set-mset wait*
E b b′
¬ *empty b* ¬ *empty b′*
b ∈′ *passed*
reachable b passed ⊆ {*a*. *reachable a* ∧ ¬ *empty a*}
shows *b′* ∈′ *passed* ∪ *set-mset wait*
proof −
from ‹*b* ∈′ -› **obtain** *b1* **where** *b1*: *b* ⪯ *b1 b1* ∈ *passed*
by *blast*
with *mono*[*OF this*(*1*) ‹*E b b′*›] ‹*passed* ⊆ -› ‹*reachable b*› ‹¬ *empty b*›
obtain *b1′* **where**
E b1 b1′ b′ ⪯ *b1′*
by *auto*
moreover then have ¬ *empty b1′*
using *assms*(*5*) *empty-mono* **by** *blast*
moreover from *assms b1* **have** ¬ *b1* ∈′ *set-mset wait*
by (*blast intro*: *trans*)
ultimately show *?thesis*
using *assms*(*1*) *b1*
unfolding *pw-inv-frontier′-def*
by (*blast intro*: *trans*)
qed

private lemma *pw-inv-frontier-empty-elem*:
assumes *pw-inv-frontier passed wait passed* ⊆ {*a*. *reachable a* ∧ ¬ *empty a*} *empty a*
shows *pw-inv-frontier passed* (*wait* − {#*a*#})
using *assms* **unfolding** *pw-inv-frontier-def*

by *simp*
(*smt UnCI UnE diff-single-trivial empty-mono insert-DiffM2 mset-cancel-elem*(*1*)
subset-Collect-conv)

private lemma *aux3*:
assumes
set-mset wait ⊆ *Collect reachable*
a ∈# *wait*
∀ *s* ∈ *set-mset* (*wait* − {#*a*#}). ∃ *s'* ∈ *set-mset wait'*. *s* ⪯ *s'*
∀ *s* ∈ {*a'*. *E a a'* ∧ ¬ *empty a'*}. ∃ *s'* ∈ *passed* ∪ *set-mset wait'*. *s* ⪯ *s'*
∀ *s* ∈ *passed* ∪ {*a*}. ∃ *s'* ∈ *passed'*. *s* ⪯ *s'*
passed' ⊆ *passed* ∪ {*a*} ∪ {*a'* . *E a a'* ∧ ¬ *empty a*}
pw-inv-frontier passed wait
passed ⊆ {*a*. *reachable a* ∧ ¬ *empty a*}
shows *pw-inv-frontier passed' wait'*
proof −
from *assms*(*1*,*2*) **have** *reachable a*
by (*simp add*: *subset-iff*)
from *assms* **have** *assms'*:
set-mset wait ⊆ *Collect reachable*
a ∈# *wait*
∀ *s*. *s* ∈' *set-mset* (*wait* − {#*a*#}) ⟶ *s* ∈' *set-mset wait'*
∀ *s* ∈ {*a'*. *E a a'* ∧ ¬ *empty a'*}. *s* ∈' *passed* ∪ *set-mset wait'*
∀ *s*. *s* ∈' *passed* ∪ {*a*} ⟶ *s* ∈' *passed'*
passed' ⊆ *passed* ∪ {*a*} ∪ {*a'* . *E a a'*}
pw-inv-frontier' *passed wait*
passed ⊆ {*a*. *reachable a* ∧ ¬ *empty a*}
by (*blast intro*: *trans pw-inv-frontier-frontier'*)+

show *?thesis* **unfolding** *pw-inv-frontier-def*
apply *safe*
unfolding *Bex-def*
subgoal for *b b'*
proof (*goal-cases*)
case *A*: *1*
from *A*(*1*) *assms*(*6*) **consider** *b* ∈ *passed* | *a* = *b* | *E a b*
by *auto*
note *cases* = *this*
from *cases* ‹¬ *b* ∈' *set-mset wait'*› *assms'*(*4*) ‹*reachable a*› ‹*passed* ⊆ -› **have** *reachable b*
by *cases* (*auto intro*: *reachable-step*)
with *A*(*3*,*4*) **have** ¬ *empty b* **by** (*auto simp*: *empty-E*)
from *cases this* ‹*reachable b*› **consider** *a* = *b* | *a* ≠ *b b* ∈' *passed*
reachable b

```
        using ‹¬ b ∈' set-mset wait'› assms'(4) by cases (fastforce intro:
reachable-step)+
      then consider b ⪯ a reachable b | ¬ b ⪯ a b ∈' passed reachable b
       using ‹¬ b ∈' set-mset wait'› assms'(4) ‹reachable a› by fastforce+
      then show ?case
      proof cases
        case 1
        with A(3,4) have ¬ empty b
          by (auto simp: empty-E)
       with mono[OF 1(1) ‹E b b'› 1(2) ‹reachable a›] obtain b1' where
          E a b1' b' ⪯ b1'
          by auto
        with ‹¬ empty b'› have b1' ∈' passed ∪ set-mset wait'
          using assms'(4) by (auto dest: empty-mono)
        with ‹b' ⪯ -› assms'(5) show ?thesis
          by (auto intro: trans)
      next
        case 2
        with A(3,4) have ¬ empty b
          by (auto simp: empty-E)
        from 2 ‹¬ b ∈' set-mset wait'› assms'(2,3) have ¬ b ∈' set-mset
wait
          by (metis insert-DiffM2 mset-right-cancel-elem)
        from
          aux3-aux[OF
            assms'(7) this ‹E b b'› ‹¬ empty b› ‹¬ empty b'› ‹b ∈' passed›
‹reachable b› assms'(8)
            ]
        have b' ∈' passed ∪ set-mset wait .
        with assms'(2,3,5) show ?thesis
          by auto (metis insert-DiffM insert-noteq-member)
      qed
    qed
    done
  qed

  private lemma aux6:
    assumes
      a ∈# wait
      start-subsumed passed wait
      ∀ s ∈ set-mset (wait − {#a#}) ∪ {a'. E a a' ∧ ¬ empty a'}. ∃ s' ∈
set-mset wait'. s ⪯ s'
    shows start-subsumed (insert a passed) wait'
    using assms unfolding start-subsumed-def
```

```
    apply clarsimp
    apply (erule disjE)
     apply blast
    subgoal premises prems for x
    proof (cases a = x)
      case True
      with prems show ?thesis by simp
    next
      case False
      with ‹x ∈# wait› have x ∈ set-mset (wait − {#a#})
        by (metis insert-DiffM insert-noteq-member prems(1))
      with prems(2,4) ‹- ≼ x› show ?thesis
        by (auto dest: trans)
    qed
  done

  lemma empty-E-star:
    empty x' if E** x x' reachable x empty x
    using that unfolding reachable-def
    by (induction rule: converse-rtranclp-induct)
     (blast intro: empty-E[unfolded reachable-def] rtranclp.rtrancl-into-rtrancl)+

  lemma aux4:
    assumes pw-inv-frontier passed {#} reachable x start-subsumed passed
{#}
          passed ⊆ {a. reachable a ∧ ¬ empty a} ¬ empty x
    shows ∃ x' ∈ passed. x ≼ x'
  proof −
    from ‹reachable x› have E** a0 x by (simp add: reachable-def)
    have ¬ empty a0 using ‹E** a0 x› assms(5) empty-E-star by blast
   with assms(3) obtain b where a0 ≼ b b ∈ passed unfolding start-subsumed-def
by auto
    have ∃x'. ∃ x''. E** b x' ∧ x ≼ x' ∧ x' ≼ x'' ∧ x'' ∈ passed if
                    E** a x  a ≼ b   b ≼ b'   b' ∈ passed
                    reachable a reachable b for a b b'
   using that proof (induction arbitrary: b b' rule: converse-rtranclp-induct)
      case base
      then show ?case by auto
    next
      case (step a a1 b b')
      show ?case
      proof (cases empty a)
        case True
         with step.prems step.hyps have empty x by − (rule empty-E-star,
```

```
auto)
      with step.prems show ?thesis by (auto intro: empty-subsumes)
    next
      case False
      with ‹E a a1› ‹a ⪯ b› ‹reachable a› ‹reachable b› obtain b1 where
        E b b1 a1 ⪯ b1
        using mono by blast
      show ?thesis
      proof (cases empty b1)
        case True
        with empty-mono ‹a1 ⪯ b1› have empty a1 by blast
        with step.prems step.hyps have empty x by − (rule empty-E-star,
auto simp: reachable-def)
        with step.prems show ?thesis by (auto intro: empty-subsumes)
      next
        case False
        from ‹E b b1› ‹a1 ⪯ b1› obtain b1' where E b' b1' b1 ⪯ b1'
          using ‹¬ empty a› empty-mono assms(4) mono step.prems by
blast
        from empty-mono[OF ‹¬ empty b1› ‹b1 ⪯ b1'›] have ¬ empty b1'
          by auto
        with ‹E b' b1'› ‹b' ∈ passed› assms(1) obtain b1'' where b1'' ∈
passed b1' ⪯ b1''
          unfolding pw-inv-frontier-def by auto
        with ‹b1 ⪯ -› have b1 ⪯ b1'' using trans by blast
        with step.IH[OF ‹a1 ⪯ b1› this ‹b1'' ∈ passed›] ‹reachable a› ‹E
a a1› ‹reachable b› ‹E b b1›
        obtain x' x'' where
          E** b1 x' x ⪯ x' x' ⪯ x'' x'' ∈ passed
          by (auto intro: reachable-step)
        moreover from ‹E b b1› ‹E** b1 x'› have E** b x' by auto
        ultimately show ?thesis by auto
      qed
    qed
  qed
  from this[OF ‹E** a0 x› ‹a0 ⪯ b› refl ‹b ∈ -›] assms(4) ‹b ∈ passed›
show ?thesis
    by (auto intro: trans)
qed

lemmas [intro] = reachable-step

private lemma aux7:
  assumes
```

```
    a ∈# wait
    set-mset wait ⊆ Collect reachable
    set-mset wait' ⊆ set-mset (wait − {#a#}) ∪ Collect (E a)
    x ∈# wait'
  shows reachable x
  using assms by (auto dest: in-diffD)

 private lemma aux8:
  x ∈ reachable-subsumed S' if x ∈ reachable-subsumed S ∀ s∈S. ∃ x∈S'.
s ≼ x
  using that unfolding reachable-subsumed-def by (auto intro: trans)

 private lemma aux9:
  assumes
    set-mset wait' ⊆ set-mset (wait − {#a#}) ∪ Collect (E a)
    x ∈# wait' ∀ a'. E a a' ⟶ ¬ F a' F x
    ∀ a∈passed ∪ set-mset wait. ¬ F a
  shows False
 proof −
  from assms(1,2) have x ∈ set-mset wait ∨ x ∈ Collect (E a)
    by (meson UnE in-diffD subsetCE)
  with assms(3,4,5) show ?thesis
    by auto
 qed

 private lemma aux10:
  assumes ∀ a∈passed' ∪ set-mset wait. ¬ F a F x x ∈# wait − {#a#}
  shows False
  by (meson UnI2 assms in-diffD)

 lemma aux12:
  size wait' < size wait if wait' ⊆# wait − {#a#} a ∈# wait
  using that
  by (metis
    Diff-eq-empty-iff-mset add-diff-cancel-left' add-mset-add-single add-mset-not-empty
      insert-subset-eq-iff mset-le-add-mset-decr-left1 mset-subset-size sub-
set-mset-def)

 lemma aux13:
  assumes
  passed ⊆ {a. reachable a ∧ ¬ empty a}
  passed' ⊆ insert a (passed ∪ {a'. E a a' ∧ ¬ empty a'})
  ¬ empty a
  reachable a
```

$\forall\, s{\in}passed.\ \exists\, x{\in}passed'.\ s \preceq x$
$a'' \in passed'$
$\forall\, x{\in}passed.\ \neg\ a'' \preceq x$
shows
$passed' \subseteq \{a.\ reachable\ a \wedge \neg\ empty\ a\} \wedge reachable\text{-}subsumed\ passed \subset reachable\text{-}subsumed\ passed'$
$\vee\ passed' = passed \wedge size\ wait'' < size\ wait$
proof −
have $passed' \subseteq \{a.\ reachable\ a \wedge \neg\ empty\ a\}$
using ‹$passed \subseteq$ -› ‹$passed' \subseteq$ -› ‹$\neg\ empty\ a$› ‹$reachable\ a$› **by** *auto*
moreover have $reachable\text{-}subsumed\ passed \subset reachable\text{-}subsumed\ passed'$
unfolding *reachable-subsumed-def*
using ‹$\forall\, s{\in}passed.\ \exists\, x{\in}passed'.\ s \preceq x$› *assms*(5−) ‹$passed' \subseteq \{a.\ reachable\ a \wedge \neg\ empty\ a\}$›
by (*auto 4 3 intro*: *trans*)
ultimately show *?thesis*
using ‹$passed \subseteq$ -› **unfolding** *pw-var-def* **by** *auto*
qed

method *solve-vc* =
rule aux3 aux5 aux7 aux10 aux11 pw-inv-frontier-empty-elem; *assumption*; *fail* |
rule aux3; *auto*; *fail* | *auto intro*: *aux9*; *fail* | *auto dest*: *in-diffD*; *fail*

end — Context

end — Search Space

theorem (**in** *Search-Space′-finite*) *pw-algo-correct*:
$pw\text{-}algo \leq SPEC\ (\lambda\ (brk, passed).$
$(brk \longleftrightarrow F\text{-}reachable)$
$\wedge\ (\neg\ brk \longrightarrow$
$(\forall\ a.\ reachable\ a \wedge \neg\ empty\ a \longrightarrow (\exists\ b \in passed.\ a \preceq b))$
$\wedge\ passed \subseteq \{a.\ reachable\ a \wedge \neg\ empty\ a\})$
)
proof −
note [*simp*] = *size-Diff-submset pred-not-lt-is-zero*
note [*dest*] = *set-mset-mp*
show *?thesis*
unfolding *pw-algo-def init-pw-spec-def add-pw-spec-def F-reachable-def*
apply (*refine-vcg wf-worklist-var*)

apply (*solves auto*)

subgoal
using *empty-E-star final-non-empty* **unfolding** *reachable-def*
by *auto*
subgoal
using *empty-E-star final-non-empty* **unfolding** *reachable-def*
by *auto*
subgoal
using *empty-E-star final-non-empty* **unfolding** *reachable-def*
by *auto*
subgoal
using *empty-E-star final-non-empty* **unfolding** *reachable-def*
by *auto*
subgoal
using *empty-E-star final-non-empty* **unfolding** *reachable-def*
by *auto*

apply (*fastforce simp*: *pw-inv-def pw-inv-frontier-def start-subsumed-def*
split: *if-split-asm dest*: *mset-subset-eqD*)

apply (*simp*; *fail*)

apply (*solves* ‹*auto simp*: *pw-inv-def*›)

subgoal for - *passed wait - passed′ - - brk - a wait′*
by (*clarsimp simp*: *pw-inv-def split*: *if-split-asm*; *safe*; *solve-vc*)

apply (*clarsimp simp*: *pw-var-def nonempty-has-size*; *fail*)

subgoal for - *passed wait - passed′ - - brk - a wait′*
by (*clarsimp simp*: *pw-inv-def split*: *if-split-asm*; *safe*; *solve-vc*)

subgoal for - - - - *passed - wait brk - a wait′*
by (*clarsimp simp*: *pw-inv-def split*: *if-split-asm*; *safe*)
(*simp-all add*: *aux12 aux13 pw-var-def*)

using *F-mono* **by** (*fastforce simp*: *pw-inv-def dest*!: *aux4 dest*: *final-non-empty*)+
qed

lemmas (**in** *Search-Space′-finite*) [*refine-vcg*] = *pw-algo-correct*[*THEN Orderings.order.trans*]

end — End of Theory

theory *Unified-PW-Hashing*
imports
Unified-PW
Refine-Imperative-HOL.IICF-List-Mset
Worklist-Algorithms-Misc
Worklist-Algorithms-Tracing
begin

3.3 Towards an Implementation of the Unified Passed-Waiting List

context *Worklist1-Defs*
begin

definition *add-pw-unified-spec passed wait a* ≡ *SPEC* (λ(*passed'*,*wait'*,*brk*).
if ∃ *x* ∈ *set* (*succs a*). *F x then brk*
else passed' ⊆ *passed* ∪ {*x* ∈ *set* (*succs a*). ¬ (∃ *y* ∈ *passed*. *x* ≼ *y*)}
∧ *passed* ⊆ *passed'*
∧ *wait* ⊆# *wait'*
∧ *wait'* ⊆# *wait* + *mset* ([*x* ← *succs a*. ¬ (∃ *y* ∈ *passed*. *x* ≼ *y*)])
∧ (∀ *x* ∈ *set* (*succs a*). ∃ *y* ∈ *passed'*. *x* ≼ *y*)
∧ (∀ *x* ∈ *set* (*succs a*). ¬ (∃ *y* ∈ *passed*. *x* ≼ *y*) ⟶ (∃ *y* ∈# *wait'*.
x ≼ *y*))
∧ ¬ *brk*)

definition *add-pw passed wait a* ≡
nfoldli (*succs a*) (λ(-, -, *brk*). ¬*brk*)
(λ*a* (*passed*, *wait*, *brk*). *RETURN* (
if F a then
(*passed*, *wait*, *True*)
else if ∃ *x* ∈ *passed*. *a* ≼ *x then*
(*passed*, *wait*, *False*)
else (*insert a passed*, *add-mset a wait*, *False*)
))
(*passed*, *wait*, *False*)

end — Worklist1 Defs

context *Worklist1*
begin

lemma *add-pw-unified-spec-ref*:

add-pw-unified-spec passed wait a $\leq$ *add-pw-spec passed wait a*
if *reachable a a* $\in$ *passed*
using *succs-correct*[*OF that*(*1*)] *that*(*2*)
unfolding *add-pw-unified-spec-def add-pw-spec-def*
apply *simp*
apply *safe*
apply (*all* ‹*auto simp*: *empty-subsumes*; *fail* | *succeed*›)
using *mset-subset-eqD* **apply** *force*
using *mset-subset-eqD* **apply** *force*
subgoal premises *prems*
using *prems*
by (*auto 4 5 simp*: *filter-mset-eq-empty-iff intro*: *trans elim*!: *subset-mset.ord-le-eq-trans*)

by (*clarsimp*, *smt UnE mem-Collect-eq subsetCE*)

lemma *add-pw-ref*:
add-pw passed wait a $\leq \Downarrow$ *Id* (*add-pw-unified-spec passed wait a*)
unfolding *add-pw-def add-pw-unified-spec-def*
apply (*refine-vcg*
nfoldli-rule[**where** $I =$
λ *l1 l2* (*passed*$'$, *wait*$'$, *brk*).
if brk then $\exists\ a' \in$ *set* (*succs a*). $F\ a'$
else passed$' \subseteq$ *passed* $\cup$ $\{x \in$ *set l1*. $\neg\ (\exists\ y \in$ *passed*. $x \preceq y)\}$
$\wedge$ *passed* $\subseteq$ *passed*$'$
$\wedge$ *wait* $\subseteq\#$ *wait*$'$
$\wedge$ *wait*$' \subseteq\#$ *wait* + *mset* $[x \leftarrow$ *l1*. $\neg\ (\exists\ y \in$ *passed*. $x \preceq y)]$
$\wedge\ (\forall\ x \in$ *set l1*. $\exists\ y \in$ *passed*$'$. $x \preceq y)$
$\wedge\ (\forall\ x \in$ *set l1*. $\neg\ (\exists\ y \in$ *passed*. $x \preceq y) \longrightarrow (\exists\ y \in\#$ *wait*$'$. $x \preceq$
$y))$
$\wedge$ *set l1* $\cap$ *Collect* $F = \{\}$

])
apply (*solves auto*)
apply (*clarsimp split*: *if-split-asm*)
apply *safe*[]
apply (*solves* ‹*auto simp add*: *subset-mset.le-iff-add*›)+
subgoal premises *prems*
using *prems trans* **by** (*metis* (*no-types*, *lifting*) *Un-iff in-mono mem-Collect-eq*)
by (*auto simp*: *subset-mset.le-iff-add*)

end — Worklist 1

context *Worklist2-Defs*
begin

definition *add-pw′ passed wait a* ≡
nfoldli (*succs a*) (λ(-, -, *brk*). ¬*brk*)
(λ*a* (*passed*, *wait*, *brk*). *RETURN* (
if F a then
(*passed*, *wait*, *True*)
else if empty a then
(*passed*, *wait*, *False*)
else if ∃ *x* ∈ *passed*. *a* ⊴ *x then*
(*passed*, *wait*, *False*)
else (*insert a passed*, *add-mset a wait*, *False*)
))
(*passed*, *wait*, *False*)

definition *pw-algo-unified* **where**
pw-algo-unified = *do*
{
if F a_0 *then RETURN* (*True*, {})
else if empty a_0 *then RETURN* (*False*, {})
else do {
(*passed*, *wait*) ← *RETURN* ({a_0}, {#a_0#});
(*passed*, *wait*, *brk*) ← *WHILEIT pw-inv* (λ (*passed*, *wait*, *brk*). ¬ *brk* ∧ *wait* ≠ {#})
(λ (*passed*, *wait*, *brk*). *do*
{
(*a*, *wait*) ← *take-from-mset wait*;
ASSERT (*reachable a*);
if empty a then RETURN (*passed*, *wait*, *brk*) *else add-pw′ passed wait a*
}
)
(*passed*, *wait*, *False*);
RETURN (*brk*, *passed*)
}
}

end — Worklist 2 Defs

context *Worklist2*
begin

lemma *empty-subsumes′2*:

empty x ∨ *x* ⊴ *y* ⟷ *x* ≼ *y*
using *empty-subsumes′ empty-subsumes* **by** *auto*

lemma *bex-or*:
P ∨ (∃ *x* ∈ *S*. *Q x*) ⟷ (∃ *x* ∈ *S*. *P* ∨ *Q x*) **if** *S* ≠ {}
using *that* **by** *auto*

lemma *add-pw′-ref′*:
add-pw′ passed wait a ≤ ⇓ (*Id* ∩ {((*p*, *w*, -), -). *p* ≠ {} ∧ *set-mset w* ⊆ *p*}) (*add-pw passed wait a*)
if *passed* ≠ {} *set-mset wait* ⊆ *passed*
unfolding *add-pw′-def add-pw-def*
apply (*rule nfoldli-refine*)
apply *refine-dref-type*
using *that* **apply** (*solves auto*)+
apply *refine-rcg*
apply (*rule Set.IntI*)
unfolding *z3-rule*(*44*)
apply (*subst bex-or*)
by (*auto simp add*: *empty-subsumes′2*)

lemma *add-pw′-ref1*[*refine*]:
add-pw′ passed wait a
≤ ⇓ (*Id* ∩ {((*p*, *w*, -), -). *p* ≠ {} ∧ *set-mset w* ⊆ *p*}) (*add-pw-spec passed′ wait′ a′*)
if *passed* ≠ {} *set-mset wait* ⊆ *passed reachable a a* ∈ *passed*
and [*simp*]: *passed* = *passed′ wait* = *wait′ a* = *a′*
proof −
from *add-pw-unified-spec-ref*[*OF that*(*3*−*4*), *of wait*] *add-pw-ref*[*of passed wait a*] **have**
add-pw passed wait a ≤ ⇓ *Id* (*add-pw-spec passed wait a*)
by *simp*
moreover note *add-pw′-ref′*[*OF that*(*1*,*2*), *of a*]
ultimately show *?thesis*
by (*auto simp*: *pw-le-iff refine-pw-simps*)
qed

lemma *refine-weaken*:
p ≤ ⇓ *R p′* **if** *p* ≤ ⇓ *S p′ S* ⊆ *R*
using *that* **by** (*simp add*: *pw-le-iff refine-pw-simps*; *blast*)

lemma *add-pw′-ref*:
add-pw′ passed wait a ≤
⇓ ({((*p*, *w*, *b*), (*p′*, *w′*, *b′*)). *p* ≠ {} ∧ *p* = *p′* ∪ *set-mset w* ∧ *w* = *w′* ∧

$b = b'\}$)
(*add-pw-spec passed' wait' a'*)
if *passed* $\neq \{\}$ *set-mset wait* $\subseteq$ *passed reachable a a* $\in$ *passed*
and [*simp*]: *passed* = *passed' wait* = *wait' a* = *a'*
by (*rule add-pw'-ref1*[*OF that*, *THEN refine-weaken*]; *auto*)

lemma
(($\{a_0\}$, $\{\#a_0\#\}$, *False*), $\{\}$, $\{\#a_0\#\}$, *False*)
$\in \{((p, w, b), (p', w', b')).\ p = p' \cup$ *set-mset* $w' \wedge w = w' \wedge b = b'\}$
by *auto*

lemma [*refine*]:
RETURN ($\{a_0\}$, $\{\#a_0\#\}$) $\leq \Downarrow$ (*Id* $\cap$ $\{((p, w), (p', w')).\ p \neq \{\} \wedge$ *set-mset* $w \subseteq p\}$) *init-pw-spec*
if $\neg$ *empty* a_0
using *that* **unfolding** *init-pw-spec-def* **by** (*auto simp*: *pw-le-iff refine-pw-simps*)

lemma [*refine*]:
take-from-mset wait $\leq$
$\Downarrow$ $\{((x, wait), (y, wait')).\ x = y \wedge wait = wait' \wedge$ *set-mset wait* $\subseteq$ *passed* $\wedge\ x \in$ *passed*$\}$
(*take-from-mset wait'*)
if *wait* = *wait' set-mset wait* $\subseteq$ *passed wait* $\neq \{\#\}$
using *that*
by (*auto 4 5 simp*: *pw-le-iff refine-pw-simps dest*: *in-diffD dest*!: *take-from-mset-correct*)

lemma *pw-algo-unified-ref*:
pw-algo-unified $\leq \Downarrow$ *Id pw-algo*
unfolding *pw-algo-unified-def pw-algo-def*
by *refine-rcg* (*auto simp*: *init-pw-spec-def*)

end — Worklist 2

Utilities **definition** *take-from-list* **where**
take-from-list s = *ASSERT* ($s \neq []$) $\gg$ *SPEC* (λ (x, s'). $s = x \#\ s'$)

lemma *take-from-list-correct*:
assumes $s \neq []$
shows *take-from-list s* $\leq$ *SPEC* (λ (x, s'). $s = x \#\ s'$)
using *assms* **unfolding** *take-from-list-def* **by** *simp*

lemmas [*refine-vcg*] = *take-from-list-correct*[*THEN order.trans*]

context *Worklist-Map-Defs*
begin

definition
add-pw′-map passed wait a ≡
nfoldli (*succs a*) (λ(-, -, *brk*). ¬*brk*)
(λ*a* (*passed*, *wait*, -).
do {
RETURN (
if F a then (*passed*, *wait*, *True*) *else*
let k = *key a*; *passed′* = (*case passed k of Some passed′* ⇒ *passed′* |
None ⇒ {})
in
if empty a then
(*passed*, *wait*, *False*)
else if ∃ *x* ∈ *passed′*. *a* ⊴ *x then*
(*passed*, *wait*, *False*)
else
(*passed*(*k* ↦ (*insert a passed′*)), *a* # *wait*, *False*)
)
}
)
(*passed*,*wait*,*False*)

definition
pw-map-inv ≡ λ (*passed*, *wait*, *brk*).
∃ *passed′ wait′*.
(*passed*, *passed′*) ∈ *map-set-rel* ∧ (*wait*, *wait′*) ∈ *list-mset-rel* ∧
pw-inv (*passed′*, *wait′*, *brk*)

definition *pw-algo-map* **where**
pw-algo-map = *do*
{
if F a_0 *then RETURN* (*True*, *Map.empty*)
else if empty a_0 *then RETURN* (*False*, *Map.empty*)
else do {
(*passed*, *wait*) ← *RETURN* ([*key* a_0 ↦ {a_0}], [a_0]);
(*passed*, *wait*, *brk*) ← *WHILEIT pw-map-inv* (λ (*passed*, *wait*, *brk*).
¬ *brk* ∧ *wait* ≠ [])
(λ (*passed*, *wait*, *brk*). *do*
{
(*a*, *wait*) ← *take-from-list wait*;
ASSERT (*reachable a*);

if empty a then RETURN (*passed*, *wait*, *brk*) *else add-pw′-map passed wait a*
}
)
(*passed*, *wait*, *False*);
RETURN (*brk*, *passed*)
}
}

end — Worklist Map Defs

lemma *ran-upd-cases*:
(*x* ∈ *ran m*) ∨ (*x* = *y*) **if** *x* ∈ *ran* (*m*(*a* ↦ *y*))
using *that* **unfolding** *ran-def* **by** (*auto split*: *if-split-asm*)

lemma *ran-upd-cases2*:
(∃ *k*. *m k* = *Some x* ∧ *k* ≠ *a*) ∨ (*x* = *y*) **if** *x* ∈ *ran* (*m*(*a* ↦ *y*))
using *that* **unfolding** *ran-def* **by** (*auto split*: *if-split-asm*)

context *Worklist-Map*
begin

lemma *add-pw′-map-ref*[*refine*]:
add-pw′-map passed wait a ≤ ⇓ (*map-set-rel* ×$_r$ *list-mset-rel* ×$_r$ *bool-rel*) (*add-pw′ passed′ wait′ a′*)
if (*passed*, *passed′*) ∈ *map-set-rel* (*wait*, *wait′*) ∈ *list-mset-rel* (*a*, *a′*) ∈ *Id*
using *that*
unfolding *add-pw′-map-def add-pw′-def*
apply *refine-rcg*
apply *refine-dref-type*
apply (*solves auto*)
apply (*solves auto*)
apply (*solves auto*)
subgoal premises *assms* **for** *a a′ - - passed′ - wait′ f′ passed - wait f*
proof −
from *assms* **have** [*simp*]: *a′* = *a f* = *f′* **by** *simp*+
from *assms* **have** *rel-passed*: (*passed*, *passed′*) ∈ *map-set-rel* **by** *simp*
then have *union*: *passed′* = ⋃ (*ran passed*)
unfolding *map-set-rel-def* **by** *auto*
from *assms* **have** *rel-wait*: (*wait*, *wait′*) ∈ *list-mset-rel* **by** *simp*
from *rel-passed* **have** *keys*[*simp*]: *key v* = *k* **if** *passed k* = *Some xs v* ∈ *xs* **for** *k xs v*
using *that* **unfolding** *map-set-rel-def* **by** *auto*

```
    define k where k ≡ key a
    define xs where xs ≡ case passed k of None ⇒ {} | Some p ⇒ p
    have xs-ran: x ∈ ⋃(ran passed) if x ∈ xs for x
      using that unfolding xs-def ran-def by (auto split: option.split-asm)
    have ∗: (∃ x ∈ xs. a ⊴ x) ⟷ (∃ x∈passed'. a' ⊴ x)
    proof standard
      assume ∃ x∈xs. a ⊴ x
      with rel-passed show ∃ x∈passed'. a' ⊴ x
        unfolding xs-def union by (auto intro: ranI split: option.split-asm)
    next
      assume ∃ x∈passed'. a' ⊴ x
      with rel-passed show ∃ x∈xs. a ⊴ x unfolding xs-def union ran-def
k-def map-set-rel-def
        using empty-subsumes'2 by force
    qed
    have (passed(k ↦ insert a xs), insert a' passed') ∈ map-set-rel
      using ‹(passed, passed') ∈ map-set-rel›
      unfolding map-set-rel-def
      apply safe
      subgoal
        unfolding union by (auto dest!: ran-upd-cases xs-ran)
      subgoal
        unfolding ran-def by auto
      subgoal for a''
        unfolding union ran-def
        apply clarsimp
        subgoal for k'
          unfolding xs-def by (cases k' = k) auto
        done
      by (clarsimp split: if-split-asm, safe,
        auto intro!: keys simp: xs-def k-def split: option.split-asm if-split-asm)
    with rel-wait rel-passed show ?thesis
      unfolding ∗[symmetric]
      unfolding xs-def k-def Let-def
      unfolding list-mset-rel-def br-def
      by auto
  qed
done

lemma init-map-ref[refine]:
  (([key a₀ ↦ {a₀}], [a₀]), {a₀}, {#a₀#}) ∈ map-set-rel ×ᵣ list-mset-rel
  unfolding map-set-rel-def list-mset-rel-def br-def by auto

lemma take-from-list-ref[refine]:
```

take-from-list xs ≤ ⇓ (*Id* ×$_r$ *list-mset-rel*) (*take-from-mset ms*) **if** (*xs*, *ms*) ∈ *list-mset-rel*
using *that* **unfolding** *take-from-list-def take-from-mset-def list-mset-rel-def br-def*
by (*clarsimp simp*: *pw-le-iff refine-pw-simps*)

lemma *pw-algo-map-ref*:
pw-algo-map ≤ ⇓ (*Id* ×$_r$ *map-set-rel*) *pw-algo-unified*
unfolding *pw-algo-map-def pw-algo-unified-def*
apply *refine-rcg*
unfolding *pw-map-inv-def list-mset-rel-def br-def map-set-rel-def* **by** *auto*

end — Worklist Map

context *Worklist-Map2-Defs*
begin

definition
add-pw′-map2 passed wait a ≡
nfoldli (*succs a*) (λ(-, -, *brk*). ¬*brk*)
(λ*a* (*passed*, *wait*, -).
do {
RETURN (
if empty a then
(*passed*, *wait*, *False*)
else if F′ a then (*passed*, *wait*, *True*)
else
let k = *key a*; *passed′* = (*case passed k of Some passed′* ⇒ *passed′* | *None* ⇒ {})
in
if ∃ *x* ∈ *passed′*. *a* ⊴ *x then*
(*passed*, *wait*, *False*)
else
(*passed*(*k* ↦ (*insert a passed′*)), *a* # *wait*, *False*)
)
}
)
(*passed*,*wait*,*False*)

definition *pw-algo-map2* **where**
pw-algo-map2 = *do*
{
if F a$_0$ *then RETURN* (*True*, *Map.empty*)

```
    else if empty a0 then RETURN (False, Map.empty)
    else do {
      (passed, wait) ← RETURN ([key a0 ↦ {a0}], [a0]);
      (passed, wait, brk) ← WHILEIT pw-map-inv (λ (passed, wait, brk).
¬ brk ∧ wait ≠ [])
        (λ (passed, wait, brk). do
          {
            (a, wait) ← take-from-list wait;
            ASSERT (reachable a);
            if empty a
            then RETURN (passed, wait, brk)
            else do {
              TRACE (ExploredState); add-pw'-map2 passed wait a
            }
          }
        )
        (passed, wait, False);
        RETURN (brk, passed)
    }
  }
```

end — Worklist Map 2 Defs

context *Worklist-Map2*
begin

lemma *add-pw'-map2-ref*[*refine*]:
  add-pw'-map2 passed wait a ≤ ⇓ *Id* (*add-pw'-map passed' wait' a'*)
  if (*passed*, *passed'*) ∈ *Id* (*wait*, *wait'*) ∈ *Id* (*a*, *a'*) ∈ *Id*
  using *that*
  unfolding *add-pw'-map2-def add-pw'-map-def*
  apply *refine-rcg*
    apply *refine-dref-type*
  by (*auto simp*: *F-split*)

lemma *pw-algo-map2-ref*[*refine*]:
  pw-algo-map2 ≤ ⇓ *Id pw-algo-map*
  unfolding *pw-algo-map2-def pw-algo-map-def TRACE-bind*
  apply *refine-rcg*
        apply *refine-dref-type*
  by *auto*

end — Worklist Map 2

lemma (**in** *Worklist-Map2-finite*) *pw-algo-map2-correct*:
pw-algo-map2 ≤ *SPEC* (λ (*brk*, *passed*).
(*brk* ⟷ *F-reachable*) ∧
(¬ *brk* ⟶
(∃ *p*.
(*passed*, *p*) ∈ *map-set-rel* ∧ (∀ *a*. *reachable a* ∧ ¬ *empty a* ⟶ (∃ *b*∈*p*.
a ≼ *b*))
∧ *p* ⊆ {*a*. *reachable a* ∧ ¬ *empty a*})
)
)
proof −
note *pw-algo-map2-ref*
also note *pw-algo-map-ref*
also note *pw-algo-unified-ref*
also note *pw-algo-correct*
finally show *?thesis*
unfolding *conc-fun-def Image-def* **by** (*fastforce intro*: *Orderings.order.trans*)

qed

end — End of Theory

3.4 Heap Hash Map

theory *Heap-Hash-Map*
imports
Separation-Logic-Imperative-HOL.Sep-Main Separation-Logic-Imperative-HOL.Sep-Examples
Refine-Imperative-HOL.IICF
begin

no-notation *Ref.update* (‹- := -› *62*)

definition *big-star* :: *assn multiset* ⇒ *assn* (‹⋀* -› [*60*] *90*) **where**
big-star S ≡ *fold-mset* (∗) *emp S*

interpretation *comp-fun-commute-mult*:
comp-fun-commute (∗) :: ('*a* :: *ab-semigroup-mult* ⇒ - ⇒ -)
by *standard* (*auto simp*: *ab-semigroup-mult-class.mult.left-commute*)

lemma *sep-big-star-insert* [*simp*]: ⋀* (*add-mset x S*) = (*x* ∗ ⋀* *S*)
by (*auto simp*: *big-star-def*)

lemma *sep-big-star-union* [*simp*]: ⋀* (*S* + *T*) = (⋀* *S*) ∗ (⋀* *T*)

by (*auto simp*: *big-star-def comp-fun-commute-mult.fold-mset-fun-left-comm*)

lemma *sep-big-star-empty* [*simp*]: $\bigwedge* \{\#\} = emp$
by (*simp add*: *big-star-def*)

lemma *big-star-entatilst-mono*:
$\bigwedge* T \Longrightarrow_t \bigwedge* S$ **if** $S \subseteq\# T$
using *that*
proof (*induction T arbitrary*: *S*)
case *empty*
then show *?case*
by *auto*
next
case (*add x T*)
then show *?case*
proof (*cases x* $\in\#$ *S*)
case *True*
then obtain S' **where** $S' \subseteq\# T$ $S = add\text{-}mset\ x\ S'$
by (*metis add.prems mset-add mset-subset-eq-add-mset-cancel*)
with *add.IH*[*of S'*] *add.prems* **have** $\bigwedge* T \Longrightarrow_t \bigwedge* S'$
by *auto*
then show *?thesis*
unfolding ‹S = -› **by** (*sep-auto intro*: *entt-star-mono*)
next
case *False*
with *add.prems* **have** $S \subseteq\# T$
by (*metis add-mset-remove-trivial diff-single-trivial mset-le-subtract*)
then have $\bigwedge* T \Longrightarrow_t \bigwedge* S$
by (*rule add.IH*)
then show *?thesis*
using *entt-fr-drop star-aci*(*2*) **by** *fastforce*
qed
qed

definition *map-assn V m mi* $\equiv$
$\uparrow (dom\ mi = dom\ m \wedge finite\ (dom\ m)) *$
$(\bigwedge* \{\#\ V\ (the\ (m\ k))\ (the\ (mi\ k))\ .\ k \in\#\ mset\text{-}set\ (dom\ m)\#\})$

lemma *map-assn-empty-map*[*simp*]:
map-assn A Map.empty Map.empty = *emp*
unfolding *map-assn-def* **by** *auto*

lemma *in-mset-union-split*:
mset-set S = *mset-set* $(S - \{k\}) + \{\#k\#\}$ **if** $k \in S$ *finite* S
using *that* **by** (*auto simp*: *mset-set.remove*)

lemma *in-mset-dom-union-split*:
mset-set (*dom* m) = *mset-set* (*dom* $m - \{k\}) + \{\#k\#\}$ **if** $m\ k$ = *Some* v *finite* (*dom* m)
apply (*rule in-mset-union-split*)
using *that* **by** (*auto*)

lemma *dom-remove-not-in-dom-simp*[*simp*]:
dom $m - \{k\}$ = *dom* m **if** $m\ k$ = *None*
using *that* **unfolding** *dom-def* **by** *auto*

lemma *map-assn-delete*:
map-assn $A\ m\ mh \Longrightarrow_A$
map-assn $A\ (m(k := None))\ (mh(k := None)) * option\text{-}assn\ A\ (m\ k)$ $(mh\ k)$
unfolding *map-assn-def*
apply *sep-auto*
apply (*sep-auto cong*: *multiset.map-cong-simp*)
apply (*cases* $m\ k$; *cases* $mh\ k$; *simp*)
apply (*solves auto*)+
by (*subst in-mset-dom-union-split*, *assumption*+, *sep-auto*)

lemma *in-mset-set-iff-in-set*[*simp*]:
$z \in\#$ *mset-set* $S \longleftrightarrow z \in S$ **if** *finite* S
using *that* **by** *auto*

lemma *ent-refl'*:
$a = b \Longrightarrow a \Longrightarrow_A b$
by *auto*

lemma *map-assn-update-aux*:
map-assn $A\ m\ mh * A\ v\ vi \Longrightarrow_A$ *map-assn* $A\ (m(k \mapsto v))\ (mh(k \mapsto vi))$
if $k \notin$ *dom* m
unfolding *map-assn-def*
apply (*sep-auto simp*: *that cong*: *multiset.map-cong-simp*)
apply (*subst mult.commute*)
apply (*rule ent-star-mono*)
apply *simp*
apply (*rule ent-refl'*)
apply (*rule arg-cong*[**where** f = *big-star*])
apply (*rule image-mset-cong*)

using *that* **by** *auto*

lemma *map-assn-update*:
map-assn A m mh ∗ *A v vi* ⟹$_A$
map-assn A (*m*(*k* ↦ *v*)) (*mh*(*k* ↦ *vi*)) ∗ *true*
apply (*rule ent-frame-fwd*[*OF map-assn-delete*[**where** *k* = *k*]], *frame-inference*)
apply (*rule ent-frame-fwd*[*OF map-assn-update-aux*[**where** *k* = *k* **and** *A* = *A*], *rotated*], *frame-inference*)
by *sep-auto*+

— This is a generic pattern to enhance a map-implementation to heap-based values, a la Charguéraud.
definition (**in** *imp-map*) *hms-assn A m mi* ≡ ∃$_A$*mh*. *is-map mh mi* ∗ *map-assn A m mh*

definition (**in** *imp-map*) *hms-assn′ K A* = *hr-comp* (*hms-assn A*) (⟨*the-pure K*, *Id*⟩*map-rel*)
declare (**in** *imp-map*) *hms-assn′-def*[*symmetric*, *fcomp-norm-unfold*]

definition (**in** *imp-map-empty*) [*code-unfold*]: *hms-empty* ≡ *empty*

lemma (**in** *imp-map-empty*) *hms-empty-rule* [*sep-heap-rules*]:
<*emp*> *hms-empty* <*hms-assn A Map.empty*>$_t$
unfolding *hms-empty-def hms-assn-def*
by (*sep-auto eintros*: *exI*[**where** *x* = *Map.empty*])

definition (**in** *imp-map-update*) [*code-unfold*]: *hms-update* = *update*

lemma (**in** *imp-map-update*) *hms-update-rule* [*sep-heap-rules*]:
<*hms-assn A m mi* ∗ *A v vi*> *hms-update k vi mi* <*hms-assn A* (*m*(*k* ↦ *v*))>$_t$
unfolding *hms-update-def hms-assn-def*
apply (*sep-auto eintros del*: *exI*)
subgoal for *mh mi*
apply (*rule exI*[**where** *x* = *mh*(*k* ↦ *vi*)])
apply (*rule ent-frame-fwd*[*OF map-assn-update*[**where** *A* = *A*]], *frame-inference*)
by *sep-auto*
done

lemma *restrict-not-in-dom-simp*[*simp*]:
m |‘ (− {*k*}) = *m* **if** *m k* = *None*
using *that* **by** (*auto simp*: *restrict-map-def*)

definition [*code*]:

```
  hms-extract lookup delete k m =
    do {
      vo ← lookup k m;
      case vo of
        None ⇒ return (None, m) |
        Some v ⇒ do {
          m ← delete k m;
          return (Some v, m)
        }
    }


definition [code]:
  hms-lookup lookup copy k m =
    do {
      vo ← lookup k m;
      case vo of
        None ⇒ return None |
        Some v ⇒ do {
          v' ← copy v;
          return (Some v')
        }
    }


locale imp-map-extract-derived = imp-map-delete + imp-map-lookup
begin

lemma map-assn-domain-simps[simp]:
  assumes vassn-tag (map-assn A m mh)
  shows mh k = None ⟷ m k = None dom mh = dom m finite (dom m)
  using assms unfolding map-assn-def vassn-tag-def by auto

lemma hms-extract-rule [sep-heap-rules]:
  <hms-assn A m mi>
    hms-extract lookup delete k mi
  <λ (vi, mi'). option-assn A (m k) vi ∗ hms-assn A (m(k := None)) mi'>t
  unfolding hms-extract-def hms-assn-def
  apply (sep-auto eintros del: exI)
  subgoal
    unfolding map-assn-def by auto
  subgoal for mh
    apply (rule exI[where x = mh(k := None)])
    apply (rule fr-refl)
```

apply (*rule ent-star-mono*, *simp add*: *fun-upd-idem*)
apply (*rule entails-preI*, *simp add*: *fun-upd-idem*)
done
apply (*sep-auto eintros del*: *exI*)
subgoal for *mh*
apply (*rule exI*[**where** $x = mh(k := None)$])
apply (*rule ent-frame-fwd*[*OF map-assn-delete*[**where** $A = A$]], *frame-inference*)
by (*sep-auto simp add*: *map-upd-eq-restrict*)+
done

lemma *hms-lookup-rule* [*sep-heap-rules*]:
assumes
(*copy*, *RETURN o COPY*) $\in A^k \rightarrow_a A$
shows
<*hms-assn A m mi*>
hms-lookup lookup copy k mi
<λ *vi*. *hms-assn A m mi* $*$ *option-assn A* (*m k*) *vi*>$_t$
proof −
have *0*:
<*is-map mh mi* $*$ *map-assn A* ($m(k := None)$) ($mh(k := None)$) $*$ *option-assn A* (*m k*) (*mh k*) $*$ *true*>
copy x'
<$\lambda r.\ \exists_A x.$
is-map mh mi $*$ *map-assn A* ($m(k := None)$) ($mh(k := None)$) $*$ *option-assn A* (*m k*) (*mh k*)
$*$ *A x r* $* \uparrow(m\ k = Some\ x)$
$*$ *true*>
if $mh\ k = Some\ x'$ **for** *mh* x'
supply [*sep-heap-rules*] = *assms*[*to-hnr*, *unfolded hn-refine-def hn-ctxt-def*, *simplified*]
by (*sep-auto simp add*: *that option-assn-alt-def split*: *option.splits*)
have *1*:
<*is-map mh mi* $*$ *map-assn A m mh* $*$ *true*>
copy x'
<$\lambda r.\ \exists_A x.$ *is-map mh mi* $*$ *map-assn A m mh* $*$ *A x r* $* \uparrow(m\ k = Some\ x)$ $*$ *true*>
if $mh\ k = Some\ x'$ **for** *mh* x'
apply (*rule cons-rule*[*rotated 2*], *rule 0*, *rule that*)
apply (*rule ent-frame-fwd*[*OF map-assn-delete*[**where** $A = A$]], *frame-inference*, *frame-inference*)
apply *sep-auto*
apply (*fr-rot 4*)
apply (*fr-rot-rhs 3*)
apply (*rule fr-refl*)

apply (*fr-rot 1*)
apply (*fr-rot-rhs 1*)
apply (*rule fr-refl*)
apply (*fr-rot 2*)
apply (*fr-rot-rhs 1*)
unfolding *option-assn-alt-def*
apply (*sep-auto split*: *option.split*)
subgoal for $x\ x'$
apply (*subgoal-tac* $m(k \mapsto x) = m$)
apply (*subgoal-tac* $mh(k \mapsto x') = mh$)
using *map-assn-update*[*of A* $m(k := None)$ $mh(k := None)$ $x\ x'\ k$]
by (*auto simp add*: *ent-true-drop*(*1*))
done
show *?thesis*
unfolding *hms-lookup-def hms-assn-def*
apply (*sep-auto eintros del*: *exI*)
subgoal
unfolding *map-assn-def* **by** *auto*
subgoal for *mh*
by (*rule exI*[**where** $x = mh$]) *sep-auto*
by (*sep-auto intro*: *1*)
qed

end

context *imp-map-update*
begin

lemma *hms-update-hnr*:
(*uncurry2 hms-update*, *uncurry2* (*RETURN ooo op-map-update*)) ∈
id-assn^k $*_a$ A^d $*_a$ (*hms-assn A*)^d $\to_a$ *hms-assn A*
by *sepref-to-hoare sep-auto*

sepref-decl-impl *update*: *hms-update-hnr* **uses** *op-map-update.fref*[**where** $V = Id$] **.**

end

context *imp-map-empty*
begin

lemma *hms-empty-hnr*:
(*uncurry0 hms-empty*, *uncurry0* (*RETURN op-map-empty*)) ∈ *unit-assn*^{k} $\to_a$ *hms-assn A*

by *sepref-to-hoare sep-auto*

sepref-decl-impl (*no-register*) *empty*: *hms-empty-hnr* **uses** *op-map-empty.fref*[**where** $V = Id$] **.**

definition *op-hms-empty* ≡ *IICF-Map.op-map-empty*

sublocale *hms*: *map-custom-empty op-hms-empty*
by *unfold-locales* (*simp add*: *op-hms-empty-def*)

lemmas [*sepref-fr-rules*] = *empty-hnr*[*folded op-hms-empty-def*]

lemmas *hms-fold-custom-empty* = *hms.fold-custom-empty*

end

sepref-decl-op *map-extract*:
$\lambda k\ m.\ (m\ k,\ m(k := None)) :: K \to \langle K,V\rangle$*map-rel* $\to \langle V\rangle$*option-rel* $\times_r$ $\langle K,V\rangle$*map-rel*
where *single-valued K single-valued* (K^{-1})
apply (*rule fref-ncI*)
apply *parametricity*
unfolding *map-rel-def*
apply (*elim IntE*)
apply *parametricity*
apply (*elim IntE*; *rule IntI*)
apply *parametricity*
apply (*simp add*: *pres-eq-iff-svb*; *fail*)
by *auto*

context *imp-map-extract-derived*
begin

lemma *hms-extract-hnr*:
(*uncurry* (*hms-extract lookup delete*), *uncurry* (*RETURN oo op-map-extract*))
∈
id-assn^k $*_a$ (*hms-assn A*)^d $\to_a$ *prod-assn* (*option-assn A*) (*hms-assn A*)
by *sepref-to-hoare sep-auto*

lemma *hms-lookup-hnr*:
(*uncurry* (*hms-lookup lookup copy*), *uncurry* (*RETURN oo op-map-lookup*))
∈
id-assn^k $*_a$ (*hms-assn A*)^k $\to_a$ *option-assn A* **if** (*copy*, *RETURN o COPY*)

$\in A^k \rightarrow_a A$
using *that* **by** *sepref-to-hoare sep-auto*

sepref-decl-impl *extract*: *hms-extract-hnr* **uses** *op-map-extract.fref*[**where** $V = Id$] **.**

end

interpretation *hms-hm*: *imp-map-extract-derived is-hashmap hm-delete hm-lookup* **by** *standard*

end
theory *Unified-PW-Impl*
imports *Refine-Imperative-HOL.IICF Unified-PW-Hashing Heap-Hash-Map*
begin

3.5 Imperative Implementation

We now obtain an imperative implementation using the Sepref tool. We will implement the waiting list as a HOL list and the passed set as an imperative hash map.

context notes [*split*!] = *list.split* **begin**
sepref-decl-op *list-hdtl*: $\lambda\ (x\ \#\ xs) \Rightarrow (x,\ xs) :: [\lambda l.\ l \neq []]_f\ \langle A \rangle$*list-rel* $\rightarrow A \times_r \langle A \rangle$*list-rel*
by *auto*
end

context *Worklist-Map2-Defs*
begin

definition *trace* **where**
trace $\equiv \lambda$*type a*. *RETURN* ()

definition
explored-string = *''Explored''*

definition
final-string = *''Final''*

definition
added-string = *''Add''*

definition

```
  subsumed-string = ''Subsumed''

definition
  empty-string = ''Empty''

lemma add-pw'-map2-alt-def:
  add-pw'-map2 passed wait a = do {
   trace explored-string a;
   nfoldli (succs a) (λ(-, -, brk). ¬brk)
    (λa (passed, wait, -).
      do {
      RETURN (
        if empty a then
            (passed, wait, False)
        else if F' a then (passed, wait, True)
        else
          let
            k = key a;
            (v, passed) = op-map-extract k passed
          in
            case v of
              None ⇒ (passed(k ↦ {COPY a}), a # wait, False) |
              Some passed' ⇒
                if ∃ x ∈ passed'. a ≤ x then
                  (passed(k ↦ passed'), wait, False)
                else
                  (passed(k ↦ (insert (COPY a) passed')), a # wait, False)
      )
      }
    )
    (passed,wait,False)
  }
  unfolding add-pw'-map2-def id-def op-map-extract-def trace-def
  apply simp
  apply (fo-rule fun-cong)
  apply (fo-rule arg-cong)
  apply (rule ext)+
  by (auto 4 3 simp: Let-def split: option.split)

  lemma add-pw'-map2-full-trace-def:
  add-pw'-map2 passed wait a = do {
   trace explored-string a;
   nfoldli (succs a) (λ(-, -, brk). ¬brk)
    (λa (passed, wait, -).
```

```
      do {
        if empty a then
            do {trace empty-string a; RETURN (passed, wait, False)}
          else if F' a then do {trace final-string a; RETURN (passed, wait,
True)}
        else
          let
            k = key a;
            (v, passed) = op-map-extract k passed
          in
            case v of
            None ⇒ do {trace added-string a; RETURN (passed(k ↦ {COPY
a}), a # wait, False)} |
              Some passed' ⇒
                if ∃ x ∈ passed'. a ⊴ x then
                 do {trace subsumed-string a; RETURN (passed(k ↦ passed'),
wait, False)}
                else do {
                    trace added-string a;
                     RETURN (passed(k ↦ (insert (COPY a) passed')), a #
wait, False)
                  }
      }
    )
    (passed,wait,False)
  }
    unfolding add-pw'-map2-alt-def
    unfolding trace-def
    apply (simp add:)
    apply (fo-rule fun-cong)
    apply (fo-rule arg-cong)
    apply (rule ext)+
    apply (auto simp add: Let-def split: option.split)
    done

end

locale Worklist-Map2-Impl =
   Worklist4-Impl + Worklist-Map2-Impl-Defs + Worklist-Map2 +
  fixes K
  assumes [sepref-fr-rules]: (keyi,RETURN o PR-CONST key) ∈ A^k →_a
K
  assumes [sepref-fr-rules]: (copyi, RETURN o COPY) ∈ A^k →_a A
  assumes [sepref-fr-rules]: (uncurry tracei,uncurry trace) ∈ id-assn^k *_a A^k
```

$\rightarrow_a$ *id-assn*
assumes *pure-K*: *is-pure K*
assumes *left-unique-K*: *IS-LEFT-UNIQUE* (*the-pure K*)
assumes *right-unique-K*: *IS-RIGHT-UNIQUE* (*the-pure K*)
begin
sepref-register
PR-CONST a_0 *PR-CONST F' PR-CONST* ($\trianglelefteq$) *PR-CONST succs PR-CONST empty PR-CONST key*
PR-CONST F trace

lemma [*def-pat-rules*]:
$a_0 \equiv$ *UNPROTECT* a_0 *F'* $\equiv$ *UNPROTECT F'* ($\trianglelefteq$) $\equiv$ *UNPROTECT* ($\trianglelefteq$) *succs* $\equiv$ *UNPROTECT succs*
empty $\equiv$ *UNPROTECT empty keyi* $\equiv$ *UNPROTECT keyi F* $\equiv$ *UNPROTECT F key* $\equiv$ *UNPROTECT key*
by *simp-all*

lemma *take-from-list-alt-def*:
take-from-list xs = *do* {- $\leftarrow$ *ASSERT* (*xs* $\neq$ []); *RETURN* (*hd-tl xs*)}
unfolding *take-from-list-def* **by** (*auto simp*: *pw-eq-iff refine-pw-simps*)

lemma [*safe-constraint-rules*]: *CN-FALSE is-pure A* $\Longrightarrow$ *is-pure A* **by** *simp*

lemmas [*sepref-fr-rules*] = *hd-tl-hnr*

lemmas [*safe-constraint-rules*] = *pure-K left-unique-K right-unique-K*

lemma [*sepref-import-param*]:
(*explored-string*, *explored-string*) $\in$ *Id*
(*subsumed-string*, *subsumed-string*) $\in$ *Id*
(*added-string*, *added-string*) $\in$ *Id*
(*final-string*, *final-string*) $\in$ *Id*
(*empty-string*, *empty-string*) $\in$ *Id*
unfolding
explored-string-def subsumed-string-def added-string-def final-string-def empty-string-def
by *simp+*

lemmas [*sepref-opt-simps*] =
explored-string-def
subsumed-string-def
added-string-def
final-string-def

empty-string-def

sepref-thm *pw-algo-map2-impl* **is**
uncurry0 (*do* {(*r*, *p*) ← *pw-algo-map2*; *RETURN r*}) :: *unit-assn*^{k} →$_a$ *bool-assn*
unfolding *pw-algo-map2-def add-pw'-map2-full-trace-def PR-CONST-def TRACE'-def*[*symmetric*]
supply [[*goals-limit* = *1*]]
supply *conv-to-is-Nil*[*simp*]
unfolding *fold-lso-bex*
unfolding *take-from-list-alt-def*
apply (*rewrite* **in** {a_0} *lso-fold-custom-empty*)
unfolding *hm.hms-fold-custom-empty*
apply (*rewrite* **in** [a_0] *HOL-list.fold-custom-empty*)
apply (*rewrite* **in** {} *lso-fold-custom-empty*)
unfolding *F-split*
by *sepref*

concrete-definition (**in** −) *pw-impl*
for *Lei* a_0*i Fi succsi emptyi*
uses *Worklist-Map2-Impl.pw-algo-map2-impl.refine-raw* **is** (*uncurry0 ?f*,-)∈-

end — Worklist Map2 Impl

locale *Worklist-Map2-Impl-finite* = *Worklist-Map2-Impl* + *Worklist-Map2-finite*
begin

lemma *pw-algo-map2-correct'*:
(*do* {(*r*, *p*) ← *pw-algo-map2*; *RETURN r*}) ≤ *SPEC* (λ*brk*. *brk* = *F-reachable*)
using *pw-algo-map2-correct*
apply *clarsimp*
apply (*cases pw-algo-map2*)
apply (*solves simp*)
unfolding *RETURN-def*
apply *clarsimp*
subgoal for *X*
apply (*cases do* {(*r*, *p*) ← *RES X*; *RES* {*r*}})
apply (*subst* (*asm*) *Refine-Basic.bind-def*, *force*)
subgoal premises *prems* **for** *X'*
proof −
have *r* = *F-reachable* **if** (*r*, *p*) ∈ *X* **for** *r p*
using *that prems*(*1*) **by** *auto*
then show *?thesis*
by (*auto simp*: *pw-le-iff refine-pw-simps*)

qed
done
done

lemma *pw-impl-hnr-F-reachable*:
(*uncurry0* (*pw-impl keyi copyi tracei Lei* $a_0 i$ *Fi succsi emptyi*), *uncurry0* (*RETURN F-reachable*))
$\in$ *unit-assn*^{k} $\rightarrow_a$ *bool-assn*
using
pw-impl.refine[
OF Worklist-Map2-Impl-axioms,
FCOMP pw-algo-map2-correct'[*THEN Id-SPEC-refine*, *THEN nres-relI*]
]
by (*simp add*: *RETURN-def*)

end

locale *Worklist-Map2-Hashable* =
Worklist-Map2-Impl-finite
begin

sepref-decl-op *F-reachable* :: *bool-rel* .
lemma [*def-pat-rules*]: *F-reachable* $\equiv$ *op-F-reachable* **by** *simp*

lemma *hnr-op-F-reachable*:
assumes *GEN-ALGO* $a_0 i$ ($\lambda a_0 i$. (*uncurry0* $a_0 i$, *uncurry0* (*RETURN* a_0)) $\in$ *unit-assn*^{k} $\rightarrow_a$ *A*)
assumes *GEN-ALGO Fi* (λ*Fi*. (*Fi*,*RETURN o F'*) $\in$ A^k $\rightarrow_a$ *bool-assn*)
assumes *GEN-ALGO Lei* (λ*Lei*. (*uncurry Lei*,*uncurry* (*RETURN oo* ($\preceq$))) $\in$ A^k $*_a$ A^k $\rightarrow_a$ *bool-assn*)
assumes *GEN-ALGO succsi* (λ*succsi*. (*succsi*,*RETURN o succs*) $\in$ A^k $\rightarrow_a$ *list-assn A*)
assumes *GEN-ALGO emptyi* (λ*Fi*. (*Fi*,*RETURN o empty*) $\in$ A^k $\rightarrow_a$ *bool-assn*)
assumes [*sepref-fr-rules*]: (*keyi*,*RETURN o PR-CONST key*) $\in$ A^k $\rightarrow_a$ *K*
assumes [*sepref-fr-rules*]: (*copyi*, *RETURN o COPY*) $\in$ A^k $\rightarrow_a$ *A*
shows
(*uncurry0* (*pw-impl keyi copyi tracei Lei* $a_0 i$ *Fi succsi emptyi*),
uncurry0 (*RETURN* (*PR-CONST op-F-reachable*)))
$\in$ *unit-assn*^{k} $\rightarrow_a$ *bool-assn*
proof −
from *assms* **interpret**

Worklist-Map2-Impl
E a_0 *F* ($\preceq$) *succs empty* ($\trianglelefteq$) *F'* *A succsi* a_0*i Fi Lei emptyi key keyi copyi*
by (*unfold-locales*; *simp add*: *GEN-ALGO-def*)

from *pw-impl-hnr-F-reachable* **show** *?thesis* **by** *simp*
qed

sepref-decl-impl *hnr-op-F-reachable* **.**

end — Worklist Map 2

end — End of Theory

4 Generic Worklist Algorithm With Subsumption

theory *Worklist-Subsumption-Multiset*
imports
Refine-Imperative-HOL.Sepref
Worklist-Algorithms-Misc
Worklist-Locales
Unified-PW — only for shared definitions
begin

This section develops an implementation of the worklist algorithm for reachability without a shared passed-waiting list. The obtained imperative implementation may be less efficient for the purpose of timed automata model checking but the variants obtained from the refinement steps are more general and could serve a wider range of future use cases.

4.1 Utilities

definition *take-from-set* **where**
take-from-set s = *ASSERT* ($s \neq \{\}$) $\gg$ *SPEC* (λ (x, s'). $x \in s \wedge s' = s - \{x\}$)

lemma *take-from-set-correct*:
assumes $s \neq \{\}$
shows *take-from-set s* $\leq$ *SPEC* (λ (x, s'). $x \in s \wedge s' = s - \{x\}$)
using *assms* **unfolding** *take-from-set-def* **by** *simp*

lemmas [*refine-vcg*] = *take-from-set-correct*[*THEN order.trans*]

definition *take-from-mset* **where**
take-from-mset s = *ASSERT* (*s* ≠ {#}) ≫ *SPEC* (λ (*x*, *s'*). *x* ∈# *s* ∧ *s'* = *s* − {#*x*#})

lemma *take-from-mset-correct*:
assumes *s* ≠ {#}
shows *take-from-mset s* ≤ *SPEC* (λ (*x*, *s'*). *x* ∈# *s* ∧ *s'* = *s* − {#*x*#})
using *assms* **unfolding** *take-from-mset-def* **by** *simp*

lemmas [*refine-vcg*] = *take-from-mset-correct*[*THEN order.trans*]

lemma *set-mset-mp*: *set-mset m* ⊆ *s* ⟹ *n* < *count m x* ⟹ *x*∈*s*
by (*meson count-greater-zero-iff le-less-trans subsetCE zero-le*)

lemma *pred-not-lt-is-zero*: (¬ *n* − *Suc 0* < *n*) ⟷ *n*=*0* **by** *auto*

lemma (**in** *Search-Space-finite-strict*) *finitely-branching*:
assumes *reachable a*
shows *finite* (*Collect* (*E a*))
by (*metis assms finite-reachable finite-subset mem-Collect-eq reachable-step subsetI*)

4.2 Standard Worklist Algorithm

context *Search-Space-Defs-Empty* **begin**

definition *worklist-start-subsumed passed wait* = (∃ *a* ∈ *passed* ∪ *set-mset wait*. $a_0 \preceq a$)

definition
worklist-var =
inv-image (*finite-psupset* (*Collect reachable*) <∗*lex*∗> *measure size*) (λ (*a*, *b*,*c*). (*a*,*b*))

definition *worklist-inv-frontier passed wait* =
(∀ *a* ∈ *passed*. ∀ *a'*. *E a a'* ∧ ¬ *empty a'* ⟶ (∃ *b'* ∈ *passed* ∪ *set-mset wait*. $a' \preceq b'$))

definition *worklist-inv* ≡ λ (*passed*, *wait*, *brk*).
passed ⊆ *Collect reachable* ∧
(*brk* ⟶ (∃ *f*. *reachable f* ∧ *F f*)) ∧

```
  (¬ brk ⟶
    worklist-inv-frontier passed wait
  ∧ (∀ a ∈ passed ∪ set-mset wait. ¬ F a)
  ∧ worklist-start-subsumed passed wait
  ∧ set-mset wait ⊆ Collect reachable)


 definition add-succ-spec wait a ≡ SPEC (λ(wait',brk).
  if ∃ a'. E a a' ∧ F a' then
    brk
  else
    ¬brk ∧ set-mset wait' ⊆ set-mset wait ∪ {a' . E a a'} ∧
     (∀ s ∈ set-mset wait ∪ {a' . E a a' ∧ ¬ empty a'}. ∃ s' ∈ set-mset
wait'. s ≼ s')
 )

 definition worklist-algo where
  worklist-algo = do
   {
    if F a₀ then RETURN True
    else do {
      let passed = {};
      let wait = {#a₀#};
     (passed, wait, brk) ← WHILEIT worklist-inv (λ (passed, wait, brk).
¬ brk ∧ wait ≠ {#})
        (λ (passed, wait, brk). do
          {
            (a, wait) ← take-from-mset wait;
            ASSERT (reachable a);
            if (∃ a' ∈ passed. a ≼ a') then RETURN (passed, wait, brk)
else
            do
              {
                (wait,brk) ← add-succ-spec wait a;
                let passed = insert a passed;
                RETURN (passed, wait, brk)
              }
          }
        )
        (passed, wait, False);
        RETURN brk
    }
   }
```

end

Correctness Proof **lemma** (**in** *Search-Space*) *empty-E-star*:
empty x' **if** E^{**} x x' *reachable x empty x*
using *that* **unfolding** *reachable-def*
by (*induction rule*: *converse-rtranclp-induct*)
(*blast intro*: *empty-E*[*unfolded reachable-def*] *rtranclp.rtrancl-into-rtrancl*)+

context *Search-Space-finite-strict* **begin**

lemma *wf-worklist-var*:
wf worklist-var
unfolding *worklist-var-def* **by** (*auto simp*: *finite-reachable*)

context
begin

private lemma *aux1*:
assumes $\forall\, x{\in}passed.\ \neg\ a \preceq x$
and *passed* $\subseteq$ *Collect reachable*
and *reachable a*
shows
((*insert a passed*, *wait'*, *brk'*),
passed, *wait*, *brk*)
$\in$ *worklist-var*
proof −
from *assms* **have** $a \notin passed$ **by** *auto*
with *assms*(*2*,*3*) **show** *?thesis*
by (*auto simp*: *worklist-inv-def worklist-var-def finite-psupset-def*)
qed

private lemma *aux2*:
assumes
$a' \in passed$
$a \preceq a'$
$a \in\#\ wait$
worklist-inv-frontier passed wait
shows *worklist-inv-frontier passed* (*wait* − {#*a*#})
using *assms* **unfolding** *worklist-inv-frontier-def*
using *trans*
apply *clarsimp*
by (*metis* (*no-types*, *lifting*) *Un-iff count-eq-zero-iff count-single mset-contains-eq*

mset-un-cases)

private lemma *aux5*:
assumes
$a' \in$ *passed*
$a \preceq a'$
$a \in\#$ *wait*
worklist-start-subsumed passed wait
shows *worklist-start-subsumed passed* (*wait* − {#*a*#})
using *assms* **unfolding** *worklist-start-subsumed-def* **apply** *clarsimp*
by (*metis Un-iff insert-DiffM2 local.trans mset-right-cancel-elem*)

private lemma *aux3*:
assumes
set-mset wait ⊆ *Collect reachable*
a ∈# *wait*
∀ *s* ∈ *set-mset* (*wait* − {#*a*#}) ∪ {*a'*. *E a a'* ∧ ¬ *empty a'*}. ∃ *s'* ∈ *set-mset wait'*. *s* ⪯ *s'*
worklist-inv-frontier passed wait
shows *worklist-inv-frontier* (*insert a passed*) *wait'*
proof −
from *assms*(*1*,*2*) **have** *reachable a*
by (*simp add*: *subset-iff*)
with *finitely-branching* **have** [*simp*, *intro*!]: *finite* (*Collect* (*E a*)) **.**

show *?thesis* **unfolding** *worklist-inv-frontier-def*
apply *safe*
subgoal
using *assms* **by** *auto*
subgoal for *b b'*
proof −
assume *A*: *E b b'* ¬ *empty b' b* ∈ *passed*
with *assms* **obtain** *b''* **where** *b''*: *b''* ∈ *passed* ∪ *set-mset wait b'* ⪯ *b''*
unfolding *worklist-inv-frontier-def* **by** *blast*
from *this*(*1*) **show** *?thesis*
apply *standard*
subgoal
using ‹*b'* ⪯ *b''*› **by** *auto*
subgoal
apply (*cases a* = *b''*)
subgoal
using *b''*(*2*) **by** *blast*
subgoal premises *prems*

```
          proof −
            from prems have b'' ∈# wait − {#a#}
              by (auto simp: mset-remove-member)
            with assms prems ‹b' ≼ b''› show ?thesis
              by (blast intro: local.trans)
          qed
          done
        done
      qed
      done
  qed

  private lemma aux6:
    assumes
      a ∈# wait
      worklist-start-subsumed passed wait
      ∀ s ∈ set-mset (wait − {#a#}) ∪ {a'. E a a' ∧ ¬ empty a'}. ∃ s' ∈
set-mset wait'. s ≼ s'
    shows worklist-start-subsumed (insert a passed) wait'
    using assms unfolding worklist-start-subsumed-def
    apply clarsimp
    apply (erule disjE)
     apply blast
    subgoal premises prems for x
    proof (cases a = x)
      case True
      with prems show ?thesis by simp
    next
      case False
      with ‹x ∈# wait› have x ∈ set-mset (wait − {#a#})
        by (metis insert-DiffM insert-noteq-member prems(1))
      with prems(2,4) ‹- ≼ x› show ?thesis
        by (auto dest: trans)
    qed
  done

  lemma aux4:
   assumes worklist-inv-frontier passed {#} reachable x worklist-start-subsumed
passed {#}
          passed ⊆ Collect reachable
    shows ∃ x' ∈ passed. x ≼ x'
  proof −
    from ‹reachable x› have E** a0 x by (simp add: reachable-def)
     from assms(3) obtain b where a0 ≼ b b ∈ passed unfolding work-
```

list-start-subsumed-def **by** *auto*
have $\exists x'.\ \exists\ x''.\ E^{**}\ b\ x' \wedge x \preceq x' \wedge x' \preceq x'' \wedge x'' \in \mathit{passed}$ **if**
$E^{**}\ a\ x\ \ a \preceq b\ \ \ b \preceq b'\ \ \ b' \in \mathit{passed}$
reachable a reachable b **for** *a b b'*
using *that* **proof** (*induction arbitrary*: *b b' rule*: *converse-rtranclp-induct*)
case *base*
then show *?case* **by** *auto*
next
case (*step a a1 b b'*)
show *?case*
proof (*cases empty a*)
case *True*
with *step.prems step.hyps* **have** *empty x* **by** − (*rule empty-E-star*, *auto*)
with *step.prems* **show** *?thesis* **by** (*auto intro*: *empty-subsumes*)
next
case *False*
with ‹*E a a1*› ‹$a \preceq b$› ‹*reachable a*› ‹*reachable b*› **obtain** *b1* **where**
E b b1 a1 $\preceq$ *b1*
using *mono* **by** *blast*
show *?thesis*
proof (*cases empty b1*)
case *True*
with *empty-mono* ‹$a1 \preceq b1$› **have** *empty a1* **by** *blast*
with *step.prems step.hyps* **have** *empty x* **by** − (*rule empty-E-star*, *auto simp*: *reachable-def*)
with *step.prems* **show** *?thesis* **by** (*auto intro*: *empty-subsumes*)
next
case *False*
from ‹*E b b1*› ‹$a1 \preceq b1$› **obtain** *b1'* **where** *E b' b1' b1* $\preceq$ *b1'*
using ‹¬ *empty a*› *empty-mono assms*(*4*) *mono step.prems* **by** *blast*
from *empty-mono*[*OF* ‹¬ *empty b1*› ‹$b1 \preceq b1'$›] **have** ¬ *empty b1'*
by *auto*
with ‹*E b' b1'*› ‹$b' \in \mathit{passed}$› *assms*(*1*) **obtain** *b1''* **where** *b1''* ∈ *passed b1'* $\preceq$ *b1''*
unfolding *worklist-inv-frontier-def* **by** *auto*
with ‹$b1 \preceq$ -› **have** *b1* $\preceq$ *b1''* **using** *trans* **by** *blast*
with *step.IH*[*OF* ‹$a1 \preceq b1$› *this* ‹*b1''* ∈ *passed*›] ‹*reachable a*› ‹*E a a1*› ‹*reachable b*› ‹*E b b1*›
obtain *x' x''* **where**
*E*** *b1 x' x* $\preceq$ *x' x'* $\preceq$ *x'' x''* ∈ *passed*
by *auto*
moreover from ‹*E b b1*› ‹*E*** *b1 x'*› **have** *E*** *b x'* **by** *auto*

```
        ultimately show ?thesis by auto
      qed
    qed
  qed
  from this[OF ‹E** a0 x› ‹a0 ≼ b› refl ‹b ∈ -›] assms(4) ‹b ∈ passed›
show ?thesis
    by (auto intro: trans)
 qed

 theorem worklist-algo-correct:
  worklist-algo ≤ SPEC (λ brk. brk ⟷ F-reachable)
 proof −
  note [simp] = size-Diff-submset pred-not-lt-is-zero
  note [dest] = set-mset-mp
  show ?thesis
  unfolding worklist-algo-def add-succ-spec-def F-reachable-def
   apply (refine-vcg wf-worklist-var)

   apply (solves auto)

   apply (solves ‹auto simp:
    worklist-inv-def worklist-inv-frontier-def worklist-start-subsumed-def›)

   apply (simp; fail)


    apply (solves ‹auto simp: worklist-inv-def›)

    apply (solves ‹auto simp: worklist-inv-def aux2 aux5
          dest: in-diffD
          split: if-split-asm›)

    apply (solves
      ‹auto simp: worklist-inv-def worklist-var-def intro: finite-subset[OF
- finite-reachable]›)



    apply (clarsimp split: if-split-asm)

     apply (clarsimp simp: worklist-inv-def; blast)

   apply (auto
    simp: worklist-inv-def aux3 aux6 finitely-branching
```

dest: *in-diffD*)[]
apply (*metis Un-iff in-diffD insert-subset mem-Collect-eq mset-add set-mset-add-mset-insert*)
subgoal for *ab aaa baa aba aca - x*
proof −

assume
reachable aba
set-mset aca ⊆ *set-mset* (*aaa* − {#*aba*#}) ∪ *Collect* (*E aba*)
set-mset aaa ⊆ *Collect reachable*
x ∈# *aca*
then show *?thesis*
by (*auto dest*: *in-diffD*)
qed
apply (*solves* ‹*auto simp*: *worklist-inv-def aux1*›)

using *F-mono* **by** (*fastforce simp*: *worklist-inv-def dest*!: *aux4*)
qed

lemmas [*refine-vcg*] = *worklist-algo-correct*[*THEN Orderings.order.trans*]

end — Context

end — Search Space

context *Search-Space″-Defs*
begin

definition *worklist-inv-frontier′ passed wait* =
(∀ *a* ∈ *passed*. ∀ *a′*. *E a a′* ∧ ¬ *empty a′* ⟶ (∃ *b′* ∈ *passed* ∪ *set-mset wait*. *a′* ≼ *b′*))

definition *worklist-start-subsumed′ passed wait* = (∃ *a* ∈ *passed* ∪ *set-mset wait*. *a*$_0$ ≼ *a*)

definition *worklist-inv′* ≡ λ (*passed*, *wait*, *brk*).
worklist-inv (*passed*, *wait*, *brk*) ∧ (∀ *a* ∈ *passed*. ¬ *empty a*) ∧ (∀ *a* ∈ *set-mset wait*. ¬ *empty a*)

definition *add-succ-spec′ wait a* ≡ *SPEC* (λ(*wait′*,*brk*).
(

```
    if ∃ a'. E a a' ∧ F a' then
        brk
      else
        ¬brk ∧ set-mset wait' ⊆ set-mset wait ∪ {a' . E a a'} ∧
        (∀ s ∈ set-mset wait ∪ {a' . E a a' ∧ ¬ empty a'}. ∃ s' ∈ set-mset
wait'. s ≼ s')
    ) ∧ (∀ s ∈ set-mset wait'. ¬ empty s)
    )

  definition worklist-algo' where
    worklist-algo' = do
      {
        if F a0 then RETURN True
        else do {
          let passed = {};
          let wait = {#a0#};
         (passed, wait, brk) ← WHILEIT worklist-inv' (λ (passed, wait, brk).
¬ brk ∧ wait ≠ {#})
            (λ (passed, wait, brk). do
              {
                (a, wait) ← take-from-mset wait;
                ASSERT (reachable a);
                if (∃ a' ∈ passed. a ≼ a') then RETURN (passed, wait, brk)
else
                do
                  {
                    (wait,brk) ← add-succ-spec' wait a;
                    let passed = insert a passed;
                    RETURN (passed, wait, brk)
                  }
              }
            )
            (passed, wait, False);
            RETURN brk
        }
      }


end — Search Space" Defs


context Search-Space''-start
begin
```

lemma *worklist-algo-list-inv-ref*[*refine*]:
fixes $x\ x'$
assumes
$\neg\ F\ a_0\ \neg\ F\ a_0$
$(x,\ x') \in \{((passed, wait, brk),\ (passed', wait', brk'))$.
$passed = passed' \wedge\ wait = wait' \wedge\ brk = brk' \wedge\ (\forall\ a \in passed.\ \neg$
empty $a)$
$\wedge\ (\forall\ a \in$ *set-mset wait*. $\neg$ *empty* $a)\}$
worklist-inv x'
shows *worklist-inv'* x
using *assms*
unfolding *worklist-inv'-def worklist-inv-def*
by *auto*

lemma [*refine*]:
take-from-mset wait $\leq$
$\Downarrow\ \{((x,\ wait),\ (y,\ wait'))$. $x = y \wedge\ wait = wait' \wedge\ \neg$ *empty* $x \wedge\ (\forall\ a \in$
set-mset wait. $\neg$ *empty* $a)\}$
(*take-from-mset wait'*)
if $wait = wait' \forall\ a \in$ *set-mset wait*. $\neg$ *empty* a *wait* $\neq \{\#\}$
using *that*
by (*auto 4 5 simp*: *pw-le-iff refine-pw-simps dest*: *in-diffD dest*!: *take-from-mset-correct*)

lemma [*refine*]:
add-succ-spec' *wait* $x \leq$
$\Downarrow\ (\{(wait,\ wait')$. $wait = wait' \wedge\ (\forall\ a \in$ *set-mset wait*. $\neg$ *empty* $a)\} \times_r$
bool-rel)
(*add-succ-spec wait'* x')
if $wait = wait'\ x = x' \forall\ a \in$ *set-mset wait*. $\neg$ *empty* a
using *that*
unfolding *add-succ-spec'-def add-succ-spec-def*
by (*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *worklist-algo'-ref*[*refine*]: *worklist-algo'* $\leq \Downarrow Id$ *worklist-algo*
using [[*goals-limit=15*]]
unfolding *worklist-algo'-def worklist-algo-def*
apply (*refine-rcg*)
prefer *3*
apply *assumption*
apply *refine-dref-type*
by (*auto simp*: *empty-subsumes'*)

end — Search Space" Start

context *Search-Space″-Defs*
begin

definition *worklist-algo″* **where**
worklist-algo″ ≡
if empty a_0 *then RETURN False else worklist-algo′*

end — Search Space" Defs

context *Search-Space″-finite-strict*
begin

lemma *worklist-algo″-correct*:
worklist-algo″ ≤ *SPEC* (λ *brk*. *brk* ⟷ *F-reachable*)
proof (*cases empty* a_0)
case *True*
then show *?thesis*
unfolding *worklist-algo″-def F-reachable-def reachable-def*
using *empty-E-star final-non-empty* **by** *auto*
next
case *False*
interpret *Search-Space″-start*
by *standard* (*rule* ‹¬ *empty* -›)
note *worklist-algo′-ref*
also note *worklist-algo-correct*
finally show *?thesis*
using *False* **unfolding** *worklist-algo″-def* **by** *simp*
qed

end — Search Space" (strictly finite)

end — End of Theory

theory *Worklist-Subsumption1*
imports *Worklist-Subsumption-Multiset*
begin

4.3 From Multisets to Lists

Utilities **definition** *take-from-list* **where**

take-from-list s = *ASSERT* $(s \neq [])$ $\gg$ *SPEC* $(\lambda\ (x,\ s').\ s = x\ \#\ s')$

lemma *take-from-list-correct*:
assumes $s \neq []$
shows *take-from-list* $s \leq$ *SPEC* $(\lambda\ (x,\ s').\ s = x\ \#\ s')$
using *assms* **unfolding** *take-from-list-def* **by** *simp*

lemmas [*refine-vcg*] = *take-from-list-correct*[*THEN order.trans*]

context *Search-Space-Defs-Empty*
begin

definition *worklist-inv-frontier-list passed wait* =
($\forall$ $a \in$ *passed*. $\forall$ a'. *E a a'* $\wedge$ $\neg$ *empty a'* $\longrightarrow$ ($\exists$ $b' \in$ *passed* $\cup$ *set wait*. $a' \preceq b'$))

definition *start-subsumed-list passed wait* = ($\exists$ $a \in$ *passed* $\cup$ *set wait*. $a_0 \preceq a$)

definition *worklist-inv-list* $\equiv \lambda$ (*passed*, *wait*, *brk*).
passed $\subseteq$ *Collect reachable* $\wedge$
(*brk* $\longrightarrow$ ($\exists$ *f*. *reachable f* $\wedge$ *F f*)) $\wedge$
($\neg$ *brk* $\longrightarrow$
worklist-inv-frontier-list passed wait
$\wedge$ ($\forall$ $a \in$ *passed* $\cup$ *set wait*. $\neg$ *F a*)
$\wedge$ *start-subsumed-list passed wait*
$\wedge$ *set wait* $\subseteq$ *Collect reachable*)
$\wedge$ ($\forall$ $a \in$ *passed*. $\neg$ *empty a*) $\wedge$ ($\forall$ $a \in$ *set wait*. $\neg$ *empty a*)

definition *add-succ-spec-list wait a* $\equiv$ *SPEC* $(\lambda$(*wait'*,*brk*).
(
if $\exists$ *a'*. *E a a'* $\wedge$ *F a'* *then*
brk
else
$\neg$*brk* $\wedge$ *set wait'* $\subseteq$ *set wait* $\cup$ $\{a'$. *E a a'*$\}$ $\wedge$
($\forall$ $s \in$ *set wait* $\cup$ $\{a'$. *E a a'* $\wedge$ $\neg$ *empty a'*$\}$. $\exists$ $s' \in$ *set wait'*. $s \preceq s'$)
) $\wedge$ ($\forall$ $s \in$ *set wait'*. $\neg$ *empty s*)
)

end — Search Space Empty Defs

context *Search-Space''-Defs*

begin
definition *worklist-algo-list* **where**
worklist-algo-list = *do*
{
if F a_0 *then RETURN True*
else do {
let passed = {};
let wait = [a_0];
(*passed*, *wait*, *brk*) ← *WHILEIT worklist-inv-list* (λ (*passed*, *wait*, *brk*). ¬ *brk* ∧ *wait* ≠ [])
(λ (*passed*, *wait*, *brk*). *do*
{
(*a*, *wait*) ← *take-from-list wait*;
ASSERT (*reachable a*);
if (∃ *a'* ∈ *passed*. *a* ⊴ *a'*) *then RETURN* (*passed*, *wait*, *brk*) *else*
do
{
(*wait*,*brk*) ← *add-succ-spec-list wait a*;
let passed = *insert a passed*;
RETURN (*passed*, *wait*, *brk*)
}
}
)
(*passed*, *wait*, *False*);
RETURN brk
}
}

end — Search Space" Defs

context *Search-Space''-pre*
begin

lemma *worklist-algo-list-inv-ref*:
fixes *x x'*
assumes
¬ *F* a_0 ¬ *F* a_0
(*x*, *x'*) ∈ {((*passed*,*wait*,*brk*), (*passed'*,*wait'*,*brk'*)). *passed* = *passed'* ∧ *mset wait* = *wait'* ∧ *brk* = *brk'* }
worklist-inv' x'
shows *worklist-inv-list x*
using *assms*

unfolding *worklist-inv'-def worklist-inv-def worklist-inv-list-def*
unfolding *worklist-inv-frontier-def worklist-inv-frontier-list-def*
unfolding *worklist-start-subsumed-def start-subsumed-list-def*
by *auto*

lemma *take-from-list-take-from-mset-ref*[*refine*]:
take-from-list xs ≤ ⇓ {((*x*, *xs*),(*y*, *m*)). *x* = *y* ∧ *mset xs* = *m*} (*take-from-mset m*)
if *mset xs* = *m*
using *that* **unfolding** *take-from-list-def take-from-mset-def*
by (*clarsimp simp*: *pw-le-iff refine-pw-simps*)

lemma *add-succ-spec-list-add-succ-spec-ref*[*refine*]:
add-succ-spec-list xs b ≤ ⇓ {((*xs*, *b*), (*m*, *b'*)). *mset xs* = *m* ∧ *b* = *b'*} (*add-succ-spec' m b'*)
if *mset xs* = *m b* = *b'*
using *that* **unfolding** *add-succ-spec-list-def add-succ-spec'-def*
by (*clarsimp simp*: *pw-le-iff refine-pw-simps*)

lemma *worklist-algo-list-ref*[*refine*]: *worklist-algo-list* ≤ ⇓*Id worklist-algo'*
unfolding *worklist-algo-list-def worklist-algo'-def*
apply (*refine-rcg*)
apply *blast*
prefer *2*
apply (*rule worklist-algo-list-inv-ref*; *assumption*)
by *auto*

end — Search Space"

4.4 Towards an Implementation

context *Worklist1-Defs*
begin

definition
add-succ1 wait a ≡
nfoldli (*succs a*) (λ(-,*brk*). ¬*brk*)
(λ*a* (*wait*,*brk*).
do {
ASSERT (∀ *x* ∈ *set wait*. ¬ *empty x*);
if F a then RETURN (*wait*,*True*) *else RETURN* (
if (∃ *x* ∈ *set wait*. *a* ≼ *x* ∧ ¬ *x* ≼ *a*) ∨ *empty a*
then [*x* ← *wait*. ¬ *x* ≼ *a*]
else a # [*x* ← *wait*. ¬ *x* ≼ *a*],*False*)

```
      }
    )
    (wait,False)

end

context Worklist2-Defs
begin

  definition
    add-succ1' wait a ≡
     nfoldli (succs a) (λ(-,brk). ¬brk)
      (λa (wait,brk).
        if F a then RETURN (wait,True) else RETURN (
          if empty a
          then wait
          else if ∃ x ∈ set wait. a ≼ x ∧ ¬ x ≼ a
          then [x ← wait. ¬ x ≼ a]
          else a # [x ← wait. ¬ x ≼ a],False)
    )
    (wait,False)

end

context Worklist1
begin

lemma add-succ1-ref[refine]:
  add-succ1 wait a ≤ ⇓(Id ×r bool-rel) (add-succ-spec-list wait' a')
  if (wait,wait')∈Id (a,a')∈b-rel Id reachable ∀ x ∈ set wait'. ¬ empty x
  using that
  apply simp
  unfolding add-succ-spec-list-def add-succ1-def
  apply (refine-vcg nfoldli-rule[where I =
      λl1 - (wait',brk).
          (if brk then ∃ a'. E a a' ∧ F a'
          else set wait' ⊆ set wait ∪ set l1 ∧ set l1 ∩ Collect F = {}
            ∧ (∀ x ∈ set wait ∪ set l1. ¬ empty x ⟶ (∃ x' ∈ set wait'. x
≼ x')))
          ∧ (∀ x ∈ set wait'. ¬ empty x)]
      )
         apply (solves auto)
        apply (solves auto)
       using succs-correct[of a] apply (solves auto)
```

```
        using succs-correct[of a] apply (solves auto)
       apply (solves auto)
  subgoal
   apply (clarsimp split: if-split-asm; intro conjI)
      apply blast
     apply blast
    apply (meson empty-mono local.trans)
   apply blast
   apply (meson empty-mono local.trans)
   done
 using succs-correct[of a] apply (solves auto)+
 done

end

context Worklist2
begin

 lemma add-succ1'-ref[refine]:
   add-succ1' wait a ≤ ⇓(Id ×r bool-rel) (add-succ1 wait' a')
   if (wait,wait')∈Id (a,a')∈b-rel Id reachable ∀ x ∈ set wait'. ¬ empty x
   unfolding add-succ1'-def add-succ1-def
   using that
   apply (refine-vcg nfoldli-refine)
       apply refine-dref-type
       apply (solves ‹auto simp: empty-subsumes' cong: filter-cong›)+
   apply (simp add: empty-subsumes' cong: filter-cong)
   by (metis empty-mono empty-subsumes' filter-True)


 lemma add-succ1'-ref'[refine]:
   add-succ1' wait a ≤ ⇓(Id ×r bool-rel) (add-succ-spec-list wait' a')
   if (wait,wait')∈Id (a,a')∈b-rel Id reachable ∀ x ∈ set wait'. ¬ empty x
   using that
 proof −
   note add-succ1'-ref
   also note add-succ1-ref
   finally show ?thesis
     using that by − auto
 qed

 definition worklist-algo1' where
   worklist-algo1' = do
     {
```

```
      if F a_0 then RETURN True
      else do {
        let passed = {};
        let wait = [a_0];
        (passed, wait, brk) ← WHILEIT worklist-inv-list (λ (passed, wait,
brk). ¬ brk ∧ wait ≠ [])
          (λ (passed, wait, brk). do
            {
              (a, wait) ← take-from-list wait;
              if (∃ a' ∈ passed. a ⊴ a') then RETURN (passed, wait, brk)
else
              do
                {
                  (wait,brk) ← add-succ1' wait a;
                  let passed = insert a passed;
                  RETURN (passed, wait, brk)
                }
            }
          )
          (passed, wait, False);
          RETURN brk
      }
    }
```

lemma *take-from-list-ref*[*refine*]:
take-from-list wait ≤
⇓ {((*x*, *wait*), (*y*, *wait'*)). *x* = *y* ∧ *wait* = *wait'* ∧ ¬ *empty x* ∧ (∀ *a* ∈ *set wait*. ¬ *empty a*)}
(*take-from-list wait'*)
if *wait* = *wait'* ∀ *a* ∈ *set wait*. ¬ *empty a wait* ≠ []
using *that*
by (*auto 4 4 simp*: *pw-le-iff refine-pw-simps dest*!: *take-from-list-correct*)

lemma *worklist-algo1-list-ref*[*refine*]: *worklist-algo1'* ≤ ⇓*Id worklist-algo-list*
unfolding *worklist-algo1'-def worklist-algo-list-def*
apply (*refine-rcg*)
apply *refine-dref-type*
unfolding *worklist-inv-list-def*
by *auto*

definition *worklist-algo1* **where**
worklist-algo1 ≡ *if empty* a_0 *then RETURN False else worklist-algo1'*

```
  lemma worklist-algo1-ref[refine]: worklist-algo1 ≤ ⇓Id worklist-algo''
  proof −
    have worklist-algo1' ≤ worklist-algo' if ¬ empty a₀
    proof −
      note worklist-algo1-list-ref
      also note worklist-algo-list-ref
      finally show worklist-algo1' ≤ worklist-algo'
        by simp
    qed
    then show ?thesis
      unfolding worklist-algo1-def worklist-algo''-def by simp
  qed

end — Worklist2

context Worklist3-Defs
begin

  definition
    add-succ2 wait a ≡
     nfoldli (succs a) (λ(-,brk). ¬brk)
      (λa (wait,brk).
        if empty a then RETURN (wait, False)
        else if F' a then RETURN (wait,True)
        else RETURN (
          if ∃ x ∈ set wait. a ≼ x ∧ ¬ x ≼ a
          then [x ← wait. ¬ x ≼ a]
          else a # [x ← wait. ¬ x ≼ a],False)
      )
      (wait,False)

  definition
    filter-insert-wait wait a ≡
     if ∃ x ∈ set wait. a ≼ x ∧ ¬ x ≼ a
     then [x ← wait. ¬ x ≼ a]
     else a # [x ← wait. ¬ x ≼ a]

end


context Worklist3
begin

  lemma filter-insert-wait-alt-def:
```

```
  filter-insert-wait wait a = (
    let
       (f, xs) =
         fold (λ x (f, xs). if x ⊴ a then (f, xs) else (f ∨ a ⊴ x, x # xs))
wait (False, [])
     in
       if f then rev xs else a # rev xs
  )

  proof −
    have
      fold (λ x (f, xs). if x ⊴ a then (f, xs) else (f ∨ a ⊴ x, x # xs)) wait
(f, xs)
     = (f ∨ (∃ x ∈ set wait. a ⊴ x ∧ ¬ x ⊴ a), rev [x ← wait. ¬ x ⊴ a] @
xs) for f xs
      by (induction wait arbitrary: f xs; simp)
    then show ?thesis unfolding filter-insert-wait-def by auto
  qed

  lemma add-succ2-alt-def:
    add-succ2 wait a ≡
     nfoldli (succs a) (λ(-,brk). ¬brk)
      (λa (wait,brk).
        if empty a then RETURN (wait, False)
        else if F' a then RETURN (wait, True)
        else RETURN (filter-insert-wait wait a, False)
      )
      (wait,False)
   unfolding add-succ2-def filter-insert-wait-def by (intro HOL.eq-reflection
HOL.refl)

  lemma add-succ2-ref[refine]:
    add-succ2 wait a ≤ ⇓(Id ×r bool-rel) (add-succ1' wait' a')
    if (wait,wait')∈Id (a,a')∈Id
    unfolding add-succ2-def add-succ1'-def
    apply (rule nfoldli-refine)
      apply refine-dref-type
    using that by (auto simp: F-split)

  definition worklist-algo2' where
    worklist-algo2' = do
      {
        if F' a0 then RETURN True
        else do {
```

```
        let passed = {};
        let wait = [a0];
        (passed, wait, brk) ← WHILEIT worklist-inv-list (λ (passed, wait,
brk). ¬ brk ∧ wait ≠ [])
          (λ (passed, wait, brk). do
            {
              (a, wait) ← take-from-list wait;
              if (∃ a' ∈ passed. a ⊴ a') then RETURN (passed, wait, brk)
else
              do
                {
                  (wait,brk) ← add-succ2 wait a;
                  let passed = insert a passed;
                  RETURN (passed, wait, brk)
                }
            }
          )
          (passed, wait, False);
          RETURN brk
      }
    }
```

lemma *worklist-algo2'-ref*[*refine*]: *worklist-algo2'* ≤ ⇓*Id worklist-algo1'* **if** ¬ *empty* a_0
unfolding *worklist-algo2'-def worklist-algo1'-def*
using *that*
supply *take-from-list-ref* [*refine del*]
apply *refine-rcg*
apply *refine-dref-type*
by (*auto simp*: *F-split*)

definition *worklist-algo2* **where**
worklist-algo2 ≡ *if empty* a_0 *then RETURN False else worklist-algo2'*

lemma *worklist-algo2-ref*[*refine*]: *worklist-algo2* ≤ ⇓*Id worklist-algo''*
proof −
have *worklist-algo2* ≤ ⇓*Id worklist-algo1*
unfolding *worklist-algo2-def worklist-algo1-def*
by *refine-rcg standard*
also note *worklist-algo1-ref*
finally show *?thesis* **.**
qed
end — Worklist3

end — Theory
theory *Worklist-Subsumption-Impl1*
imports *Refine-Imperative-HOL.IICF Worklist-Subsumption1*
begin

4.5 Implementation on Lists

lemma *list-filter-foldli*:
[$x \leftarrow xs.\ P\ x$] = *rev* (*foldli xs* ($\lambda\ x.\ True$) ($\lambda\ x\ xs.$ *if* $P\ x$ *then* $x \#\ xs$ *else* *xs*) [])
(**is** - = *rev* (*foldli xs ?c ?f* []))
proof −
have ∗:
rev (*foldli xs ?c ?f* (*ys* @ *zs*)) = *rev zs* @ *rev* (*foldli xs ?c ?f ys*) **for** *xs ys zs*
proof (*induction xs arbitrary*: *ys*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons*[*of* $x \#\ ys$] *Cons*[*of ys*] **show** *?case* **by** *simp*
qed
show *?thesis*
apply (*induction xs*)
apply (*simp*; *fail*)
apply *simp*
apply (*subst* (*2*) ∗[*of* - [], *simplified*])
by *simp*
qed

context notes [*split*!] = *list.split* **begin**
sepref-decl-op *list-hdtl*: $\lambda\ (x \#\ xs) \Rightarrow (x,\ xs) :: [\lambda l.\ l \neq []]_f\ \langle A \rangle$*list-rel* $\rightarrow A \times_r \langle A \rangle$*list-rel*
by *auto*
end

context *Worklist4-Impl*
begin
sepref-register *PR-CONST* a_0 *PR-CONST* F' *PR-CONST* ($\trianglelefteq$) *PR-CONST succs PR-CONST empty*

lemma [*def-pat-rules*]:
$a_0 \equiv$ *UNPROTECT* a_0 $F' \equiv$ *UNPROTECT* F' ($\trianglelefteq$) $\equiv$ *UNPROTECT*

$(\trianglelefteq)$ *succs* $\equiv$ *UNPROTECT succs*
empty $\equiv$ *UNPROTECT empty*
by *simp-all*

lemma *take-from-list-alt-def*:
take-from-list xs = *do* {- $\leftarrow$ *ASSERT* (*xs* $\neq$ []); *RETURN* (*hd-tl xs*)}
unfolding *take-from-list-def* **by** (*auto simp*: *pw-eq-iff refine-pw-simps*)

lemma [*safe-constraint-rules*]: *CN-FALSE is-pure A* $\Longrightarrow$ *is-pure A* **by** *simp*

sepref-thm *filter-insert-wait-impl* **is**
uncurry (*RETURN oo PR-CONST filter-insert-wait*) :: (*list-assn A*)^d $*_a$ A^d $\rightarrow_a$ *list-assn A*
unfolding *filter-insert-wait-alt-def list-ex-foldli list-filter-foldli*
unfolding *HOL-list.fold-custom-empty*
unfolding *PR-CONST-def*
by *sepref*

concrete-definition (**in** $-$) *filter-insert-wait-impl*
uses *Worklist4-Impl.filter-insert-wait-impl.refine-raw* **is** (*uncurry ?f*, -) $\in$ -

lemmas [*sepref-fr-rules*] = *filter-insert-wait-impl.refine*[*OF Worklist4-Impl-axioms*]

sepref-register *filter-insert-wait*

lemmas [*sepref-fr-rules*] = *hd-tl-hnr*

sepref-thm *worklist-algo2-impl* **is** *uncurry0 worklist-algo2* :: *unit-assn*^{k} $\rightarrow_a$ *bool-assn*
unfolding *worklist-algo2-def worklist-algo2'-def add-succ2-alt-def PR-CONST-def*
supply [[*goals-limit* = *1*]]
supply *conv-to-is-Nil*[*simp*]
apply (*rewrite* **in** *Let* $\sharp$ - *lso-fold-custom-empty*)
unfolding *fold-lso-bex*
unfolding *take-from-list-alt-def*
apply (*rewrite* **in** [a_0] *HOL-list.fold-custom-empty*)
by *sepref*

concrete-definition (**in** $-$) *worklist-algo2-impl*
for *Lei* a_0*i Fi succsi emptyi*

uses *Worklist4-Impl.worklist-algo2-impl.refine-raw* **is** (*uncurry0 ?f*,-)∈-

end — Worklist4 Impl

context *Worklist4-Impl-finite-strict*
begin

lemma *worklist-algo2-impl-hnr-F-reachable*:
(*uncurry0* (*worklist-algo2-impl Lei* $a_0 i$ *Fi succsi emptyi*), *uncurry0* (*RETURN F-reachable*))
∈ *unit-assn*^{k} →$_a$ *bool-assn*
using *worklist-algo2-impl.refine*[*OF Worklist4-Impl-axioms*,
FCOMP worklist-algo2-ref[*THEN nres-relI*],
FCOMP worklist-algo''-correct[*THEN Id-SPEC-refine*, *THEN nres-relI*]]
by (*simp add*: *RETURN-def*)

sepref-decl-op *F-reachable* :: *bool-rel* .
lemma [*def-pat-rules*]: *F-reachable* ≡ *op-F-reachable* **by** *simp*

lemma *hnr-op-F-reachable*:
assumes *GEN-ALGO* $a_0 i$ (λ$a_0 i$. (*uncurry0* $a_0 i$, *uncurry0* (*RETURN* a_0)) ∈ *unit-assn*^{k} →$_a$ *A*)
assumes *GEN-ALGO Fi* (λ*Fi*. (*Fi*,*RETURN o F'*) ∈ A^k →$_a$ *bool-assn*)
assumes *GEN-ALGO Lei* (λ*Lei*. (*uncurry Lei*,*uncurry* (*RETURN oo* (⊴))) ∈ A^k ×$_a$ A^k →$_a$ *bool-assn*)
assumes *GEN-ALGO succsi* (λ*succsi*. (*succsi*,*RETURN o succs*) ∈ A^k →$_a$ *list-assn A*)
assumes *GEN-ALGO emptyi* (λ*Fi*. (*Fi*,*RETURN o empty*) ∈ A^k →$_a$ *bool-assn*)
shows
(*uncurry0* (*worklist-algo2-impl Lei* $a_0 i$ *Fi succsi emptyi*), *uncurry0* (*RETURN* (*PR-CONST op-F-reachable*)))
∈ *unit-assn*^{k} →$_a$ *bool-assn*
proof −
from *assms* **interpret** *Worklist4-Impl E* a_0 *F* (≼) *succs empty* (⊴) *F'* *A succsi* $a_0 i$ *Fi Lei emptyi*
by (*unfold-locales*; *simp add*: *GEN-ALGO-def*)

from *worklist-algo2-impl-hnr-F-reachable* **show** *?thesis* **by** *simp*
qed

sepref-decl-impl *hnr-op-F-reachable* .

end — Worklist4 (strictly finite)

end — End of Theory

5 Checking Always Properties

5.1 Abstract Implementation

theory *Liveness-Subsumption*
imports
Refine-Imperative-HOL.Sepref Worklist-Common Worklist-Algorithms-Subsumption-Graphs
begin

context *Search-Space-Nodes-Defs*
begin

sublocale *G*: *Subgraph-Node-Defs* **.**

no-notation *E* (‹- → -› [*100*, *100*] *40*)
notation *G.E'* (‹- → -› [*100*, *100*] *40*)
no-notation *reaches* (‹- →* -› [*100*, *100*] *40*)
notation *G.G'.reaches* (‹- →* -› [*100*, *100*] *40*)
no-notation *reaches1* (‹- →$^+$ -› [*100*, *100*] *40*)
notation *G.G'.reaches1* (‹- →$^+$ -› [*100*, *100*] *40*)

Plain set membership is also an option.

definition *check-loop v ST* = (∃ *v'* ∈ *set ST*. *v'* ≼ *v*)

definition *dfs* :: *'a set* ⇒ (*bool* × *'a set*) *nres* **where**
dfs P ≡ *do* {
(*P*,*ST*,*r*) ← *RECT* (λ*dfs* (*P*,*ST*,*v*).
do {
if check-loop v ST then RETURN (*P*, *ST*, *True*)
else do {
if ∃ *v'* ∈ *P*. *v* ≼ *v'* *then*
RETURN (*P*, *ST*, *False*)
else do {
let ST = *v* # *ST*;
(*P*, *ST'*, *r*) ←
FOREACHcd {*v'*. *v* → *v'*} (λ(-,-,*b*). ¬*b*) (λ*v'* (*P*,*ST*,-). *dfs* (*P*,*ST*,*v'*)) (*P*,*ST*,*False*);
ASSERT (*ST'* = *ST*);
let ST = *tl ST'*;
let P = *insert v P*;

```
            RETURN (P, ST, r)
          }
        }
      }
    ) (P,[],a0);
    RETURN (r, P)
  }

definition liveness-compatible where liveness-compatible P ≡
    (∀ x x' y. x → y ∧ x' ∈ P ∧ x ⪯ x' ⟶ (∃ y' ∈ P. y ⪯ y')) ∧
    (∀ s' ∈ P. ∀ s. s ⪯ s' ∧ V s ⟶
      ¬ (λ x y. x → y ∧ (∃ x' ∈ P. ∃ y' ∈ P. x ⪯ x' ∧ y ⪯ y'))^++ s s)


definition dfs-spec ≡
  SPEC (λ (r, P).
    (r ⟶ (∃ x. a0 →* x ∧ x →^+ x))
  ∧ (¬ r ⟶ ¬ (∃ x. a0 →* x ∧ x →^+ x)
      ∧ liveness-compatible P ∧ P ⊆ {x. V x}
    )
  )

end

locale Liveness-Search-Space-pre =
  Search-Space-Nodes +
  assumes finite-V: finite {a. V a}
begin

lemma check-loop-loop: ∃ v' ∈ set ST. v' ⪯ v if check-loop v ST
  using that unfolding check-loop-def by blast

lemma check-loop-no-loop: v ∉ set ST if ¬ check-loop v ST
  using that unfolding check-loop-def by blast

lemma mono:
  a ⪯ b ⟹ a → a' ⟹ V b ⟹ ∃ b'. b → b' ∧ a' ⪯ b'
  by (auto dest: mono simp: G.E'-def)

context
  fixes P :: 'a set and E1 E2 :: 'a ⇒ 'a ⇒ bool and v :: 'a
  defines [simp]: E1 ≡ λx y. x → y ∧ (∃x'∈P. x ⪯ x') ∧ (∃x∈P. y ⪯ x)
  defines [simp]: E2 ≡ λx y. x → y ∧ (x ⪯ v ∨ (∃xa∈P. x ⪯ xa)) ∧ (y ⪯
v ∨ (∃x∈P. y ⪯ x))
```

begin

interpretation G: *Graph-Defs E1* **.**
interpretation G': *Graph-Defs E2* **.**
interpretation SG: *Subgraph E2 E1* **by** *standard auto*
interpretation SG': *Subgraph-Start E* a_0 *E1* **by** *standard auto*
interpretation SG'': *Subgraph-Start E* a_0 *E2* **by** *standard auto*

lemma *G-subgraph-reaches*[*intro*]:
G.G'.reaches a b **if** *G.reaches a b*
using *that* **by** *induction auto*

lemma *G'-subgraph-reaches*[*intro*]:
G.G'.reaches a b **if** *G'.reaches a b*
using *that* **by** *induction auto*

lemma *liveness-compatible-extend*:
assumes
$V\ s\ V\ v\ s \preceq v$
liveness-compatible P
$\forall\, va.\ v \to va \longrightarrow (\exists\, x{\in}P.\ va \preceq x)$
$E2^{++}\ s\ s$
shows *False*
proof −
include *graph-automation-aggressive*
have *1*: $\exists\ b' \in P.\ b \preceq b'$ **if** *G.reaches a b* $a \preceq a'\ a' \in P$ **for** $a\ a'\ b$
using *that* **by** *cases auto*
have *2*: *E1 a b* **if** $a \to b\ a \preceq a'\ a' \in P\ V\ a$ **for** $a\ a'\ b$
using *assms*(*4*) *that* **unfolding** *liveness-compatible-def* **by** *auto*
from *assms*(*6*) **have** $E2^{++}\ s\ s$
by *auto*
have *4*: *G.reaches a b* **if** *G'.reaches a b* $a \preceq a'\ a' \in P\ V\ a$ **for** $a\ a'\ b$
using *that*
proof *induction*
case *base*
then show *?case*
by *blast*
next
case (*step b c*)
then have *G.reaches a b*
by *auto*
from ‹$V\ a$› ‹*G.reaches a b*› **have** $V\ b$
including *subgraph-automation* **by** *blast*
from ‹*G.reaches a b*› ‹$a \preceq a'$› ‹$a' \in P$› **obtain** b' **where** $b' \in P\ b \preceq b'$

```
    by (fastforce dest: 1)
  with ‹E2 b c› ‹V b› have E1 b c
    by (fastforce intro: 2)
  with ‹G.reaches a b› show ?case
    by blast
 qed
 from ‹E2^++ s s› obtain x where E2 s x G'.reaches x s
  by blast
 then have s → x
  by auto
 with ‹s ⪯ v› ‹V s› ‹V v› obtain x' where v → x' x ⪯ x'
  by (auto dest: mono)
 with assms(5) obtain x'' where x ⪯ x'' x'' ∈ P
  by (auto intro: order-trans)
 from 4[OF ‹G'.reaches x s› ‹x ⪯ x''› ‹x'' ∈ P›] ‹s → x› ‹V s› have
G.reaches x s
  including subgraph-automation by blast
 with ‹x ⪯ x''› ‹x'' ∈ P› obtain s' where s ⪯ s' s' ∈ P
  by (blast dest: 1)
 with ‹s → x› ‹x ⪯ x''› ‹x'' ∈ P› have E1 s x
  by auto
 with ‹G.reaches x s› have G.reaches1 s s
  by blast
 with assms(4) ‹s' ∈ P› ‹s ⪯ s'› ‹V s› show False
  unfolding liveness-compatible-def by auto
qed

lemma liveness-compatible-extend':
 assumes
  V s V v s ⪯ s' s' ∈ P
  ∀ va. v → va ⟶ (∃ x∈P. va ⪯ x)
  liveness-compatible P
  E2^++ s s
 shows False
proof −
 show ?thesis
 proof (cases G.reaches1 s s)
  case True
  with assms show ?thesis
   unfolding liveness-compatible-def by auto
 next
  case False
  with SG.non-subgraph-cycle-decomp[of s s] assms(7) obtain c d where
**:
```

```
    G'.reaches s c E2 c d ¬ E1 c d G'.reaches d s
    by auto
  from ‹E2 c d› ‹¬ E1 c d› have c ⪯ v ∨ d ⪯ v
    by auto
  with ‹V s› ∗∗ obtain v' where G'.reaches1 v' v' v' ⪯ v V v'
    apply atomize-elim
    apply (erule disjE)
    subgoal
      including graph-automation-aggressive
      by (blast intro: rtranclp-trans G.subgraphI)
    subgoal
      unfolding G'.reaches1-reaches-iff2
      by (blast intro: rtranclp-trans intro: G.subgraphI)
    done
  with assms show ?thesis
    by (auto intro: liveness-compatible-extend)
 qed
qed

end

lemma liveness-compatible-cycle-start:
 assumes
  liveness-compatible P a →∗ x x →+ x a ⪯ s s ∈ P
 shows False
proof −
 include graph-automation-aggressive
 have ∗: ∃ y ∈ P. x ⪯ y if G.G'.reaches a x for x
  using that assms unfolding liveness-compatible-def by induction auto
 have ∗∗:
  (λx y. x → y ∧ (∃ x'∈P. x ⪯ x') ∧ (∃ x∈P. y ⪯ x))++ a' b ⟷ a' →+
b
  if a →∗ a' for a' b
  apply standard
  subgoal
    by (induction rule: tranclp-induct; blast)
  subgoal
    using ‹a →∗ a'›
    apply − apply (induction rule: tranclp.induct)
    subgoal for a' b'
      by (auto intro: ∗)
    by (rule tranclp.intros(2), auto intro: ∗)
  done
 from assms show ?thesis
```

unfolding *liveness-compatible-def* **by** *clarsimp* (*blast dest*: ∗ ∗∗ *intro*: *G.subgraphI*)
qed

lemma *liveness-compatible-inv*:
assumes *V v liveness-compatible P* ∀ *va. v* → *va* ⟶ (∃ *x*∈*P. va* ⪯ *x*)
shows *liveness-compatible* (*insert v P*)
using *assms*
apply (*subst liveness-compatible-def*)
apply *safe*
apply *clarsimp-all*
apply (*meson mono order-trans*; *fail*)
apply (*subst* (*asm*) *liveness-compatible-def*, *meson*; *fail*)
by (*blast intro*: *liveness-compatible-extend liveness-compatible-extend′*)+

interpretation *subsumption*: *Subsumption-Graph-Pre-Nodes* (⪯) (≺) *E V*
by *standard* (*drule mono*, *auto simp*: *Subgraph-Node-Defs.E′-def*)

lemma *pre-cycle-cycle*:
(∃ *x x′. a*$_0$ →∗ *x* ∧ *x* →$^+$ *x′* ∧ *x* ⪯ *x′*) ⟷ (∃ *x. a*$_0$ →∗ *x* ∧ *x* →$^+$ *x*)
by (*meson G.E′-def G.G′.reaches1-reaches-iff1 subsumption.pre-cycle-cycle-reachable finite-V*)

lemma *reachable-alt*:
V v **if** *V a*$_0$ *a*$_0$ →∗ *v*
using *that*(*2*,*1*) **by** *cases* (*auto simp*: *G.E′-def*)

lemma *dfs-correct*:
dfs P ≤ *dfs-spec* **if** *V a*$_0$ *liveness-compatible P P* ⊆ {*x. V x*}
proof −

define *rpre* **where** *rpre* ≡ λ(*P*,*ST*,*v*).
a$_0$ →∗ *v*
∧ *V v*
∧ *P* ⊆ {*x. V x*}
∧ *set ST* ⊆ {*x. a*$_0$ →∗ *x*}
∧ *set ST* ⊆ {*x. V x*}
∧ *liveness-compatible P*
∧ (∀ *s* ∈ *set ST. s* →$^+$ *v*)
∧ *distinct ST*

define *rpost* **where** *rpost* ≡ λ(*P*,*ST*,*v*) (*P′*,*ST′*,*r*).
(*r* ⟶ (∃ *x x′. a*$_0$ →∗ *x* ∧ *x* →$^+$ *x′* ∧ *x* ⪯ *x′*) ∧ *ST′* = *ST*) ∧

```
  (¬ r ⟶
    P ⊆ P'
    ∧ P' ⊆ {x. V x}
    ∧ set ST ⊆ {x. a₀ →* x}
    ∧ ST' = ST
    ∧ (∀ s ∈ set ST. s →* v)
    ∧ liveness-compatible P'
    ∧ (∃ v' ∈ P'. v ≼ v')
    ∧ distinct ST
    )


  define inv where inv ≡ λ P ST v (- :: 'a set) it (P', ST', r).
    (r ⟶ (∃ x x'. a₀ →* x ∧ x →⁺ x' ∧ x ≼ x') ∧ ST' = ST) ∧
    (¬ r ⟶
       P ⊆ P'
    ∧ P' ⊆ {x. V x}
    ∧ set ST ⊆ {x. a₀ →* x}
    ∧ ST' = ST
    ∧ (∀ s ∈ set ST. s →* v)
    ∧ set ST ⊆ {x. V x}
    ∧ liveness-compatible P'
    ∧ (∀ v ∈ {v'. v → v'} − it. (∃ v' ∈ P'. v ≼ v'))
    ∧ distinct ST
    )


  define Termination :: (('a set × 'a list × 'a) × 'a set × 'a list × 'a) set
where
    Termination = inv-image (finite-psupset {x. V x}) (λ (a,b,c). set b)

  have rpre-init: rpre (P, [], a₀)
    unfolding rpre-def using ‹V a₀› ‹liveness-compatible P› ‹P ⊆ -› by
auto

  have wf: wf Termination
    unfolding Termination-def by (blast intro: finite-V)

  have successors-finite: finite {v'. v → v'} for v
    using finite-V unfolding G.E'-def by auto

  show ?thesis
    unfolding dfs-def dfs-spec-def
    apply (refine-rcg refine-vcg)
```

apply (*rule Orderings.order.trans*)
apply(*rule RECT-rule*[**where**
pre = *rpre* **and**
V = *Termination* **and**
M = λ*x*. *SPEC* (*rpost x*)
])

subgoal
unfolding *FOREACHcd-def* **by** *refine-mono*

apply (*rule wf*; *fail*)

apply (*rule rpre-init*; *fail*)

defer

subgoal
apply *refine-vcg*
unfolding *rpost-def pre-cycle-cycle*
including *graph-automation*
by (*auto dest*: *liveness-compatible-cycle-start*)

apply (*thin-tac* - = *f*)

apply *refine-vcg*

subgoal for *f x a b aa ba*
by (*subst rpost-def*, *subst* (*asm*) (*2*) *rpre-def*, *auto 4 4 dest*: *check-loop-loop*)

subgoal for *f x P b ST v*
by (*subst rpost-def*, *subst* (*asm*) (*2*) *rpre-def*, *force*)

subgoal for *f x P b ST v*

apply (*refine-vcg*
FOREACHcd-rule[**where** $I = inv\ P\ (v\ \#\ ST)\ v$]
)
apply *clarsimp-all*

subgoal
by (*auto intro*: *successors-finite*)

subgoal
using ‹$V\ a_0$›
by (
subst (*asm*) (*2*) *rpre-def*, *subst inv-def*,
auto dest: *check-loop-no-loop*
)

subgoal for - *it* $v'\ P'\ ST'\ c$
apply (*rule Orderings.order.trans*)
apply *rprems*

subgoal
apply (*subst rpre-def*, *subst* (*asm*) *inv-def*)
apply *clarsimp*
subgoal premises *prems*
proof −
from *prems* ‹$V\ a_0$› **have** $v \to v'$
by *auto*
with *prems* **show** *?thesis*
by (*auto intro*: *G.E'-V2*)
qed
done

subgoal
using ‹$V\ a_0$› **unfolding** *Termination-def*
by (*auto simp*: *finite-psupset-def inv-def intro*: *G.subgraphI*)

subgoal
apply (*subst inv-def*, *subst* (*asm*) *inv-def*, *subst rpost-def*)

apply (*clarsimp simp*: *it-step-insert-iff*)
by (*safe*; (*intro exI conjI*)*?*; (*blast intro*: *rtranclp-trans*))

done

subgoal
by (*subst* (*asm*) *inv-def*, *simp*)

subgoal for P' ST' c
using ‹V a_0› **by** (*subst rpost-def*, *subst* (*asm*) *inv-def*, *auto*)

subgoal
by (*subst* (*asm*) *inv-def*, *simp*)

subgoal for P' ST'
using ‹V a_0›
by (*subst rpost-def*, *subst* (*asm*) *inv-def*, *auto intro*: *liveness-compatible-inv G.subgraphI*)

done
done
qed

end

locale *Liveness-Search-Space-Defs* =
Search-Space-Nodes-Defs +
fixes *succs* :: $'a \Rightarrow 'a$ *list*
begin

definition *dfs1* :: $'a$ *set* $\Rightarrow$ (*bool* $\times$ $'a$ *set*) *nres* **where**
dfs1 $P \equiv$ *do* {
(P,ST,r) $\leftarrow$ *RECT* (λ*dfs* (P,ST,v).
do {
ASSERT (V v $\wedge$ *set* $ST \subseteq \{x.\ V\ x\}$);
if check-loop v ST *then RETURN* (P, ST, *True*)
else do {
if $\exists\ v' \in P.\ v \preceq v'$ *then*
RETURN (P, ST, *False*)
else do {

```
            let ST = v # ST;
            (P, ST', r) ←
              nfoldli (succs v) (λ(-,-,b). ¬b) (λv' (P,ST,-). dfs (P,ST,v'))
(P,ST,False);
            ASSERT (ST' = ST);
            let ST = tl ST';
            let P = insert v P;
            RETURN (P, ST, r)
          }
        }
      }
    ) (P,[],a0);
    RETURN (r, P)
  }
```

end

locale *Liveness-Search-Space* =
  Liveness-Search-Space-Defs +
  Liveness-Search-Space-pre +
  assumes *succs-correct*: $V\ a \Longrightarrow set\ (succs\ a) = \{x.\ a \to x\}$
  assumes *finite-V*: *finite* $\{a.\ V\ a\}$
begin

— The following complications only arise because we add the assertion in this refinement step.
lemma *succs-ref*[*refine*]:
  $(succs\ a,\ succs\ b) \in \langle Id\rangle list\text{-}rel$ **if** $(a,\ b) \in Id$
  using *that* **by** *auto*

lemma *start-ref*[*refine*]:
  $((P,\ [],\ a_0),\ P,\ [],\ a_0) \in Id \times_r br\ id\ (\lambda\ xs.\ set\ xs \subseteq \{x.\ V\ x\}) \times_r br\ id\ V$
if $V\ a_0$
  using *that* **by** (*auto simp*: *br-def*)

lemma *refine-aux*[*refine*]:
  $((x,\ x1c,\ True),\ x',\ x1a,\ True) \in Id \times_r br\ id\ (\lambda xs.\ set\ xs \subseteq Collect\ V) \times_r Id$
  if $(x1c,\ x1a) \in br\ id\ (\lambda xs.\ set\ xs \subseteq \{x.\ V\ x\})\ (x,\ x') \in Id$
  using *that* **by** *auto*

lemma *refine-loop*:
  $(\bigwedge x\ x'.\ (x,\ x') \in Id \times_r br\ id\ (\lambda xs.\ set\ xs \subseteq \{x.\ V\ x\}) \times_r br\ id\ V \Longrightarrow$
    $dfs'\ x \leq \Downarrow (Id \times_r br\ id\ (\lambda xs.\ set\ xs \subseteq Collect\ V) \times_r bool\text{-}rel)$

(*dfsa* x')) $\Longrightarrow$
$(x, x') \in Id \times_r br\ id\ (\lambda xs.\ set\ xs \subseteq \{x.\ V\ x\}) \times_r br\ id\ V \Longrightarrow$
$x2 = (x1a, x2a) \Longrightarrow$
$x' = (x1, x2) \Longrightarrow$
$x2b = (x1c, x2c) \Longrightarrow$
$x = (x1b, x2b) \Longrightarrow$
nfoldli (*succs x2c*) (λ(-, -, *b*). $\neg$ *b*) ($\lambda v'$ (*P*, *ST*, -). *dfs'* (*P*, *ST*, v')) (*x1b*, *x2c* # *x1c*, *False*)
$\leq \Downarrow$ (*Id* $\times_r$ *br id* ($\lambda xs.$ *set xs* $\subseteq \{x.\ V\ x\}$) $\times_r$ *bool-rel*)
(*FOREACHcd* $\{v'.\ x2a \rightarrow v'\}$ (λ(-, -, *b*). $\neg$ *b*)
($\lambda v'$ (*P*, *ST*, -). *dfsa* (*P*, *ST*, v')) (*x1*, *x2a* # *x1a*, *False*))
apply (*subgoal-tac* (*succs x2c*, *succs x2a*) $\in \langle br\ id\ V \rangle$*list-rel*)

unfolding *FOREACHcd-def*
apply *refine-rcg*
apply (*rule rhs-step-bind-SPEC*)
apply (*rule succs-correct*)
apply (*solves* ‹*auto simp*: *br-def*›)
apply (*erule nfoldli-refine*)
apply (*solves* ‹*auto simp*: *br-def*›)+
unfolding *br-def list-rel-def*
by (*simp*, *rule list.rel-refl-strong*, *auto intro*: *G.E'-V2 simp*: *succs-correct*)

lemma *dfs1-dfs-ref*[*refine*]:
dfs1 P $\leq \Downarrow$ *Id* (*dfs P*) **if** *V* a_0
using *that* **unfolding** *dfs1-def dfs-def*
apply *refine-rcg*
apply (*solves* ‹*auto simp*: *br-def*›)+
apply (*rule refine-loop*; *assumption*)
by (*auto simp*: *br-def*)

end

end

5.2 Implementation on Maps

theory *Liveness-Subsumption-Map*
imports *Liveness-Subsumption Worklist-Common*
begin

locale *Liveness-Search-Space-Key-Defs* =
Liveness-Search-Space-Defs E **for** *E* :: $'v \Rightarrow 'v \Rightarrow$ *bool* +
fixes *key* :: $'v \Rightarrow 'k$

fixes *subsumes'* :: $'v \Rightarrow 'v \Rightarrow bool$ (**infix** $\langle \trianglelefteq \rangle$ *50*)
begin

sublocale *Search-Space-Key-Defs* **where** *empty* = *undefined* **and** *subsumes'* = *subsumes'* **.**

definition *check-loop-list v ST* = $(\exists\ v' \in set\ ST.\ v' \preceq v)$

definition *insert-map-set v S* ≡
let k = *key v*; *S'* = (*case S k of Some S* ⇒ *S* | *None* ⇒ {}) *in S*($k \mapsto$ (*insert v S'*))

definition *push-map-list v S* ≡
let k = *key v*; *S'* = (*case S k of Some S* ⇒ *S* | *None* ⇒ []) *in S*($k \mapsto v$ # *S'*)

definition *check-subsumption-map-set v S* ≡
let k = *key v*; *S'* = (*case S k of Some S* ⇒ *S* | *None* ⇒ {}) *in* ($\exists\ x \in S'.\ v \trianglelefteq x$)

definition *check-subsumption-map-list v S* ≡
let k = *key v*; *S'* = (*case S k of Some S* ⇒ *S* | *None* ⇒ []) *in* ($\exists\ x \in set\ S'.\ x \trianglelefteq v$)

definition *pop-map-list v S* ≡
let k = *key v*; *S'* = (*case S k of Some S* ⇒ *tl S* | *None* ⇒ []) *in S*($k \mapsto S'$)

definition *dfs-map* :: $('k \rightharpoonup 'v\ set) \Rightarrow (bool \times ('k \rightharpoonup 'v\ set))\ nres$ **where**
dfs-map P ≡ *do* {
(*P*,*ST*,*r*) ← *RECT* (λ*dfs* (*P*,*ST*,*v*).
if check-subsumption-map-list v ST then RETURN (*P*, *ST*, *True*)
else do {
if check-subsumption-map-set v P then
RETURN (*P*, *ST*, *False*)
else do {
let ST = *push-map-list v ST*;
(*P*, *ST*, *r*) ←
nfoldli (*succs v*) (λ(-,-,*b*). ¬*b*) (λ*v'* (*P*,*ST*,-). *dfs* (*P*,*ST*,*v'*))
(*P*,*ST*,*False*);

```
        let ST = pop-map-list v ST;
        let P = insert-map-set v P;
        RETURN (P, ST, r)
      }
    }
  ) (P,(Map.empty::('k ⇀ 'v list)),a₀);
  RETURN (r, P)
}
```

end

locale *Liveness-Search-Space-Key* =
Liveness-Search-Space + Liveness-Search-Space-Key-Defs +
assumes *subsumes-key*[*intro*, *simp*]: $a \trianglelefteq b \Longrightarrow key\ a = key\ b$
assumes *V-subsumes'*: $V\ a \Longrightarrow a \preceq b \longleftrightarrow a \trianglelefteq b$
begin

definition
irrefl-trans-on $R\ S \equiv (\forall\ x \in S. \neg\ R\ x\ x) \wedge (\forall\ x \in S. \forall\ y \in S. \forall\ z \in S.$
$R\ x\ y \wedge R\ y\ z \longrightarrow R\ x\ z)$

definition
map-list-rel =
$\{(m, xs). \bigcup (set\ `\ ran\ m) = set\ xs \wedge (\forall\ k. \forall\ x. m\ k = Some\ x \longrightarrow (\forall$
$v \in set\ x. key\ v = k))$
$\wedge (\exists\ R. irrefl\text{-}trans\text{-}on\ R\ (set\ xs)$
$\wedge (\forall\ k. \forall\ x. m\ k = Some\ x \longrightarrow sorted\text{-}wrt\ R\ x) \wedge sorted\text{-}wrt\ R$
$xs)$
$\wedge distinct\ xs$
$\}$

definition *list-set-hd-rel* $x \equiv \{(l, s). set\ l = s \wedge distinct\ l \wedge l \neq [] \wedge hd\ l$
$= x\}$

lemma *empty-map-list-rel*:
$(Map.empty, []) \in map\text{-}list\text{-}rel$
unfolding *map-list-rel-def irrefl-trans-on-def* **by** *auto*

lemma *rel-start*[*refine*]:
$((P, Map.empty, a_0), P', [], a_0) \in map\text{-}set\text{-}rel \times_r map\text{-}list\text{-}rel \times_r Id$ **if**
$(P, P') \in map\text{-}set\text{-}rel$
using *that* **unfolding** *map-set-rel-def* **by** (*auto intro*: *empty-map-list-rel*)

lemma *refine-True*:

(*x1b*, *x1*) ∈ *map-set-rel* ⟹ (*x1c*, *x1a*) ∈ *map-list-rel*
⟹ ((*x1b*, *x1c*, *True*), *x1*, *x1a*, *True*) ∈ *map-set-rel* ×$_r$ *map-list-rel* ×$_r$ *Id*
by *simp*

lemma *check-subsumption-ref*[*refine*]:
V x2a ⟹ (*x1b*, *x1*) ∈ *map-set-rel* ⟹ *check-subsumption-map-set x2a x1b* = (∃ *x*∈*x1*. *x2a* ≼ *x*)
unfolding *map-set-rel-def list-set-rel-def check-subsumption-map-set-def*
unfolding *ran-def* **by** (*auto split*: *option.splits simp*: *V-subsumes'*)

lemma *check-subsumption'-ref*[*refine*]:
set xs ⊆ {*x*. *V x*} ⟹ (*m*, *xs*) ∈ *map-list-rel*
⟹ *check-subsumption-map-list x m* = *check-loop x xs*
unfolding *map-list-rel-def list-set-rel-def check-subsumption-map-list-def check-loop-def*
unfolding *ran-def* **apply** (*clarsimp split*: *option.splits*, *safe*)
subgoal for *R x' xs'*
by (*drule sym*, *drule sym*, *subst* (*asm*) *V-subsumes'*[*symmetric*], *auto*)
by (*subst* (*asm*) *V-subsumes'*; *force*)

lemma *not-check-loop-non-elem*:
x ∉ *set xs* **if** ¬ *check-loop-list x xs*
using *that* **unfolding** *check-loop-list-def* **by** *auto*

lemma *insert-ref*[*refine*]:
(*x1b*, *x1*) ∈ *map-set-rel* ⟹
(*x1c*, *x1a*) ∈ ⟨*Id*⟩*list-set-rel* ⟹
¬ *check-loop-list x2a x1c* ⟹
((*x1b*, *x2a* # *x1c*, *False*), *x1*, *insert x2a x1a*, *False*) ∈ *map-set-rel* ×$_r$ *list-set-hd-rel x2a* ×$_r$ *Id*
unfolding *list-set-hd-rel-def list-set-rel-def*
by (*auto dest*: *not-check-loop-non-elem simp*: *br-def*)

lemma *insert-map-set-ref*:
(*m*, *S*) ∈ *map-set-rel* ⟹ (*insert-map-set x m*, *insert x S*) ∈ *map-set-rel*
unfolding *insert-map-set-def insert-def map-set-rel-def*
apply (*clarsimp split*: *option.splits*, *safe*)
subgoal for *x' S'*
unfolding *ran-def Let-def* **by** (*auto split*: *option.splits split*: *if-split-asm*)
subgoal
unfolding *ran-def Let-def* **by** (*auto split*: *if-split-asm*)
subgoal
unfolding *ran-def Let-def*
apply (*clarsimp split*: *option.splits*, *safe*)

apply (*metis option.simps*(*3*))
by (*metis insertCI option.inject*)
subgoal
unfolding *ran-def Let-def* **by** (*auto split*: *option.splits if-split-asm*)
by (*auto simp*: *Let-def split*: *if-split-asm option.split*)

lemma *map-list-rel-memD*:
assumes (*m*, *xs*) ∈ *map-list-rel x* ∈ *set xs*
obtains *xs'* **where** *x* ∈ *set xs' m* (*key x*) = *Some xs'*
using *assms* **unfolding** *map-list-rel-def ran-def* **by** (*auto 4 3 dest*: *sym*)

lemma *map-list-rel-memI*:
(*m*, *xs*) ∈ *map-list-rel* ⟹ *m k* = *Some xs'* ⟹ *x'* ∈ *set xs'* ⟹ *x'* ∈ *set xs*
unfolding *map-list-rel-def ran-def* **by** *auto*

lemma *map-list-rel-grouped-by-key*:
x' ∈ *set xs'* ⟹ (*m*, *xs*) ∈ *map-list-rel* ⟹ *m k* = *Some xs'* ⟹ *key x'* = *k*
unfolding *map-list-rel-def* **by** *auto*

lemma *map-list-rel-not-memI*:
k ≠ *key x* ⟹ *m k* = *Some xs'* ⟹ (*m*, *xs*) ∈ *map-list-rel* ⟹ *x* ∉ *set xs'*
unfolding *map-list-rel-def* **by** *auto*

lemma *map-list-rel-not-memI2*:
x ∉ *set xs'* **if** *m a* = *Some xs'* (*m*, *xs*) ∈ *map-list-rel x* ∉ *set xs*
using *that* **unfolding** *map-list-rel-def ran-def* **by** *auto*

lemma *push-map-list-ref*:
x ∉ *set xs* ⟹ (*m*, *xs*) ∈ *map-list-rel* ⟹ (*push-map-list x m*, *x* # *xs*) ∈ *map-list-rel*
unfolding *push-map-list-def*
apply (*subst map-list-rel-def*)
apply (*clarsimp*, *safe*)
subgoal for *x' xs'*
unfolding *ran-def Let-def*
by (*auto split*: *option.split-asm if-split-asm intro*: *map-list-rel-memI*)
subgoal
unfolding *ran-def Let-def* **by** (*auto 4 3 split*: *option.splits if-splits*)
subgoal for *x'*
unfolding *ran-def Let-def*
apply (*erule map-list-rel-memD*, *assumption*)
apply (*cases key x* = *key x'*)

```
    subgoal for xs'
      by auto
    subgoal for xs'
      apply clarsimp
      apply (rule exI[where x = xs'])
       apply safe
      apply (inst-existentials key x')
       apply (solves auto)
      apply force
      done
    done
  subgoal for k xs' x'
    unfolding Let-def
   by (auto split: option.split-asm if-split-asm dest: map-list-rel-grouped-by-key)
  subgoal
  proof −
    assume A: x ∉ set xs (m, xs) ∈ map-list-rel
    then obtain R where ∗:
      x ∉ set xs
      (⋃ x ∈ ran m. set x) = set xs
      ∀ k x. m k = Some x ⟶ (∀ v∈set x. key v = k)
      distinct xs
      ∀ k x. m k = Some x ⟶ sorted-wrt R x
      sorted-wrt R xs
      irrefl-trans-on R (set xs)
      unfolding map-list-rel-def by auto
    have ∗∗: sorted-wrt (λa b. a = x ∧ b ≠ x ∨ a ≠ x ∧ b ≠ x ∧ R a b) xs
if
      sorted-wrt R xs x ∉ set xs for xs
      using that by (induction xs) auto

    show ?thesis
     apply (inst-existentials λ a b. a = x ∧ b ≠ x ∨ a ≠ x ∧ b ≠ x ∧ R a
b)
       apply safe
      unfolding Let-def
      subgoal
        using ‹irrefl-trans-on - -› unfolding irrefl-trans-on-def by blast
      apply (clarsimp split: option.split-asm if-split-asm)
      subgoal
        apply (drule map-list-rel-not-memI, assumption, rule A)
        using ∗(5) by (auto intro: ∗∗)
    using A(1) ∗(5) by (auto 4 3 intro: ∗ ∗∗ A dest: map-list-rel-not-memI2)
  qed
```

by (*auto simp*: *map-list-rel-def*)

lemma *insert-map-set-ref'*[*refine*]:
(*x1b*, *x1*) ∈ *map-set-rel* ⟹
(*x1c*, *x1a*) ∈ *map-set-rel* ⟹
¬ *check-subsumption'* *x2a x1c* ⟹
((*x1b*, *insert-map-set x2a x1c*, *False*), *x1*, *insert x2a x1a*, *False*) ∈ *map-set-rel* ×$_r$ *map-set-rel* ×$_r$ *Id*
by (*auto intro*: *insert-map-set-ref*)

lemma *map-list-rel-check-subsumption-map-list*:
set xs ⊆ {*x*. *V x*} ⟹ (*m*, *xs*) ∈ *map-list-rel* ⟹ ¬ *check-subsumption-map-list x m* ⟹ *x* ∉ *set xs*
unfolding *check-subsumption-map-list-def* **by** (*auto 4 3 elim*!: *map-list-rel-memD dest*: *V-subsumes'*)

lemma *push-map-list-ref'*[*refine*]:
set x1a ⊆ {*x*. *V x*} ⟹
(*x1b*, *x1*) ∈ *map-set-rel* ⟹
(*x1c*, *x1a*) ∈ *map-list-rel* ⟹
¬ *check-subsumption-map-list x2a x1c* ⟹
((*x1b*, *push-map-list x2a x1c*, *False*), *x1*, *x2a* # *x1a*, *False*) ∈ *map-set-rel* ×$_r$ *map-list-rel* ×$_r$ *Id*
by (*auto intro*: *push-map-list-ref dest*: *map-list-rel-check-subsumption-map-list*)

lemma *sorted-wrt-tl*:
sorted-wrt R (*tl xs*) **if** *sorted-wrt R xs*
using *that* **by** (*induction xs*) *auto*

lemma *irrefl-trans-on-mono*:
irrefl-trans-on R S **if** *irrefl-trans-on R S'* *S* ⊆ *S'*
using *that* **unfolding** *irrefl-trans-on-def* **by** *blast*

lemma *pop-map-list-ref*[*refine*]:
(*pop-map-list v m*, *S*) ∈ *map-list-rel* **if** (*m*, *v* # *S*) ∈ *map-list-rel*
using *that* **unfolding** *map-set-rel-def pop-map-list-def*
apply (*clarsimp split*: *option.splits if-split-asm simp*: *Let-def*, *safe*)
subgoal premises *prems*
using *prems* **unfolding** *map-list-rel-def*
by *clarsimp* (*rule map-list-rel-memD*[*OF prems*(*1*), *of v*], *simp*, *metis option.simps*(*3*))
subgoal premises *prems0* **for** *xs*
using *prems0* **unfolding** *map-list-rel-def*
apply *clarsimp*

```
    apply safe
    subgoal premises prems for R x xs'
    proof −
      have ∗: x ∈ set S if x ∈ set (tl xs) m (key v) = Some xs
      proof −
        from prems have sorted-wrt R xs
          by auto
        from ‹m (key v) = -› have v ∈ set xs
          using map-list-rel-memD[OF ‹(m, v # S) ∈ map-list-rel›, of v] by
auto
        have ∗: R v x if x ∈ set xs v ≠ x for x
        using that prems by − (drule map-list-rel-memI[OF ‹- ∈ map-list-rel›],
auto)
        have v ≠ x
        proof (rule ccontr)
          assume ¬ v ≠ x
          with that obtain a as bs where
            xs = a # as @ v # bs
            unfolding in-set-conv-decomp by (cases xs) auto
          with ‹sorted-wrt R xs› have a ∈ set xs R a v a ∈ insert v (set S)
            using map-list-rel-memI[OF ‹- ∈ map-list-rel› ‹m - = -›, of a]
            by auto
          with ∗ ‹irrefl-trans-on - -› show False
            unfolding irrefl-trans-on-def by auto
        qed
        with that show ?thesis
        by (auto elim: in-set-tlD dest: map-list-rel-memI[OF ‹- ∈ map-list-rel›])
      qed
      moreover have x ∈ set S if m a = Some xs' a ≠ key v for a :: 'b
        using prems0 prems that by (metis map-list-rel-memI set-ConsD)
      then show ?thesis
        using prems unfolding ran-def by (auto split: if-split-asm intro: ∗)
    qed
    subgoal premises prems for R x
    proof −
      from map-list-rel-memD[OF ‹- ∈ map-list-rel›, of x] ‹x ∈ -› obtain
xs' where xs':
        x ∈ set xs' m (key x) = Some xs'
        by auto
      show ?thesis
      proof (cases key x = key v)
        case True
        with prems xs' have [simp]: xs' = xs
          by auto
```

```
      from prems have x ≠ v
        by auto
      have x ∈ set (tl xs)
      proof (rule ccontr)
        assume x ∉ set (tl xs)
        with xs' have hd xs = x
          by (cases xs) auto
        from prems have sorted-wrt R xs
          by auto
        from ‹m (key v) = -› have v ∈ set xs
          using map-list-rel-memD[OF ‹(m, v # S) ∈ map-list-rel›, of v]
by auto
        have ∗: R v x if x ∈ set xs v ≠ x for x
            using that prems by − (drule map-list-rel-memI[OF ‹- ∈
map-list-rel›], auto)
        with ‹x ∈ set xs'› ‹x ≠ v› have R v x
          by auto
        from ‹v ∈ set xs› ‹hd xs = x› ‹x ≠ v› have R x v
          unfolding in-set-conv-decomp
          apply clarsimp
          using ‹sorted-wrt R xs›
          subgoal for ys
            by (cases ys) auto
          done
        with ‹R v x› ‹irrefl-trans-on - -› ‹x ∈ set S› show False
          unfolding irrefl-trans-on-def by blast
      qed
      with xs' show ?thesis
        unfolding ‹xs' = -› ran-def by auto
    next
      case False
      with xs' show ?thesis
        unfolding ran-def by auto
    qed
  qed
  subgoal
    by (meson in-set-tlD)
  subgoal for R
    by (blast intro: irrefl-trans-on-mono sorted-wrt-tl)
  done
done

lemma tl-list-set-ref:
  (m, S) ∈ map-set-rel ⟹
```

```
  (st, ST) ∈ list-set-hd-rel x ⟹
  (tl st, ST − {x}) ∈ ⟨Id⟩list-set-rel
 unfolding list-set-hd-rel-def list-set-rel-def
 by (auto simp: br-def distinct-hd-tl dest: in-set-tlD in-hd-or-tl-conv)

lemma succs-id-ref:
 (succs x, succs x) ∈ ⟨Id⟩ list-rel
 by simp

lemma dfs-map-dfs-refine':
 dfs-map P ≤ ⇓ (Id ×r map-set-rel) (dfs1 P') if (P, P') ∈ map-set-rel
 using that
 unfolding dfs-map-def dfs1-def
 apply refine-rcg
  using [[goals-limit=1]]
          apply (clarsimp, rule check-subsumption'-ref; assumption)
         apply (clarsimp, rule refine-True; assumption)
        apply (clarsimp, rule check-subsumption-ref; assumption)
        apply (simp; fail)
       apply (clarsimp; rule succs-id-ref; fail)
      apply (clarsimp, rule push-map-list-ref'; assumption)
     by (auto intro: insert-map-set-ref pop-map-list-ref)

lemma dfs-map-dfs-refine:
 dfs-map P ≤ ⇓ (Id ×r map-set-rel) (dfs P') if (P, P') ∈ map-set-rel V a0
proof −
 note dfs-map-dfs-refine'[OF ‹- ∈ map-set-rel›]
 also note dfs1-dfs-ref[OF ‹V a0›]
 finally show ?thesis .
qed

end

end
```

5.3 Imperative Implementation

```
theory Liveness-Subsumption-Impl
 imports Liveness-Subsumption-Map Heap-Hash-Map Worklist-Algorithms-Tracing
begin

no-notation Ref.update (‹- := -› 62)

locale Liveness-Search-Space-Key-Impl-Defs =
```

Liveness-Search-Space-Key-Defs - - - - - - key **for** *key* :: *'a* ⇒ *'k* +
fixes *A* :: *'a* ⇒ (*'ai* :: *heap*) ⇒ *assn*
fixes *succsi* :: *'ai* ⇒ *'ai list Heap*
fixes $a_0 i$ :: *'ai Heap*
fixes *Lei* :: *'ai* ⇒ *'ai* ⇒ *bool Heap*
fixes *keyi* :: *'ai* ⇒ *'ki* :: {*hashable*, *heap*} *Heap*
fixes *copyi* :: *'ai* ⇒ *'ai Heap*

locale *Liveness-Search-Space-Key-Impl* =
Liveness-Search-Space-Key-Impl-Defs +
Liveness-Search-Space-Key +
fixes *K*
assumes *pure-K*[*safe-constraint-rules*]: *is-pure K*
assumes *left-unique-K*[*safe-constraint-rules*]: *IS-LEFT-UNIQUE* (*the-pure K*)
assumes *right-unique-K*[*safe-constraint-rules*]: *IS-RIGHT-UNIQUE* (*the-pure K*)
assumes *refinements*[*sepref-fr-rules*]:
(*uncurry0* $a_0 i$, *uncurry0* (*RETURN* (*PR-CONST* a_0))) ∈ *unit-assn*^{k} →$_a$ *A*
(*uncurry Lei*,*uncurry* (*RETURN oo PR-CONST* (⊴))) ∈ *A*^{k} ∗$_a$ *A*^{k} →$_a$ *bool-assn*
(*succsi*,*RETURN o PR-CONST succs*) ∈ *A*^{k} →$_a$ *list-assn A*
(*keyi*,*RETURN o PR-CONST key*) ∈ *A*^{k} →$_a$ *K*
(*copyi*, *RETURN o COPY*) ∈ *A*^{k} →$_a$ *A*

context *Liveness-Search-Space-Key-Defs*
begin

— The lemma has this form to avoid unwanted eta-expansions. The eta-expansions arise from the type of *insert-map-set v S*.
lemma *insert-map-set-alt-def*: ((), *insert-map-set v S*) = (
let
k = *key v*; (*S'*, *S*) = *op-map-extract k S*
in
case S' of
Some S1 ⇒ ((), *S*(*k* ↦ (*insert v S1*)))
| *None* ⇒ ((), *S*(*k* ↦ {*v*}))
)

unfolding *insert-map-set-def op-map-extract-def* **by** (*auto simp*: *Let-def split*: *option.split*)

lemma *check-subsumption-map-set-alt-def*: *check-subsumption-map-set v S*

```
= (
  let
    k = key v; (S', S) = op-map-extract k S;
    S1' = (case S' of Some S1 ⇒ S1 | None ⇒ {})
  in (∃ x ∈ S1'. v ⊴ x)
)
```

unfolding *check-subsumption-map-set-def op-map-extract-def* **by** (*auto simp*: *Let-def*)

lemma *check-subsumption-map-set-extract*: (*S*, *check-subsumption-map-set v S*) = (
  let
    k = *key v*; (*S'*, *S*) = *op-map-extract k S*
  in (
    case S' of
      Some S1 ⇒ (*let r* = (∃ *x* ∈ *S1*. *v* ⊴ *x*) *in* (*op-map-update k S1 S*, *r*))
    | *None* ⇒ (*S*, *False*)
)
)

unfolding *check-subsumption-map-set-def op-map-extract-def op-map-update-def op-map-delete-def*
by (*auto simp*: *Let-def split*: *option.split*)

lemma *check-subsumption-map-list-extract*: (*S*, *check-subsumption-map-list v S*) = (
  let
    k = *key v*; (*S'*, *S*) = *op-map-extract k S*
  in (
    case S' of
      Some S1 ⇒ (*let r* = (∃ *x* ∈ *set S1*. *x* ⊴ *v*) *in* (*op-map-update k S1 S*, *r*))
    | *None* ⇒ (*S*, *False*)
)
)

unfolding *check-subsumption-map-list-def op-map-extract-def op-map-update-def op-map-delete-def*
by (*auto simp*: *Let-def split*: *option.split*)

lemma *push-map-list-alt-def*: ((), *push-map-list v S*) = (
  let
    k = *key v*; (*S'*, *S*) = *op-map-extract k S*

```
  in
    case S' of
      Some S1 ⇒ ((), S(k ↦ v # S1))
    | None ⇒ ((), S(k ↦ [v]))
)
```

unfolding *push-map-list-def op-map-extract-def* **by** (*auto simp*: *Let-def split*: *option.split*)

— The check for emptiness may be superfluous if we thread through the pre-condition

```
lemma pop-map-list-alt-def: ((), pop-map-list v S) = (
  let
    k = key v; (S', S) = op-map-extract k S
  in
    case S' of
      Some S1 ⇒ ((), S(k ↦ if op-list-is-empty S1 then [] else tl S1))
    | None ⇒ ((), S(k ↦ []))
)
```

unfolding *pop-map-list-def op-map-extract-def* **by** (*auto simp*: *Let-def split*: *option.split*)

```
lemmas alt-defs =
  insert-map-set-alt-def check-subsumption-map-set-extract
  check-subsumption-map-list-extract pop-map-list-alt-def push-map-list-alt-def

lemma dfs-map-alt-def:
  dfs-map = (λ P. do {
    (P,ST,r) ← RECT (λdfs (P,ST,v).
      let (ST, b) = (ST, check-subsumption-map-list v ST) in
      if b then RETURN (P, ST, True)
      else do {
        let (P, b1) = (P, check-subsumption-map-set v P) in
        if b1 then
          RETURN (P, ST, False)
        else do {
            let (-, ST1) = ((), push-map-list (COPY v) ST);
            (P1, ST2, r) ←
                nfoldli (succs v) (λ(-,-,b). ¬b) (λv' (P,ST,-). dfs (P,ST,v'))
(P,ST1,False);
            let (-, ST') = ((), pop-map-list (COPY v) ST2);
            let (-, P')  = ((), insert-map-set (COPY v) P1);
            TRACE (ExploredState);
```

```
        RETURN (P', ST', r)
      }
    }
  ) (P,Map.empty,a₀);
  RETURN (r, P)
})
unfolding dfs-map-def
unfolding Let-def
unfolding COPY-def
unfolding TRACE-bind
by auto

definition dfs-map' where
  dfs-map' a P ≡ do {
    (P,ST,r) ← RECT (λdfs (P,ST,v).
      let (ST, b) = (ST, check-subsumption-map-list v ST) in
      if b then RETURN (P, ST, True)
      else do {
        let (P, b1) = (P, check-subsumption-map-set v P) in
        if b1 then
          RETURN (P, ST, False)
        else do {
            let (-, ST1) = ((), push-map-list (COPY v) ST);
            (P1, ST2, r) ←
                nfoldli (succs v) (λ(-,-,b). ¬b) (λv' (P,ST,-). dfs (P,ST,v'))
(P,ST1,False);
            let (-, ST') = ((), pop-map-list (COPY v) ST2);
            let (-, P')  = ((), insert-map-set (COPY v) P1);
            TRACE (ExploredState);
            RETURN (P', ST', r)
          }
      }
    ) (P,Map.empty,a);
    RETURN (r, P)
  }

lemma dfs-map'-id:
  dfs-map' a₀ = dfs-map
  apply (subst dfs-map-alt-def)
  unfolding dfs-map'-def TRACE-bind ..

end

definition (in imp-map-delete) [code-unfold]: hms-delete = delete
```

lemma (**in** *imp-map-delete*) *hms-delete-rule* [*sep-heap-rules*]:
  <*hms-assn A m mi*> *hms-delete k mi* <*hms-assn A* (*m*(*k* := *None*))>$_t$
  unfolding *hms-delete-def hms-assn-def*
  apply (*sep-auto eintros del*: *exI*)
  subgoal for *mh*
    apply (*rule exI*[**where** *x* = *mh*(*k* := *None*)])
   apply (*rule ent-frame-fwd*[*OF map-assn-delete*[**where** *A* = *A*]], *frame-inference*)
    by (*sep-auto simp*: *map-upd-eq-restrict*)
  done

context *imp-map-delete*
begin

lemma *hms-delete-hnr*:
  (*uncurry hms-delete*, *uncurry* (*RETURN oo op-map-delete*)) ∈
  id-assn^k ∗$_a$ (*hms-assn A*)^d →$_a$ *hms-assn A*
  by *sepref-to-hoare sep-auto*

sepref-decl-impl *delete*: *hms-delete-hnr* **uses** *op-map-delete.fref*[**where** *V* = *Id*] **.**

end

lemma *fold-insert-set*:
  fold insert xs S = *set xs* ∪ *S*
  by (*induction xs arbitrary*: *S*) *auto*

lemma *set-alt-def*:
  set xs = *fold insert xs* {}
  by (*simp add*: *fold-insert-set*)

context *Liveness-Search-Space-Key-Impl*
begin

sepref-register
  PR-CONST a_0 *PR-CONST* (⊴) *PR-CONST succs PR-CONST key*

lemma [*def-pat-rules*]:
  a_0 ≡ *UNPROTECT* a_0 (⊴) ≡ *UNPROTECT* (⊴) *succs* ≡ *UNPROTECT succs key* ≡ *UNPROTECT key*
  by *simp-all*

abbreviation *passed-assn* ≡ *hm.hms-assn*′ *K* (*lso-assn A*)

lemma [*intf-of-assn*]:
intf-of-assn (*hm.hms-assn′ a b*) *TYPE*(('*aa*, '*bb*) *i-map*)
by *simp*

sepref-definition *dfs-map-impl* **is**
PR-CONST dfs-map :: *passed-assn*^{d} →$_a$ *prod-assn bool-assn passed-assn*
unfolding *PR-CONST-def*
apply (*subst dfs-map-alt-def*)
unfolding *TRACE′-def*[*symmetric*]
unfolding *alt-defs*
unfolding *Bex-set list-ex-foldli*
unfolding *fold-lso-bex*
unfolding *lso-fold-custom-empty hm.hms-fold-custom-empty HOL-list.fold-custom-empty*
by *sepref*

sepref-register *dfs-map*

lemmas [*sepref-fr-rules*] = *dfs-map-impl.refine-raw*

lemma *passed-empty-refine*[*sepref-fr-rules*]:
(*uncurry0 hm.hms-empty*, *uncurry0* (*RETURN* (*PR-CONST hm.op-hms-empty*)))
∈ *unit-assn*^{k} →$_a$ *passed-assn*
proof −
have *hn-refine emp hm.hms-empty emp* (*hm.hms-assn′ K* (*lso-assn A*))
(*RETURN hm.op-hms-empty*)
using *sepref-fr-rules*(*17*)[*simplified*] **.**
then show *?thesis*
by − (*rule hfrefI*, *auto simp*: *pure-unit-rel-eq-empty*)
qed

sepref-register *hm.op-hms-empty*

sepref-thm *dfs-map-impl′* **is**
uncurry0 (*do* {(*r*, *p*) ← *dfs-map Map.empty*; *RETURN r*}) :: *unit-assn*^{k}
→$_a$ *bool-assn*
unfolding *hm.hms-fold-custom-empty* **by** *sepref*

sepref-definition *dfs-map′-impl* **is**
uncurry dfs-map′
:: *A*^{d} ∗$_a$ (*hm.hms-assn′ K* (*lso-assn A*))^d →$_a$ *bool-assn* ×$_a$ *hm.hms-assn′*
K (*lso-assn A*)
unfolding *dfs-map′-def*

unfolding *PR-CONST-def TRACE′-def*[*symmetric*]
unfolding *alt-defs*
unfolding *Bex-set list-ex-foldli*
unfolding *fold-lso-bex*
unfolding *lso-fold-custom-empty hm.hms-fold-custom-empty HOL-list.fold-custom-empty*
by *sepref*

concrete-definition (**in** −) *dfs-map-impl′*
uses *Liveness-Search-Space-Key-Impl.dfs-map-impl′.refine-raw* **is** (*uncurry0 ?f*,-)∈-

lemma (**in** *Liveness-Search-Space-Key*) *dfs-map-empty*:
dfs-map Map.empty ≤ ⇓ (*bool-rel* ×$_r$ *map-set-rel*) *dfs-spec* **if** *V* a_0
proof −
have *liveness-compatible* {}
unfolding *liveness-compatible-def* **by** *auto*
have (*Map.empty*, {}) ∈ *map-set-rel*
unfolding *map-set-rel-def* **by** *auto*
note *dfs-map-dfs-refine*[*OF this* ‹*V* a_0›]
also note *dfs-correct*[*OF* ‹*V* a_0› ‹*liveness-compatible* {}›]
finally show *?thesis*
by *auto*
qed

lemma (**in** *Liveness-Search-Space-Key*) *dfs-map-empty-correct*:
do {(*r*, *p*) ← *dfs-map Map.empty*; *RETURN r*} ≤ *SPEC* (λ *r*. *r* ⟷ (∃ *x*. a_0 →∗ *x* ∧ *x* →$^+$ *x*))
if *V* a_0
supply *dfs-map-empty*[*OF* ‹*V* a_0›, *THEN Orderings.order.trans*, *refine-vcg*]
apply *refine-vcg*
unfolding *dfs-spec-def pw-le-iff* **by** (*auto simp*: *refine-pw-simps*)

lemma *dfs-map-impl′-hnr*:
(*uncurry0* (*dfs-map-impl′ succsi* a_0*i Lei keyi copyi*),
uncurry0 (*SPEC* (λ*r*. *r* = (∃ *x*. a_0 →∗ *x* ∧ *x* →$^+$ *x*)))
) ∈ *unit-assn*^{k} →$_a$ *bool-assn* **if** *V* a_0
using
dfs-map-impl′.refine[
OF Liveness-Search-Space-Key-Impl-axioms,
FCOMP dfs-map-empty-correct[*THEN Id-SPEC-refine*, *THEN nres-relI*],
OF ‹*V* a_0›
] .

lemma *dfs-map-impl′-hoare-triple*:

$<\uparrow(V\ a_0)>$
dfs-map-impl′ succsi a_0*i Lei keyi copyi*
$<\lambda r.\ \uparrow(r \longleftrightarrow (\exists\ x.\ a_0 \to* x \wedge x \to^+ x))>_t$
using *dfs-map-impl′-hnr*[*to-hnr*]
unfolding *hn-refine-def*
apply *clarsimp*
apply (*erule cons-post-rule*)
by (*sep-auto simp*: *pure-def*)

end

end
theory *Next-Key*
imports *Heap-Hash-Map*
begin

6 A Next-Key Operation for Hashmaps

lemma *insert-restrict-ran*:
insert v (*ran* (*m* |‘ (− {*k*}))) = *ran m* **if** *m k* = *Some v*
using *that* **unfolding** *ran-def restrict-map-def* **by** *force*

6.1 Definition and Key Properties

definition
hm-it-next-key ht = *do* {
n←*Array.len* (*the-array ht*);
if n = *0 then raise* (*STR ″Map is empty!″*)
else do {
i←*hm-it-adjust* (*n* − *1*) *ht*;
l←*Array.nth* (*the-array ht*) *i*;
case l of
[] ⇒ *raise* (*STR ″Map is empty!″*)
| (*x* # -) ⇒ *return* (*fst x*)
}
}

lemma *hm-it-next-key-rule*:
<*is-hashmap m ht*> *hm-it-next-key ht* <*λr. is-hashmap m ht* ∗ ↑ (*r* ∈ *dom m*)>
if *m* ≠ *Map.empty*
using *that*
unfolding *hm-it-next-key-def*

```
  unfolding is-hashmap-def unfolding is-hashmap'-def
  apply (sep-auto intro: hm-it-adjust-rule)
  subgoal
    using le-imp-less-Suc by fastforce
  subgoal premises prems for l xs i xs'
  proof −
    from prems have l ! i ≠ []
      by (auto simp: concat-take-Suc-empty)
    with prems(1−6) show ?thesis
      apply (cases xs')
       apply sep-auto
      apply sep-auto
      subgoal for a b list aa ba
        apply (rule weak-map-of-SomeI[of - b])
        apply clarsimp
        apply (rule bexI[of - l ! i])
        subgoal
          by (metis list.set-intros(1))
        subgoal
          by (cases length l; fastforce simp: take-Suc-conv-app-nth)
        done
      done
  qed
  done


definition
  next-key m = do {
    ASSERT (m ≠ Map.empty);
    k ← SPEC (λ k. k ∈ dom m);
    RETURN k
  }


lemma hm-it-next-key-next-key-aux:
  assumes is-pure K nofail (next-key m)
  shows
    <hm.assn K V m mi>
      hm-it-next-key mi
    <λr. ∃ A xa. hm.assn K V m mi ∗ K xa r ∗ true ∗ ↑ (RETURN xa ≤
next-key m)>
  using ‹nofail -› unfolding next-key-def
  apply (simp add: pw-bind-nofail pw-ASSERT(1))
  unfolding hm.assn-def hr-comp-def
```

```
  apply sep-auto
  subgoal for m'
    apply (rule cons-post-rule)
     apply (rule hm-it-next-key-rule)
     defer
     apply (sep-auto eintros del: exI)
    subgoal premises prems for x v'
    proof −
     from prems obtain k v where m k = Some v (x, k) ∈ the-pure K (v',
v) ∈ the-pure V
       apply atomize-elim
       by (meson map-rel-obtain2)
      with prems ‹is-pure K› show ?thesis
       apply −
       apply (rule exI[where x = k], rule exI[where x = m'])
       apply sep-auto
       apply (rule entailsI)
       apply sep-auto
       by (metis mod-pure-star-dist mod-star-trueI pure-def pure-the-pure)
    qed
    by force
  done

lemma hm-it-next-key-next-key:
  assumes CONSTRAINT is-pure K
  shows (hm-it-next-key, next-key) ∈ (hm.assn K V)^k →_a K
 using assms by sepref-to-hoare (sep-auto intro!: hm-it-next-key-next-key-aux)

lemma hm-it-next-key-next-key':
  (hm-it-next-key, next-key) ∈ (hm.hms-assn V)^k →_a id-assn
  unfolding hm.hms-assn-def
  apply sepref-to-hoare
  apply sep-auto
  subgoal for m m' mi
    unfolding next-key-def
    apply (rule cons-rule[where
          P' = is-hashmap mi m' * map-assn V m mi * ↑(mi ≠ Map.empty)
           and Q = λ x. is-hashmap mi m' * ↑ (x ∈ dom mi) * map-assn V
m mi]
        )
     apply (solves ‹sep-auto; unfold map-assn-def; auto simp: refine-pw-simps›)+
    by (rule norm-pre-pure-rule1, rule frame-rule[OF hm-it-next-key-rule[of
mi m']]) sep-auto
  done
```

lemma *no-fail-next-key-iff*:
nofail (*next-key m*) ⟷ *m* ≠ *Map.empty*
unfolding *next-key-def* **by** *auto*

context
fixes *mi m K*
assumes *map-rel*: (*mi*, *m*) ∈ ⟨*K*, *Id*⟩*map-rel*
begin

private lemma *k-aux*:
assumes *k* ∈ *dom mi* (*mi*, *m*) ∈ ⟨*K*, *Id*⟩*map-rel*
shows ∃ *k'*. (*k*, *k'*) ∈ *K*
using *assms* **unfolding** *map-rel-def* **by** *auto*

private lemma *k-aux2*:
assumes *k* ∈ *dom mi* (*k*, *k'*) ∈ *K*
shows *k'* ∈ *dom m*
using *assms map-rel* **unfolding** *map-rel-def* **by** (*cases m k'*) (*auto dest*: *fun-relD*)

private lemma *map-empty-iff*: *mi* ≠ *Map.empty* ⟷ *m* ≠ *Map.empty*
using *map-rel* **by** *auto*

private lemma *aux*:
assumes *RETURN k* ≤ *next-key mi*
shows *RETURN* (*SOME k'*. (*k*, *k'*) ∈ *K*) ≤ *next-key m*
using *map-rel assms*
unfolding *next-key-def*
apply (*cases m* ≠ *Map.empty*)
apply (*clarsimp simp*: *map-empty-iff*)
apply (*frule* (*1*) *k-aux*[*OF domI*])
apply (*drule someI-ex*)
apply (*auto dest*: *k-aux2*[*OF domI*])
done

private lemma *aux1*:
assumes *RETURN k* ≤ *next-key mi nofail* (*next-key m*)
shows (*k*, *SOME k'*. (*k*, *k'*) ∈ *K*) ∈ *K*
using *map-rel assms*
unfolding *next-key-def*
apply (*cases m* ≠ *Map.empty*)
apply (*clarsimp simp*: *map-empty-iff*)
apply (*drule* (*1*) *k-aux*[*OF domI*], *erule someI-ex*)

apply *auto*
done

lemmas *hm-it-next-key-next-key''-aux = aux aux1*

end

lemma *hm-it-next-key-next-key''*:
assumes *is-pure K*
shows (*hm-it-next-key*, *next-key*) ∈ (*hm.hms-assn' K V*)^k →$_a$ *K*
unfolding *hm.hms-assn'-def*
apply *sepref-to-hoare*
unfolding *hr-comp-def*
apply *sep-auto*
subgoal for *m m' mi*
using *hm-it-next-key-next-key'*[*to-hnr*, *unfolded hn-refine-def*, *of mi V m'*]
apply (*sep-auto simp*: *hn-ctxt-def*)
subgoal
unfolding *no-fail-next-key-iff* **by** *auto*
apply (*erule cons-post-rule*)
using ‹*is-pure K*›
apply *sep-auto*
apply (*erule* (*1*) *hm-it-next-key-next-key''-aux*)
apply (*subst pure-the-pure*[*symmetric*, *of K*], *assumption*)
apply (*sep-auto intro*: *hm-it-next-key-next-key''-aux simp*: *pure-def*)
done
done

6.2 Computing the Range of a Map

definition *ran-of-map* **where**
ran-of-map m ≡ *do*
{
(*xs*, *m*) ← *WHILEIT*
(λ (*xs*, *m'*). *finite* (*dom m'*) ∧ *ran m* = *ran m'* ∪ *set xs*) (λ (-, *m*). *Map.empty* ≠ *m*)
(λ (*xs*, *m*). *do*
{
k ← *next-key m*;
let (*x*, *m*) = *op-map-extract k m*;
ASSERT (*x* ≠ *None*);
RETURN (*the x* # *xs*, *m*)
}

```
      )
      ([], m);
    RETURN xs
  }
```

context
begin

private definition
ran-of-map-var = (*inv-image* (*measure* (*card o dom*)) (λ (*a*, *b*). *b*))

private lemma *wf-ran-of-map-var*:
wf ran-of-map-var
unfolding *ran-of-map-var-def* **by** *auto*

lemma *ran-of-map-correct*[*refine*]:
ran-of-map m ≤ *SPEC* (λ *r*. *set r* = *ran m*) **if** *finite* (*dom m*)
using *that* **unfolding** *ran-of-map-def next-key-def*
apply (*refine-vcg wf-ran-of-map-var*)
apply (*clarsimp*; *fail*)+
subgoal for *s xs m′ x v xs′ xs1 xs′1*
unfolding *dom-def* **by** (*clarsimp simp*: *map-upd-eq-restrict*, *auto dest*: *insert-restrict-ran*)
subgoal
unfolding *ran-of-map-var-def op-map-extract-def* **by** (*fastforce intro*: *card-Diff1-less*)
by *auto*

end — End of private context for auxiliary facts and definitions

sepref-register *next-key* :: (('*b*, '*a*) *i-map* ⇒ '*b nres*)

definition (**in** *imp-map-is-empty*) [*code-unfold*]: *hms-is-empty* ≡ *is-empty*

lemma (**in** *imp-map-is-empty*) *hms-empty-rule* [*sep-heap-rules*]:
<*hms-assn A m mi*> *hms-is-empty mi* <λ*r*. *hms-assn A m mi* ∗ ↑(*r* ⟷ *m*=*Map.empty*)>$_t$
unfolding *hms-is-empty-def hms-assn-def map-assn-def* **by** *sep-auto*

context *imp-map-is-empty*
begin

lemma *hms-is-empty-hnr*[*sepref-fr-rules*]:

(*hms-is-empty*, *RETURN o op-map-is-empty*) ∈ (*hms-assn A*)^k →$_a$ *bool-assn*
by *sepref-to-hoare sep-auto*

sepref-decl-impl *is-empty*: *hms-is-empty-hnr* **uses** *op-map-is-empty.fref*[**where** *V* = *Id*] **.**

end

lemma (**in** *imp-map*) *hms-assn'-id-hms-assn*:
hms-assn' id-assn A = *hms-assn A*
by (*subst hms-assn'-def*) *simp*

lemma [*intf-of-assn*]:
intf-of-assn (*hm.hms-assn' a b*) *TYPE*((*'aa*, *'bb*) *i-map*)
by *simp*

context
fixes *K* :: - ⇒ - :: {*hashable*, *heap*} ⇒ *assn*
assumes *is-pure-K*[*safe-constraint-rules*]: *is-pure K*
and *left-unique-K*[*safe-constraint-rules*]: *IS-LEFT-UNIQUE* (*the-pure K*)
and *right-unique-K*[*safe-constraint-rules*]: *IS-RIGHT-UNIQUE* (*the-pure K*)
notes [*sepref-fr-rules*] = *hm-it-next-key-next-key''*[*OF is-pure-K*]
begin

sepref-definition *ran-of-map-impl* **is**
ran-of-map :: (*hm.hms-assn' K A*)^d →$_a$ *list-assn A*
unfolding *ran-of-map-def hm.hms-assn'-id-hms-assn*[*symmetric*]
unfolding *op-map-is-empty-def*[*symmetric*]
unfolding *hm.hms-fold-custom-empty HOL-list.fold-custom-empty*
by *sepref*

end

lemmas *ran-of-map-impl-code*[*code*] =
ran-of-map-impl-def[*of pure Id*, *simplified*, *OF Sepref-Constraints.safe-constraint-rules*(*41*)]

context
notes [*sepref-fr-rules*] = *hm-it-next-key-next-key'*[*folded hm.hms-assn'-id-hms-assn*]
begin

sepref-definition *ran-of-map-impl'* **is**
ran-of-map :: (*hm.hms-assn A*)^d →$_a$ *list-assn A*

unfolding *ran-of-map-def hm.hms-assn′-id-hms-assn*[*symmetric*]
unfolding *op-map-is-empty-def*[*symmetric*]
unfolding *hm.hms-fold-custom-empty HOL-list.fold-custom-empty*
by *sepref*

end

end

7 Checking Leads-To Properties

7.1 Abstract Implementation

theory *Leadsto*
imports *Liveness-Subsumption Unified-PW*
begin

context *Subsumption-Graph-Pre-Nodes*
begin

context
assumes *finite-V*: *finite* {*x*. *V x*}
begin

lemma *steps-cycle-mono*:
assumes *G′.steps* (*x* # *ws* @ *y* # *xs* @ [*y*]) *x* ⪯ *x′ V x V x′*
shows ∃ *y′ xs′ ys′*. *y* ⪯ *y′* ∧ *G′.steps* (*x′* # *xs′* @ *y′* # *ys′* @ [*y′*])
proof −
let *?n* = *card* {*x*. *V x*} + *1*
let *?xs* = *y* # *concat* (*replicate ?n* (*xs* @ [*y*]))
from *assms*(*1*) **have** *G′.steps* (*x* # *ws* @ [*y*]) *G′.steps* (*y* # *xs* @ [*y*])
by (*auto intro*: *Graph-Start-Defs.graphI-aggressive2*)
with *G′.steps-replicate*[*of y* # *xs* @ [*y*] *?n*] **have** *G′.steps ?xs*
by *auto*
from *steps-mono*[*OF* ‹*G′.steps* (*x* # *ws* @ [*y*])› ‹*x* ⪯ *x′*› ‹*V x*› ‹*V x′*›]
obtain *ys* **where**
G′.steps (*x′* # *ys*) *list-all2* (⪯) (*ws* @ [*y*]) *ys*
by *auto*
then obtain *y′ ws′* **where** *G′.steps* (*x′* # *ws′* @ [*y′*]) *y* ⪯ *y′*
unfolding *list-all2-append1 list-all2-Cons1* **by** *auto*
with ‹*G′.steps* (*x* # *ws* @ [*y*])› **have** *V y V y′*
subgoal
using *G′-steps-V-last assms*(*3*) **by** *fastforce*
using *G′-steps-V-last* ‹*G′.steps* (*x′* # *ws′* @ [*y′*])› *assms*(*4*) **by** *fastforce*

```
  with steps-mono[OF ‹G'.steps ?xs› ‹y ⪯ y'›] obtain ys where ys:
    list-all2 (⪯) (concat (replicate ?n (xs @ [y]))) ys G'.steps (y' # ys)
    by auto
  let ?ys = filter ((⪯) y) ys
  have length ?ys ≥ ?n
    using list-all2-replicate-elem-filter[OF ys(1), of y]
    by auto
  have set ?ys ⊆ set ys
    by auto
  also have ... ⊆ {x. V x}
   using ‹G'.steps (y' # -)› ‹V y'› by (auto simp: list-all-iff dest: G'-steps-V-all)
  finally have ¬ distinct ?ys
    using distinct-card[of ?ys] ‹- >= ?n›
    by − (rule ccontr; drule distinct-length-le[OF finite-V]; simp)
  from not-distinct-decomp[OF this] obtain as y'' bs cs where ?ys = as @
[y''] @ bs @ [y''] @ cs
    by auto
  then obtain as' bs' cs' where
    ys = as' @ [y''] @ bs' @ [y''] @ cs'
    apply atomize-elim
    apply simp
   apply (drule filter-eq-appendD filter-eq-ConsD filter-eq-appendD[OF sym],
clarify)+
    apply clarsimp
    subgoal for as1 as2 bs1 bs2 cs'
      by (inst-existentials as1 @ as2 bs1 @ bs2) simp
    done
  from ‹G'.steps (y' # -)› have G'.steps (y' # as' @ y'' # bs' @ [y''])
   unfolding ‹ys = -› by (force intro: Graph-Start-Defs.graphI-aggressive2)
  moreover from ‹?ys = -› have y ⪯ y''
  proof −
    from ‹?ys = -› have y'' ∈ set ?ys by auto
    then show ?thesis by auto
  qed
  ultimately show ?thesis
    using ‹G'.steps (x' # ws' @ [y'])›
    by (inst-existentials y'' ws' @ y' # as' bs';
        fastforce intro: Graph-Start-Defs.graphI-aggressive1
        )
qed

lemma reaches-cycle-mono:
  assumes G'.reaches x y y →⁺ y x ⪯ x' V x V x'
  shows ∃ y'. y ⪯ y' ∧ G'.reaches x' y' ∧ y' →⁺ y'
```

proof −
from *assms* **obtain** *xs ys* **where** ∗: *G′.steps* (*x* # *xs*) *y* = *last* (*x* # *xs*)
G′.steps (*y* # *ys* @ [*y*])
apply *atomize-elim*
including *reaches-steps-iff*
apply *safe*
subgoal for *xs xs′*
by (*inst-existentials tl xs xs′*) *auto*
done
have ∗∗: ∃ *as*. *G′.steps* (*x* # *as* @ *last list* # *ys* @ [*last list*])
if *a1*: *G′.steps* (*x* # *a* # *list* @ *ys* @ [*last list*]) *list* ≠ []
for *a* :: *′a* **and** *list* :: *′a list*
proof −
from *that* **have** *butlast* (*a* # *list*) @ [*last list*] = *a* # *list*
by (*metis* (*no-types*) *append-butlast-last-id last-ConsR list.simps*(*3*))
then show *?thesis*
using *a1* **by** (*metis* (*no-types*) *Cons-eq-appendI append.assoc self-append-conv2*)
qed
from ∗ **obtain** *ws* **where** *G′.steps* (*x* # *ws* @ *y* # *ys* @ [*y*])
apply *atomize-elim*
apply (*cases xs*)
apply (*inst-existentials ys*)
apply *simp*
apply *rotate-tac*
apply (*rule G′.steps-append′*, *assumption*+, *simp*+)
apply *safe*
apply (*inst-existentials* [] :: *′a list*)
apply (*solves* ‹*auto dest*: *G′.steps-append*›)
apply (*drule G′.steps-append*)
apply *assumption*
apply *simp*
apply *simp*
by (*rule* ∗∗)
from *steps-cycle-mono*[*OF this* ‹*x* ≼ *x′*› ‹*V x*› ‹*V x′*›] **obtain** *y′ xs′ ys′*
where
y ≼ *y′*
G′.steps (*x′* # *xs′* @ *y′* # *ys′* @ [*y′*])
by *safe*
then have *G′.steps* (*x′* # *xs′* @ [*y′*]) *G′.steps* (*y′* # *ys′* @ [*y′*])
by (*force intro*: *Graph-Start-Defs.graphI-aggressive2*)+
with ‹*y* ≼ *y′*› **show** *?thesis*
including *reaches-steps-iff* **by** *force*
qed

end

end

locale *Leadsto-Search-Space* =
  A: *Search-Space'-finite E* a_0 - ($\preceq$) *empty*
  for *E* a_0 *empty* **and** *subsumes* :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** ‹$\preceq$› *50*)
  +
  fixes *P Q* :: $'a \Rightarrow$ *bool*
  assumes *P-mono*: $a \preceq a' \Longrightarrow \neg$ *empty a* $\Longrightarrow$ *P a* $\Longrightarrow$ *P a'*
  assumes *Q-mono*: $a \preceq a' \Longrightarrow \neg$ *empty a* $\Longrightarrow$ *Q a* $\Longrightarrow$ *Q a'*
  fixes *succs-Q* :: $'a \Rightarrow 'a$ *list*
  assumes *succs-Q-correct*: *A.reachable a* $\Longrightarrow$ *set* (*succs-Q a*) = {*y*. *E a y* $\wedge$ *Q y* $\wedge \neg$ *empty y*}
begin

sublocale *A'*: *Search-Space'-finite E* a_0 λ -. *False* ($\preceq$) *empty*
  apply *standard*
       apply (*rule A.refl A.trans A.mono A.empty-subsumes A.empty-mono A.empty-E*; *assumption*)+
    apply *assumption*
   apply *blast*
  apply (*rule A.finite-reachable*)
  done

sublocale *B*:
  Liveness-Search-Space
  λ *x y*. *E x y* $\wedge$ *Q y* $\wedge \neg$ *empty y* a_0 λ -. *False* ($\preceq$) λ *x*. *A.reachable x* $\wedge \neg$ *empty x*
  succs-Q
  apply *standard*
      apply (*rule A.refl A.trans*; *assumption*)+
  subgoal for *a b a'*
   by *safe* (*drule A.mono*; *auto intro*: *Q-mono dest*: *A.mono A.empty-mono*)
    apply *blast*
   apply (*solves* ‹*auto intro*: *A.finite-reachable*›)
  subgoal
    apply (*subst succs-Q-correct*)
    unfolding *Subgraph-Node-Defs.E'-def* **by** *auto*
  done

context
  fixes a_1 :: $'a$
begin

interpretation B':
Liveness-Search-Space
$\lambda\ x\ y.\ E\ x\ y \wedge Q\ y \wedge \neg\ empty\ y\ a_1\ \lambda$ -. *False* $(\preceq)\ \lambda\ x.$ *A.reachable* $x \wedge \neg$ *empty* x *succs-Q*
by *standard*

definition *has-cycle* **where**
has-cycle = *B'.dfs*

end

definition *leadsto* :: *bool nres* **where**
leadsto = *do* {
(r, *passed*) $\leftarrow$ *A'.pw-algo*;
let $P = \{x.\ x \in passed \wedge P\ x \wedge Q\ x\}$;
(r, -) $\leftarrow$
FOREACH$_C$ P $(\lambda(b,\text{-}).\ \neg b)$ $(\lambda v'\ (\text{-},P).$ *has-cycle* $v'\ P)$ (*False*,{});
RETURN r
}

definition
reaches-cycle a =
$(\exists\ b.\ (\lambda\ x\ y.\ E\ x\ y \wedge Q\ y \wedge \neg\ empty\ y)^{**}\ a\ b \wedge (\lambda\ x\ y.\ E\ x\ y \wedge Q\ y \wedge \neg\ empty\ y)^{++}\ b\ b)$

definition *leadsto-spec* **where**
leadsto-spec = *SPEC* $(\lambda\ r.\ r \longleftrightarrow (\exists\ a.$ *A.reachable* $a \wedge \neg$ *empty* $a \wedge P\ a \wedge Q\ a \wedge$ *reaches-cycle* $a))$

lemma
leadsto $\leq$ *leadsto-spec*
proof −
define *inv* **where**
inv $\equiv \lambda$ *passed it* (r, *passed'*).
$(r \longrightarrow (\exists\ a.$ *A.reachable* $a \wedge \neg$ *empty* $a \wedge P\ a \wedge Q\ a \wedge$ *reaches-cycle* $a))$
$\wedge\ (\neg\ r \longrightarrow$
$(\forall\ a \in passed - it.\ \neg$ *reaches-cycle* $a)$
$\wedge$ *B.liveness-compatible passed'*
$\wedge\ passed' \subseteq \{x.$ *A.reachable* $x \wedge \neg$ *empty* $x\}$
)

have [*simp*, *intro*]:
 ¬ *A′.F-reachable*
 unfolding *A′.F-reachable-def* **by** *simp*

have *B-reaches-empty*:
 ¬ *empty b* **if** ¬ *empty a B.reaches a b* **for** *a b*
 using *that*(*2*,*1*)**by** *induction auto*

interpret *Subgraph-Start E* a_0 λ *a x. E a x* $\wedge$ *Q x* $\wedge$ ¬ *empty x*
 by *standard auto*

have *B-A-reaches*:
 A.reaches a b **if** *B.reaches a b* **for** *a b*
 using *that* **by** (*rule reaches*)

have *reaches-iff*: *B.reaches a x* $\longleftrightarrow$ *B.G.G′.reaches a x*
 if *A.reachable a* ¬ *empty a* **for** *a x*
 unfolding *reaches-cycle-def*
 apply *standard*
 using *that*
  apply (*rotate-tac 3*)
  apply (*induction rule*: *rtranclp.induct*)
  apply *blast*
  apply (*rule rtranclp.rtrancl-into-rtrancl*)
  apply *assumption*
  apply (*subst B.G.E′-def*)
 subgoal for *a b c*
  by (*auto dest*: *B-reaches-empty*)
 subgoal
  by (*rule B.G.reaches*)
 done

have *reaches1-iff*: *B.reaches1 a x* $\longleftrightarrow$ *B.G.G′.reaches1 a x*
 if *A.reachable a* ¬ *empty a* **for** *a x*
 unfolding *reaches-cycle-def*
 apply *standard*
 subgoal
  using *that*
   apply (*rotate-tac 3*)
  apply (*induction rule*: *tranclp.induct*)
   apply (*solves* ‹*rule tranclp.intros*(*1*), *auto simp*: *B.G.E′-def*›)
  apply (
    rule tranclp.intros(*2*);
    auto 4 3 simp: *B.G.E′-def dest*:*B-reaches-empty tranclp-into-rtranclp*

```
      )
    done
  subgoal
    by (rule B.G.reaches1)
  done

 have reaches-cycle-iff: reaches-cycle a ⟷ (∃ x. B.G.G'.reaches a x ∧
B.G.G'.reaches1 x x)
  if A.reachable a ¬ empty a for a
  unfolding reaches-cycle-def
  apply (subst reaches-iff[OF that])
  using reaches1-iff B.G.G'-reaches-V that by blast

 have aux1:
  ¬ reaches-cycle x
  if
    ∀ a. A.reachable a ∧ ¬ empty a ⟶ (∃ x∈passed. a ≼ x)
    passed ⊆ {a. A.reachable a ∧ ¬ empty a}
    ∀ x ∈ passed. P x ∧ Q x ⟶ ¬ reaches-cycle x
    A.reachable x ¬ empty x P x Q x
  for x passed
 proof (rule ccontr, unfold not-not)
  assume reaches-cycle x
  from that obtain x' where x' ∈ passed x ≼ x'
    by auto
  with that have P x' Q x'
    by (auto intro: P-mono Q-mono)
  with ‹x' ∈ passed› that(3) have ¬ reaches-cycle x'
    by auto
  have A.reachable x' ¬ empty x'
    using ‹x' ∈ passed› that(2) A.empty-mono ‹x ≼ x'› that(5) by auto
  note reaches-cycle-iff' = reaches-cycle-iff[OF this] reaches-iff[OF this]
reaches1-iff[OF this]
  from ‹reaches-cycle x› obtain y where B.reaches x y B.reaches1 y y
    unfolding reaches-cycle-def by atomize-elim
  interpret
    Subsumption-Graph-Pre-Nodes
      (≼) A.subsumes-strictly λ x y. E x y ∧ Q y ∧ ¬ empty y
      λ x. A.reachable x ∧ ¬ empty x
    by standard (rule B.mono[simplified]; assumption)
  from ‹B.reaches x y› ‹x ≼ x'› ‹B.reaches1 y y› reaches-cycle-mono[OF
B.finite-V] obtain y' where
    y ≼ y' B.G.G'.reaches x' y' B.G.G'.reaches1 y' y'
    apply atomize-elim
```

```
    apply (subst (asm) reaches-iff[rotated 2])
     defer
     defer
     apply (subst (asm) reaches1-iff)
      defer
      defer
     using ‹A.reachable x› ‹¬ empty x› ‹A.reachable x'› ‹¬ empty x'›
‹B.reaches1 y y›
    by (auto simp: B.reaches1-reaches-iff2 dest!: B.G.G'-reaches-V)
   with ‹A.reachable x'› ‹¬ empty x'› have reaches-cycle x'
    unfolding reaches-cycle-iff'
    by auto
   with ‹¬ reaches-cycle x'› show False ..
  qed


  show ?thesis
   unfolding leadsto-def leadsto-spec-def
   apply (refine-rcg refine-vcg)
    subgoal for - r passed
    apply (refine-vcg
        FOREACHc-rule'[where I = inv {x ∈ passed. P x ∧ Q x}]
        )


    subgoal
     by (auto intro: finite-subset[OF - A.finite-reachable])


    subgoal
     unfolding inv-def B.liveness-compatible-def by auto


    subgoal for a1 it σ a passed'
     apply clarsimp
      subgoal premises prems
      proof −
       interpret B':
        Liveness-Search-Space
        λ x y. E x y ∧ Q y ∧ ¬ empty y a1 λ -. False (≼)
        λ x. A.reachable x ∧ ¬ empty x succs-Q
        by standard
       from ‹inv - - -› have
         B'.liveness-compatible passed' passed' ⊆ {x. A.reachable x ∧ ¬
```

empty x}
unfolding *inv-def* **by** *auto*
from *B′.dfs-correct*[*OF - this*] ‹*passed* ⊆ -› ‹a_1 ∈ -› ‹*it* ⊆ -› **have**
B′.dfs passed′ ≤ *B′.dfs-spec*
by *auto*
then show *?thesis*
unfolding *has-cycle-def*
apply (*rule order.trans*)
unfolding *B′.dfs-spec-def*
apply *clarsimp*
subgoal for *r passed′*
apply (*cases r*)
apply *simp*
subgoal
unfolding *inv-def*
using ‹*passed* ⊆ -› ‹a_1 ∈ -› ‹*it* ⊆ -›
apply *simp*
apply (*inst-existentials* a_1)
by (*auto 4 3 simp*: *reaches-cycle-iff*)
subgoal
using ‹*inv - - -*› ‹*passed* ⊆ -› *reaches-cycle-iff* **unfolding** *inv-def* **by** *blast*
done
done
qed
done

subgoal for *σ a b*
unfolding *inv-def* **by** (*auto dest*!: *aux1*)

subgoal for *it σ a b*
unfolding *inv-def* **by** *auto*
done
done
qed

definition *leadsto-spec-alt* **where**
leadsto-spec-alt =
SPEC (λ *r*.
r ⟷
(∃ *a*. (λ *x y*. *E x y* ∧ ¬ *empty y*)** a_0 *a* ∧ ¬ *empty a* ∧ *P a* ∧ *Q a* ∧ *reaches-cycle a*)

)

lemma *E-reaches-non-empty*:
(λ *x y*. *E x y* ∧ ¬ *empty y*)** *a b* **if** *a* →* *b A.reachable a* ¬ *empty b* **for** *a b*
using *that*
proof *induction*
case *base*
then show *?case* **by** *blast*
next
case (*step y z*)
from ‹*a* →* *y*› ‹*A.reachable a*› **have** *A.reachable y*
by − (*rule A.reachable-reaches*)
have ¬ *empty y*
proof (*rule ccontr*, *unfold not-not*)
assume *empty y*
from *A.empty-E*[*OF* ‹*A.reachable y*› ‹*empty y*› ‹*E y z*›] ‹¬ *empty z*›
show *False* **by** *blast*
qed
with *step* **show** *?case*
by (*auto intro*: *rtranclp.intros*(*2*))
qed

lemma *leadsto-spec-leadsto-spec-alt*:
leadsto-spec ≤ *leadsto-spec-alt*
unfolding *leadsto-spec-def leadsto-spec-alt-def*
by (*auto*
intro: *Subgraph.intro Subgraph.reaches*[*rotated*] *E-reaches-non-empty*[*OF* - *A.start-reachable*]
simp: *A.reachable-def*
)

end

end

7.2 Implementation on Maps

theory *Leadsto-Map*
imports *Leadsto Unified-PW-Hashing Liveness-Subsumption-Map Heap-Hash-Map Next-Key*
begin

definition *map-to-set* :: ('*b* ⇀ '*a set*) ⇒ '*a set* **where**

map-to-set m = (⋃ (*ran m*))

hide-const *wait*

definition
map-list-set-rel =
{(*ml*, *ms*). *dom ml* = *dom ms*
∧ (∀ *k* ∈ *dom ms*. *set* (*the* (*ml k*)) = *the* (*ms k*) ∧ *distinct* (*the* (*ml k*)))
∧ *finite* (*dom ml*)
}

context *Worklist-Map2-Defs*
begin

definition
add-pw′-map3 passed wait a ≡
nfoldli (*succs a*) (λ(-, -, *brk*). ¬*brk*)
(λ*a* (*passed*, *wait*, -).
do {
RETURN (
if empty a then
(*passed*, *wait*, *False*)
else if F′ a then (*passed*, *wait*, *True*)
else
let k = *key a*; *passed′* = (*case passed k of Some passed′* ⇒ *passed′* | *None* ⇒ [])
in
if ∃ *x* ∈ *set passed′*. *a* ⊴ *x then*
(*passed*, *wait*, *False*)
else
(*passed*(*k* ↦ (*a* # *passed′*)), *a* # *wait*, *False*)
)
}
)
(*passed*,*wait*,*False*)

definition
pw-map-inv3 ≡ λ (*passed*, *wait*, *brk*).
∃ *passed′*. (*passed*, *passed′*) ∈ *map-list-set-rel* ∧ *pw-map-inv* (*passed′*, *wait*, *brk*)

definition *pw-algo-map3* **where**

pw-algo-map3 = *do*
{
if F a_0 *then RETURN* (*True*, *Map.empty*)
else if empty a_0 *then RETURN* (*False*, *Map.empty*)
else do {
(*passed*, *wait*) ← *RETURN* ([*key* $a_0 \mapsto [a_0]$], [a_0]);
(*passed*, *wait*, *brk*) ← *WHILEIT pw-map-inv3* (λ (*passed*, *wait*, *brk*). ¬ *brk* ∧ *wait* ≠ [])
(λ (*passed*, *wait*, *brk*). *do*
{
(*a*, *wait*) ← *take-from-list wait*;
ASSERT (*reachable a*);
if empty a then RETURN (*passed*, *wait*, *brk*) *else add-pw′-map3 passed wait a*
}
)
(*passed*, *wait*, *False*);
RETURN (*brk*, *passed*)
}
}

end — Worklist Map 2 Defs

lemma *map-list-set-rel-empty*[*refine*, *simp*, *intro*]:
(*Map.empty*, *Map.empty*) ∈ *map-list-set-rel*
unfolding *map-list-set-rel-def* **by** *auto*

lemma *map-list-set-rel-single*:
(*ml*(*key* $a_0 \mapsto [a_0]$), *ms*(*key* $a_0 \mapsto \{a_0\}$)) ∈ *map-list-set-rel* **if** (*ml*, *ms*) ∈ *map-list-set-rel*
using *that* **unfolding** *map-list-set-rel-def* **by** *auto*

context *Worklist-Map2*
begin

lemma *refine-start*[*refine*]:
(([*key* $a_0 \mapsto [a_0]$], [a_0]), [*key* $a_0 \mapsto \{a_0\}$], [a_0]) ∈ *map-list-set-rel* $\times_r$ *Id*
by (*simp add*: *map-list-set-rel-single*)

lemma *pw-map-inv-ref*:
pw-map-inv (*x1*, *x2*, *x3*) ⟹ (*x1a*, *x1*) ∈ *map-list-set-rel* ⟹ *pw-map-inv3* (*x1a*, *x2*, *x3*)
unfolding *pw-map-inv3-def* **by** *auto*

lemma *refine-aux*[*refine*]:
(*x1*, *x*) ∈ *map-list-set-rel* ⟹ ((*x1*, *x2*, *False*), *x*, *x2*, *False*) ∈ *map-list-set-rel* ×$_r$ *Id* ×$_r$ *Id*
by *simp*

lemma *map-list-set-relD*:
ms k = *Some* (*set xs*) **if** (*ml*, *ms*) ∈ *map-list-set-rel ml k* = *Some xs*
using *that* **unfolding** *map-list-set-rel-def*
by *clarsimp* (*metis* (*mono-tags*, *lifting*) *domD domI option.sel*)

lemma *map-list-set-rel-distinct*:
distinct xs **if** (*ml*, *ms*) ∈ *map-list-set-rel ml k* = *Some xs*
using *that* **unfolding** *map-list-set-rel-def* **by** *clarsimp* (*metis domI option.sel*)

lemma *map-list-set-rel-NoneD1*[*dest*, *intro*]:
ms k = *None* **if** (*ml*, *ms*) ∈ *map-list-set-rel ml k* = *None*
using *that* **unfolding** *map-list-set-rel-def* **by** *auto*

lemma *map-list-set-rel-NoneD2*[*dest*, *intro*]:
ml k = *None* **if** (*ml*, *ms*) ∈ *map-list-set-rel ms k* = *None*
using *that* **unfolding** *map-list-set-rel-def* **by** *auto*

lemma *map-list-set-rel-insert*:
(*ml*, *ms*) ∈ *map-list-set-rel* ⟹
ml (*key a*) = *Some xs* ⟹
ms (*key a*) = *Some* (*set xs*) ⟹
a ∉ *set xs* ⟹
(*ml*(*key a* ↦ *a* # *xs*), *ms*(*key a* ↦ *insert a* (*set xs*))) ∈ *map-list-set-rel*
apply (*frule map-list-set-rel-distinct*) **unfolding** *map-list-set-rel-def* **by** *auto*

lemma *add-pw′-map3-ref*:
add-pw′-map3 ml xs a ≤ ⇓ (*map-list-set-rel* ×$_r$ *Id*) (*add-pw′-map2 ms xs a*)
if (*ml*, *ms*) ∈ *map-list-set-rel* ¬ *empty a*
using *that* **unfolding** *add-pw′-map3-def add-pw′-map2-def*
apply *refine-rcg*
apply *refine-dref-type*
apply (*simp*; *fail*)
apply (*simp*; *fail*)
apply (*simp*; *fail*)
apply (*clarsimp simp*: *Let-def*)

```
  apply safe
  subgoal
    by (auto dest: map-list-set-relD[OF - sym])
  subgoal
    by (simp split: option.split-asm)
       (metis (mono-tags, lifting)
          map-list-set-relD map-list-set-rel-NoneD1 option.collapse option.sel
option.simps(3)
       )
  subgoal premises prems for a ms x2a ml x2c
  proof (cases ml (key a))
    case None
    with ‹(ml, ms) ∈ map-list-set-rel› have ms (key a) = None
      by auto
    with None ‹(ml, ms) ∈ map-list-set-rel› show ?thesis
      by (auto intro: map-list-set-rel-single)
  next
    case (Some xs)
   from map-list-set-relD[OF ‹(ml, ms) ∈ map-list-set-rel› ‹ml - = -›] have
      ms (key a) = Some (set xs)
      by auto
    moreover from prems have a ∉ set xs
      by (metis Some empty-subsumes′ local.eq-refl)
    ultimately show ?thesis
    using Some ‹(ml, ms) ∈ map-list-set-rel› by (auto intro: map-list-set-rel-insert)
  qed
  done

lemma pw-algo-map3-ref[refine]:
  pw-algo-map3 ≤ ⇓ (Id ×r map-list-set-rel) pw-algo-map2
  unfolding pw-algo-map3-def pw-algo-map2-def
  apply refine-rcg
            apply refine-dref-type
            apply (simp; fail)+
         apply (clarsimp, rule refine-aux; assumption)
  by (auto intro: add-pw′-map3-ref pw-map-inv-ref)

lemma pw-algo-map2-ref′:
  pw-algo-map2 ≤ ⇓ (bool-rel ×r map-set-rel) pw-algo
proof −
  note pw-algo-map2-ref
  also note pw-algo-map-ref
  also note pw-algo-unified-ref
  finally show ?thesis .
```

qed

lemma *pw-algo-map3-ref'*[*refine*]:
pw-algo-map3 $\leq \Downarrow$ (*bool-rel* $\times_r$ (*map-list-set-rel O map-set-rel*)) *pw-algo*
proof −
have [*simp*]:
((*bool-rel* $\times_r$ *map-list-set-rel*) *O* (*bool-rel* $\times_r$ *map-set-rel*))
= (*bool-rel* $\times_r$ (*map-list-set-rel O map-set-rel*))
unfolding *relcomp-def prod-rel-def* **by** *auto*
note *pw-algo-map3-ref*
also note *pw-algo-map2-ref'*
finally show *?thesis*
by (*simp add*: *conc-fun-chain*)
qed

end — Worklist Map 2 Defs

lemma (**in** *Worklist-Map2-finite*) *map-set-rel-finite-domI*[*intro*]:
finite (*dom m*) **if** (*m*, *S*) ∈ *map-set-rel*
using *that* **unfolding** *map-set-rel-def* **by** *auto*

lemma (**in** *Worklist-Map2-finite*) *map-set-rel-finiteI*:
finite S **if** (*m*, *S*) ∈ *map-set-rel*
using *that* **unfolding** *map-set-rel-def*
apply *clarsimp*
apply (*rule finite-Union*)
apply (*solves* ‹*auto intro*: *map-dom-ran-finite*›)+
apply (*solves* ‹*auto simp*: *ran-def*›)
done

lemma (**in** *Worklist-Map2-finite*) *map-set-rel-finite-ranI*[*intro*]:
finite S' **if** (*m*, *S*) ∈ *map-set-rel S'* ∈ *ran m*
using *that* **unfolding** *map-set-rel-def ran-def* **by** *auto*

locale *Leadsto-Search-Space-Key* =
A: *Worklist-Map2* - - - - - - - *succs1* +
Leadsto-Search-Space **for** *succs1*
begin

sublocale *A'*: *Worklist-Map2-finite* a_0 λ -. *False* ($\preceq$) *empty* ($\trianglelefteq$) *E key succs1* λ -. *False*
by (*standard*; *blast intro*!: *A.succs-correct*)

interpretation *B*:

Liveness-Search-Space-Key
λ x y. E x y ∧ Q y ∧ ¬ empty y a₀ λ -. False (≼) λ x. A.reachable x ∧ ¬ empty x
succs-Q key
by *standard* (*auto intro*!: *A.empty-subsumes'*)

context
fixes $a_1 :: 'a$
begin

interpretation B':
Liveness-Search-Space-Key-Defs
a_1 *λ -. False (≼) λ x. A.reachable x ∧ ¬ empty x*
succs-Q λ x y. E x y ∧ Q y ∧ ¬ empty y key **.**

definition *has-cycle-map* **where**
has-cycle-map = B'*.dfs-map*

context
assumes *A.reachable* a_1
begin

interpretation B':
Liveness-Search-Space-Key
λ x y. E x y ∧ Q y ∧ ¬ empty y a_1 *λ -. False (≼) λ x. A.reachable x ∧ ¬ empty x*
succs-Q key
by *standard*

lemmas *has-cycle-map-ref*[*refine*] = B'*.dfs-map-dfs-refine*[*folded has-cycle-map-def has-cycle-def*]

end

end

definition *outer-inv* **where**
outer-inv passed done todo ≡ *λ* (*r*, *passed'*).
(*r* ⟶ (∃ *a. A.reachable a ∧ ¬ empty a ∧ P a ∧ Q a ∧ reaches-cycle a*))
∧ (¬ *r* ⟶
(∀ *a* ∈ ⋃ *done. P a ∧ Q a* ⟶ ¬ *reaches-cycle a*)
∧ *B.liveness-compatible passed'*
∧ *passed'* ⊆ {*x. A.reachable x ∧ ¬ empty x*}

)

definition *inner-inv* **where**
inner-inv passed done todo ≡ λ (*r*, *passed'*).
(*r* ⟶ (∃ *a*. *A.reachable a* ∧ ¬ *empty a* ∧ *P a* ∧ *Q a* ∧ *reaches-cycle a*))
∧ (¬ *r* ⟶
(∀ *a* ∈ *done*. *P a* ∧ *Q a* ⟶ ¬ *reaches-cycle a*)
∧ *B.liveness-compatible passed'*
∧ *passed'* ⊆ {*x*. *A.reachable x* ∧ ¬ *empty x*}
)

definition *leadsto'* :: *bool nres* **where**
leadsto' = *do* {
(*r*, *passed*) ← *A'.pw-algo-map2*;
let passed = *ran passed*;
(*r*, -) ← *FOREACHcdi* (*outer-inv passed*) *passed* (λ(*b*,-). ¬*b*)
(λ *passed'* (-,*acc*).
FOREACHcdi (*inner-inv acc*) *passed'* (λ(*b*,-). ¬*b*)
(λ*v'* (-,*passed*).
do {
ASSERT(*A.reachable v'* ∧ ¬ *empty v'*);
if P v' ∧ *Q v' then has-cycle v' passed else RETURN* (*False*, *passed*)
}
)
(*False*, *acc*)
)
(*False*, {});
RETURN r
}

lemma *leadsto'-correct*:
leadsto' ≤ *leadsto-spec*
proof −
define *inv* **where**
inv ≡ λ *passed it* (*r*, *passed'*).
(*r* ⟶ (∃ *a*. *A.reachable a* ∧ ¬ *empty a* ∧ *P a* ∧ *Q a* ∧ *reaches-cycle a*))
∧ (¬ *r* ⟶
(∀ *a* ∈ *passed* − *it*. ¬ *reaches-cycle a*)
∧ *B.liveness-compatible passed'*

```
    ∧ passed' ⊆ {x. A.reachable x ∧ ¬ empty x}
  )

have [simp, intro]:
  ¬ A'.F-reachable
  unfolding A'.F-reachable-def by simp

have B-reaches-empty:
  ¬ empty b if ¬ empty a B.reaches a b for a b
  using that(2,1)by induction auto

interpret Subgraph-Start E a₀ λ a x. E a x ∧ Q x ∧ ¬ empty x
  by standard auto

have B-A-reaches:
  A.reaches a b if B.reaches a b for a b
  using that by (rule reaches)

have reaches-iff: B.reaches a x ⟷ B.G.G'.reaches a x
  if A.reachable a ¬ empty a for a x
  unfolding reaches-cycle-def
  apply standard
  using that
    apply (rotate-tac 3)
   apply (induction rule: rtranclp.induct)
    apply blast
   apply (rule rtranclp.rtrancl-into-rtrancl)
    apply assumption
   apply (subst B.G.E'-def)
  subgoal for a b c
    by (auto dest: B-reaches-empty)
  subgoal
    by (rule B.G.reaches)
  done

have reaches1-iff: B.reaches1 a x ⟷ B.G.G'.reaches1 a x
  if A.reachable a ¬ empty a for a x
  unfolding reaches-cycle-def
  apply standard
  subgoal
    using that
      apply (rotate-tac 3)
    apply (induction rule: tranclp.induct)
```

```
      apply (solves ‹rule tranclp.intros(1), auto simp: B.G.E'-def›)
     apply (
        rule tranclp.intros(2);
       auto 4 3 simp: B.G.E'-def dest:B-reaches-empty tranclp-into-rtranclp
        )
     done
   subgoal
     by (rule B.G.reaches1)
   done

  have reaches-cycle-iff: reaches-cycle a ⟷ (∃ x. B.G.G'.reaches a x ∧
B.G.G'.reaches1 x x)
   if A.reachable a ¬ empty a for a
   unfolding reaches-cycle-def
   apply (subst reaches-iff[OF that])
   using reaches1-iff B.G.G'-reaches-V that by blast

  have aux1:
   ¬ reaches-cycle x
   if
     ∀ a. A.reachable a ∧ ¬ empty a ⟶ (∃ x∈passed. a ≼ x)
     passed ⊆ {a. A.reachable a ∧ ¬ empty a}
     ∀ y ∈ ran passed'. ∀ x ∈ y. P x ∧ Q x ⟶ ¬ reaches-cycle x
     A.reachable x ¬ empty x P x Q x
     (passed', passed) ∈ A'.map-set-rel
   for x passed passed'
  proof (rule ccontr, unfold not-not)
   assume reaches-cycle x
   from that obtain x' where x':x' ∈ passed x ≼ x'
     by auto
   with ‹(-, -) ∈ -› obtain y where y: y ∈ ran passed' x' ∈ y
     unfolding A'.map-set-rel-def by auto
   from x' that have P x' Q x'
     by (auto intro: P-mono Q-mono)
   with ‹x' ∈ passed› that(3) y have ¬ reaches-cycle x'
     by auto
   have A.reachable x' ¬ empty x'
     using ‹x' ∈ passed› that(2) A.empty-mono ‹x ≼ x'› that(5) by auto
    note reaches-cycle-iff' = reaches-cycle-iff[OF this] reaches-iff[OF this]
reaches1-iff[OF this]
   from ‹reaches-cycle x› obtain y where B.reaches x y B.reaches1 y y
     unfolding reaches-cycle-def by atomize-elim
   interpret
     Subsumption-Graph-Pre-Nodes
```

```
      (≼) A.subsumes-strictly λ x y. E x y ∧ Q y ∧ ¬ empty y
      λ x. A.reachable x ∧ ¬ empty x
    by standard (rule B.mono[simplified]; assumption)
  from ‹B.reaches x y› ‹x ≼ x'› ‹B.reaches1 y y› reaches-cycle-mono[OF
B.finite-V] obtain y' where
    y ≼ y' B.G.G'.reaches x' y' B.G.G'.reaches1 y' y'
    apply atomize-elim
    apply (subst (asm) reaches-iff[rotated 2])
     defer
     defer
     apply (subst (asm) reaches1-iff)
      defer
      defer
     using ‹A.reachable x› ‹¬ empty x› ‹A.reachable x'› ‹¬ empty x'›
‹B.reaches1 y y›
    by (auto simp: B.reaches1-reaches-iff2 dest!: B.G.G'-reaches-V)
  with ‹A.reachable x'› ‹¬ empty x'› have reaches-cycle x'
    unfolding reaches-cycle-iff'
    by auto
  with ‹¬ reaches-cycle x'› show False ..
 qed

 note [refine-vcg] = A'.pw-algo-map2-correct[THEN order.trans]

 show ?thesis
  unfolding leadsto'-def leadsto-spec-def
  apply (refine-rcg refine-vcg)


  subgoal
    by (auto intro: map-dom-ran-finite)


    subgoal
      unfolding outer-inv-def B.liveness-compatible-def by simp


    subgoal
      by auto


    subgoal for x a b S1 S2 todo σ aa passed
      unfolding inner-inv-def outer-inv-def by simp
```

subgoal
unfolding *inner-inv-def outer-inv-def A′.map-set-rel-def* **by** *auto*

subgoal
unfolding *inner-inv-def outer-inv-def A′.map-set-rel-def* **by** *auto*

subgoal for - - *b S1 S2 xa σ aa passed S1′ S2′ a*$_1$ *σ′ ab passed′*
unfolding *outer-inv-def*
apply *clarsimp*
subgoal premises *prems* **for** *p*
proof −
from *prems* **have** $a_1 \in p$
unfolding *A′.map-set-rel-def* **by** *auto*
with ‹*passed* ⊆ -› ‹*p* ⊆ -› **have** *A.reachable a*$_1$
by *auto*
interpret *B′*:
Liveness-Search-Space
λ *x y. E x y* ∧ *Q y* ∧ ¬ *empty y a*$_1$ λ -. *False* (⪯)
λ *x. A.reachable x* ∧ ¬ *empty x succs-Q*
by *standard*
from ‹*inner-inv* - - - -› **have**
B′.liveness-compatible passed′ passed′ ⊆ {*x. A.reachable x* ∧ ¬ *empty x*}
unfolding *inner-inv-def* **by** *auto*
from *B′.dfs-correct*[*OF* - *this*] ‹*passed* ⊆ -› ‹*a*$_1$ ∈ -› ‹*p* ⊆ -› **have**
B′.dfs passed′ ≤ *B′.dfs-spec*
by *auto*
then show *?thesis*
unfolding *has-cycle-def*
apply (*rule order.trans*)
unfolding *B′.dfs-spec-def*
apply *clarsimp*
subgoal for *r passed1*
apply (*cases r*)
apply *simp*
subgoal
unfolding *inner-inv-def*
using ‹*passed* ⊆ -› ‹*a*$_1$ ∈ -› ‹*p* ⊆ -›
apply *simp*
apply (*inst-existentials a*$_1$)
by (*auto 4 3 simp*: *reaches-cycle-iff intro*: *prems*)

```
        subgoal
          using ‹inner-inv - - - -› ‹passed ⊆ -› ‹a1 ∈ -› ‹p ⊆ -›
reaches-cycle-iff
          unfolding inner-inv-def by auto
          done
        done
      qed
      done


      subgoal for x a b S1 S2 xa σ aa ba S1a S2a xb σ' ab bb
        unfolding inner-inv-def by auto


      subgoal for x a b S1 S2 xa σ aa ba S1a S2a σ'
        unfolding inner-inv-def outer-inv-def by auto


      subgoal for x a b S1 S2 xa σ aa ba σ'
        unfolding inner-inv-def outer-inv-def by auto


      subgoal for x a b S1 S2 σ aa ba
        unfolding outer-inv-def by auto



      subgoal for x a b σ aa ba
        unfolding outer-inv-def by (auto dest!: aux1)

      done
    qed

lemma init-ref[refine]:
  ((False, Map.empty), False, {}) ∈ bool-rel ×r A'.map-set-rel
  unfolding A'.map-set-rel-def by auto

lemma has-cycle-map-ref'[refine]:
  assumes (P1, P1') ∈ A'.map-set-rel (a, a') ∈ Id A.reachable a ¬ empty
a
  shows has-cycle-map a P1 ≤ ⇓ (bool-rel ×r A'.map-set-rel) (has-cycle a'
P1')
  using has-cycle-map-ref assms by auto
```

```
definition leadsto-map3' :: bool nres where
  leadsto-map3' = do {
    (r, passed) ← A'.pw-algo-map2;
    let passed = ran passed;
    (r, -) ← FOREACHcd passed (λ(b,-). ¬b)
      (λ passed' (-,acc).
        do {
          passed' ← SPEC (λl. set l = passed');
          nfoldli passed' (λ(b,-). ¬b)
            (λv' (-,passed).
             if P v' ∧ Q v' then has-cycle-map v' passed else RETURN (False,
passed)
            )
            (False, acc)
        }
      )
      (False, Map.empty);
    RETURN r
  }

definition pw-algo-map2-copy = A'.pw-algo-map2

lemma [refine]:
  A'.pw-algo-map2 ≤
    ⇓ (br id (λ (-, m). finite (dom m) ∧ (∀ k S. m k = Some S ⟶ finite
S))) pw-algo-map2-copy
proof −
  have [refine]:
    (x, x') ∈ Id ⟹
     x' = (x1, x2) ⟹
     x = (x1a, x2a) ⟹
     A'.pw-map-inv (x1, x2, False)
    ⟹ ((x1a, x2a, False), x1, x2, False) ∈
       (br id (λ m. finite (dom m) ∧ (∀ k S. m k = Some S ⟶ finite S)))
×r Id ×r Id
    for x x' x1 x2 x1a x2a
    by (auto simp: br-def A'.pw-map-inv-def A'.map-set-rel-def)
  show ?thesis
    unfolding pw-algo-map2-copy-def
    unfolding A'.pw-algo-map2-def
    apply refine-rcg
                apply refine-dref-type
                prefer 5
                apply assumption
```

```
            apply (assumption | solves ‹simp add: br-def› | solves ‹auto
simp: br-def›)+
    subgoal
      apply (clarsimp simp: br-def)
      subgoal premises prems
        using ‹finite -› ‹∀ k S. - ⟶ finite -›
        unfolding A'.add-pw'-map2-def
        apply refine-rcg
          apply refine-dref-type
          apply (auto simp: Let-def split!: option.split)
        done
      done
    by (simp add: br-def)
qed

lemma leadsto-map3'-ref[refine]:
  leadsto-map3' ≤ ⇓ Id leadsto'
  unfolding leadsto-map3'-def leadsto'-def
  apply (subst (2) pw-algo-map2-copy-def[symmetric])
  apply (subst (2) FOREACHcdi-def)
  apply (subst (2) FOREACHcd-def)
  apply refine-rcg
            apply refine-dref-type
  by (auto simp: br-def intro: map-dom-ran-finite)

definition leadsto-map3 :: bool nres where
  leadsto-map3 = do {
    (r, passed) ← A'.pw-algo-map3;
    let passed = ran passed;
    (r, -) ← FOREACHcd passed (λ(b,-). ¬b)
      (λ passed' (-,acc).
          nfoldli passed' (λ(b,-). ¬b)
            (λv' (-,passed).
             if P v' ∧ Q v' then has-cycle-map v' passed else RETURN (False,
passed)
            )
            (False, acc)
      )
      (False, Map.empty);
    RETURN r
  }

lemma start-ref:
  ((False, Map.empty), False, Map.empty) ∈ Id ×r map-list-set-rel
```

by *simp*

lemma *map-list-set-rel-ran-set-rel*:
(*ran ml*, *ran ms*) ∈ ⟨*br set* (λ-. *True*)⟩*set-rel* **if** (*ml*, *ms*) ∈ *map-list-set-rel*
using *that* **unfolding** *map-list-set-rel-def set-rel-def*
apply *safe*
subgoal for *x*
by (*auto simp*: *ran-def dom-def in-br-conv dest*: *A.map-list-set-relD*[*OF that*])
subgoal premises *prems* **for** *x′*
proof −
from *prems*(*4*) **obtain** *a* **where** *ms a* = *Some x′*
unfolding *ran-def* **by** *clarsimp*
with *prems*(*1*) **obtain** *m′* **where**
ml a = *Some* (*m′ a*)
by (*fastforce simp*: *dom-def ran-def*)
with *prems*(*2*) ‹*ms a* = -› **show** *?thesis*
by (*fastforce simp*: *in-br-conv dom-def ran-def*)
qed
done

lemma *Id-list-rel-ref*:
(*x′a*, *x′a*) ∈ ⟨*Id*⟩*list-rel*
by *simp*

lemma *map-list-set-rel-finite-ran*:
finite (*ran ml*) **if** (*ml*, *ms*) ∈ *map-list-set-rel*
using *that* **unfolding** *map-list-set-rel-def* **by** (*auto intro*: *map-dom-ran-finite*)

lemma *leadsto-map3-ref*[*refine*]:
leadsto-map3 ≤ ⇓ *Id leadsto′*
unfolding *leadsto-map3-def leadsto′-def*
apply (*subst* (*2*) *FOREACHcdi-def*)
apply (*subst* (*2*) *FOREACHcd-def*)
apply (*refine-rcg map-list-set-rel-ran-set-rel map-list-set-rel-finite-ran*)
prefer *4*
apply (*rule rhs-step-bind-SPEC*)
apply (*clarsimp simp*: *br-def*; *rule HOL.refl*; *fail*)
apply (*refine-rcg Id-list-rel-ref*; *simp*; *fail*)
by *auto*

definition *leadsto-map4* :: *bool nres* **where**
leadsto-map4 = *do* {
(*r*, *passed*) ← *A′.pw-algo-map3*;

```
    ASSERT (finite (dom passed));
    passed ← ran-of-map passed;
    (r, -) ← nfoldli passed (λ(b,-). ¬b)
      (λ passed' (-,acc).
          nfoldli passed' (λ(b,-). ¬b)
            (λv' (-,passed).
             if P v' ∧ Q v' then has-cycle-map v' passed else RETURN (False,
passed)
            )
            (False, acc)
      )
      (False, Map.empty);
    RETURN r
  }

lemma ran-of-map-ref:
   ran-of-map m ≤ SPEC (λc. (c, ran m') ∈ br set (λ -. True)) if finite
(dom m) (m, m') ∈ Id
  using ran-of-map-correct[OF that(1)] that(2) unfolding br-def by (simp
add: pw-le-iff)

lemma aux-ref:
  (xa, x'a) ∈ Id ⟹
       x'a = (x1b, x2b) ⟹ xa = (x1c, x2c) ⟹ (x1c, x1b) ∈ bool-rel
  by simp

definition foo = A'.pw-algo-map3

lemma [refine]:
  A'.pw-algo-map3 ≤ ⇓ (br id (λ (-, m). finite (dom m))) foo
proof −
  have [refine]:
    (x, x') ∈ Id ⟹
     x' = (x1, x2) ⟹
     x = (x1a, x2a) ⟹
     A'.pw-map-inv3 (x1, x2, False)
    ⟹ ((x1a, x2a, False), x1, x2, False) ∈ (br id (λ m. finite (dom m)))
×r Id ×r Id
    for x x' x1 x2 x1a x2a
    by (auto simp: br-def A'.pw-map-inv3-def map-list-set-rel-def)
  show ?thesis
    unfolding foo-def
    unfolding A'.pw-algo-map3-def
    apply refine-rcg
```

```
                apply refine-dref-type
                prefer 5
                apply assumption
               apply (assumption | solves ‹simp add: br-def› | solves ‹auto
simp: br-def›)+
    subgoal
      apply (clarsimp simp: br-def)
      subgoal premises prems
        using ‹finite -›
        unfolding A'.add-pw'-map3-def
        apply refine-rcg
          apply refine-dref-type
          apply (auto simp: Let-def)
        done
      done
    by (simp add: br-def)
qed

lemma leadsto-map4-ref[refine]:
  leadsto-map4 ≤ ⇓ Id leadsto-map3
  unfolding leadsto-map4-def leadsto-map3-def FOREACHcd-def
  apply (subst (2) foo-def[symmetric])
  apply (refine-rcg ran-of-map-ref)
     apply refine-dref-type
     apply (simp add: br-def; fail)
    apply (simp add: br-def; fail)
   apply (rule rhs-step-bind-SPEC)
  by (auto simp: br-def)

definition leadsto-map4' :: bool nres where
  leadsto-map4' = do {
    (r, passed) ← A'.pw-algo-map2;
    ASSERT (finite (dom passed));
    passed ← ran-of-map passed;
    (r, -) ← nfoldli passed (λ(b,-). ¬b)
      (λ passed' (-,acc).
        do {
          passed' ← SPEC (λl. set l = passed');
          nfoldli passed' (λ(b,-). ¬b)
            (λv' (-,passed).
             if P v' ∧ Q v' then has-cycle-map v' passed else RETURN (False,
passed)
            )
            (False, acc)
```

```
      }
    )
    (False, Map.empty);
  RETURN r
}

lemma leadsto-map4'-ref:
  leadsto-map4' ≤ ⇓ Id leadsto-map3'
  unfolding leadsto-map4'-def leadsto-map3'-def FOREACHcd-def
  apply (subst (2) pw-algo-map2-copy-def[symmetric])
  apply (refine-rcg ran-of-map-ref)
    apply refine-dref-type
    apply (simp add: br-def; fail)
   apply (simp add: br-def; fail)
  apply (rule rhs-step-bind-SPEC)
  by (auto simp: br-def)

lemma leadsto-map4'-correct:
  leadsto-map4' ≤ leadsto-spec-alt
proof −
  note leadsto-map4'-ref
  also note leadsto-map3'-ref
  also note leadsto'-correct
  also note leadsto-spec-leadsto-spec-alt
  finally show ?thesis .
qed

end

end
```

7.3 Imperative Implementation

```
theory Leadsto-Impl
  imports Leadsto-Map Unified-PW-Impl Liveness-Subsumption-Impl
begin

definition
  list-of-set S = SPEC (λl. set l = S)

lemma lso-id-hnr:
  (return o id, list-of-set) ∈ (lso-assn A)^d →_a list-assn A
 unfolding list-of-set-def lso-assn-def hr-comp-def br-def by sepref-to-hoare
sep-auto
```

sepref-register *hm.op-hms-empty*

context *Worklist-Map2-Impl*
begin

sepref-thm *pw-algo-map2-impl* **is**
uncurry0 (*pw-algo-map2*) ::
unit-assn^k $\rightarrow_a$ *bool-assn* $\times_a$ (*hm.hms-assn'* *K* (*lso-assn A*))
unfolding *pw-algo-map2-def add-pw'-map2-alt-def PR-CONST-def TRACE'-def*[*symmetric*]
supply [[*goals-limit* = *1*]]
supply *conv-to-is-Nil*[*simp*]
unfolding *fold-lso-bex*
unfolding *take-from-list-alt-def*
apply (*rewrite* **in** {a_0} *lso-fold-custom-empty*)
unfolding *hm.hms-fold-custom-empty*
apply (*rewrite* **in** [a_0] *HOL-list.fold-custom-empty*)
apply (*rewrite* **in** {} *lso-fold-custom-empty*)
unfolding *F-split*
by *sepref*

end

locale *Leadsto-Search-Space-Key-Impl* =
Leadsto-Search-Space-Key a_0 *F - empty - E key F' P Q succs-Q succs1* +
liveness: *Liveness-Search-Space-Key-Impl* a_0 *F - V succs-Q* λ *x y. E x y* $\wedge$ $\neg$ *empty y* $\wedge$ *Q y*
- key A succsi a_0*i Lei keyi copyi*
for *key* :: $'v \Rightarrow 'k$
and a_0 *F F' copyi P Q V empty succs-Q succs1 E A succsi* a_0*i Lei keyi* +
fixes *succs1i* **and** *emptyi* **and** *Pi Qi* **and** *tracei*
assumes *succs1-impl*: (*succs1i*, (*RETURN* $\circ\circ$ *PR-CONST*) *succs1*) $\in$ $A^k \rightarrow_a$ *list-assn A*
and *empty-impl*:
(*emptyi*,*RETURN o PR-CONST empty*) $\in$ $A^k \rightarrow_a$ *bool-assn*
assumes [*sepref-fr-rules*]:
(*Pi*,*RETURN o PR-CONST P*) $\in$ $A^k \rightarrow_a$ *bool-assn* (*Qi*,*RETURN o PR-CONST Q*) $\in$ $A^k \rightarrow_a$ *bool-assn*
assumes *trace-impl*:
(*uncurry tracei*,*uncurry* (λ(- :: *string*) -. *RETURN* ())) $\in$ *id-assn*^{k} $*_a$ $A^k \rightarrow_a$ *id-assn*
begin

sublocale *Worklist-Map2-Impl - - λ -. False - succs1 - - λ-. False - succs1i*
-
λ-. return False Lei
apply *standard*
unfolding *A′.trace-def*
apply (*rule liveness.refinements succs1-impl*)
subgoal
by *sepref-to-hoare sep-auto*
by (*rule liveness.refinements succs1-impl empty-impl*
liveness.pure-K liveness.left-unique-K liveness.right-unique-K trace-impl)+

sepref-register *pw-algo-map2-copy*
sepref-register *PR-CONST P PR-CONST Q*

lemmas [*sepref-fr-rules*] =
lso-id-hnr
ran-of-map-impl.refine[*OF pure-K left-unique-K right-unique-K*]

lemma *pw-algo-map2-copy-fold*:
PR-CONST pw-algo-map2-copy = *A′.pw-algo-map2*
unfolding *pw-algo-map2-copy-def* **by** *simp*

lemmas [*sepref-fr-rules*] = *pw-algo-map2-impl.refine-raw*[*folded pw-algo-map2-copy-fold*]

definition *has-cycle-map-copy* ≡ *has-cycle-map*

lemma *has-cycle-map-copy-fold*:
PR-CONST has-cycle-map-copy = *has-cycle-map*
unfolding *has-cycle-map-copy-def* **by** *simp*

sepref-register *has-cycle-map-copy*

lemma *has-cycle-map-fold*:
has-cycle-map = *liveness.dfs-map′*
unfolding *has-cycle-map-def liveness.dfs-map′-def*
by (*subst Liveness-Search-Space-Key-Defs.dfs-map-alt-def*) *standard*

lemmas [*sepref-fr-rules*] =
liveness.dfs-map′-impl.refine-raw[*folded has-cycle-map-fold*, *folded has-cycle-map-copy-fold*]

sepref-thm *leadsto-impl* **is**
uncurry0 leadsto-map4′ :: *unit-assn*^{k} →$_a$ *bool-assn*
unfolding *leadsto-map4′-def*
apply (*rewrite* **in** *P PR-CONST-def*[*symmetric*])

apply (*rewrite* **in** *Q PR-CONST-def*[*symmetric*])
unfolding *pw-algo-map2-copy-def*[*symmetric*]
unfolding *has-cycle-map-copy-def*[*symmetric*]
unfolding *hm.hms-fold-custom-empty*
unfolding *list-of-set-def*[*symmetric*]
by *sepref*

concrete-definition (**in** −) *leadsto-impl*
uses *Leadsto-Search-Space-Key-Impl.leadsto-impl.refine-raw* **is** (*uncurry0 ?f*,-)∈-

lemma *leadsto-impl-hnr*:
(*uncurry0* (
leadsto-impl copyi succsi a_0*i Lei keyi succs1i emptyi Pi Qi tracei*
),
uncurry0 leadsto-spec-alt
) ∈ *unit-assn*^{k} →$_a$ *bool-assn* **if** *V* a_0
unfolding *leadsto-spec-alt-def*
using *leadsto-impl.refine*[
OF Leadsto-Search-Space-Key-Impl-axioms,
FCOMP leadsto-map4'-correct[*unfolded leadsto-spec-alt-def*, *THEN Id-SPEC-refine*, *THEN nres-relI*]
] .

end

end

References

[1] G. Behrmann, K. G. Larsen, and J. I. Rasmussen. Beyond Liveness: Efficient Parameter Synthesis for Time Bounded Liveness. In *FORMATS*, volume 3829 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2005.

[2] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[3] S. Wimmer. *Trustworthy Verification of Realtime Systems*. PhD thesis, Technical University of Munich, Germany, 2020.

[4] S. Wimmer and P. Lammich. Verified Model Checking of Timed Automata. In *TACAS (1)*, volume 10805 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 2018.