

Worklist Algorithms

Simon Wimmer, Peter Lammich

August 21, 2024

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	3
1.1	Search Spaces	3
1.2	Miscellaneous	8
2	Subsumption Graphs	11
2.1	Preliminaries	11
2.2	Definitions	13
2.3	Reachability	18
2.4	Liveness	28
2.5	Appendix	28
2.6	Old Material	35
3	Unified Passed-Waiting-List	38
3.1	Utilities	38
3.2	Generalized Worklist Algorithm	39
3.3	Towards an Implementation of the Unified Passed-Waiting List	51
3.4	Heap Hash Map	61
3.5	Imperative Implementation	69

4	Generic Worklist Algorithm With Subsumption	75
4.1	Utilities	75
4.2	Standard Worklist Algorithm	76
4.3	From Multisets to Lists	86
4.4	Towards an Implementation	89
4.5	Implementation on Lists	96
5	Checking Always Properties	99
5.1	Abstract Implementation	99
5.2	Implementation on Maps	110
5.3	Imperative Implementation	119
6	A Next-Key Operation for Hashmaps	127
6.1	Definition and Key Properties	127
6.2	Computing the Range of a Map	131
7	Checking Leads-To Properties	134
7.1	Abstract Implementation	134
7.2	Implementation on Maps	143
7.3	Imperative Implementation	161

1 Preliminaries

theory *Worklist-Locales*

imports *Refine-Imperative-HOL.Sepref Collections.HashCode Probabilistic-Timed-Automata.Graphs*

HOL-ex.Sketch-and-Explore

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 $E :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ — Step relation

and $a_0 :: 'a$ — Start state

and $F :: 'a \Rightarrow \text{bool}$ — Final states

and $\text{subsumes} :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ (**infix** $\preceq 50$) — Subsumption preorder

begin

sublocale *Graph-Start-Defs* $E a_0$.

definition *subsumes-strictly* (**infix** $\prec 50$) **where**

subsumes-strictly $x y = (x \preceq y \wedge \neg y \preceq x)$

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

definition *F-reachable* $\equiv \exists a. \text{reachable } a \wedge F a$

end

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

fixes $V :: 'a \Rightarrow \text{bool}$

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

fixes *empty* $:: 'a \Rightarrow \text{bool}$

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*: $a \preceq a' \implies F a \implies F a'$
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*]: $a \preceq b \implies b \preceq c \implies a \preceq c$

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

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

sublocale *search-space*:
 $\text{Search-Space-Nodes } \lambda x y. E x y \wedge \neg \text{empty } y \ a_0 F (\preceq) \lambda v. V v \wedge \neg$
 $\text{empty } v$
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*]: $a \preceq b \implies b \preceq c \implies a \preceq c$

```

assumes mono:
   $a \preceq b \implies E a a' \implies \text{reachable } a \implies \text{reachable } b \implies \neg \text{empty } a \implies$ 
 $\exists b'. E b b' \wedge a' \preceq b'$ 
  and empty-subsumes:  $\text{empty } a \implies a \preceq a'$ 
  and empty-mono:  $\neg \text{empty } a \implies a \preceq b \implies \neg \text{empty } b$ 
  and empty-E:  $\text{reachable } x \implies \text{empty } x \implies E x x' \implies \text{empty } x'$ 
  and F-mono:  $a \preceq a' \implies F a \implies F a'$ 
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:  $\text{finite } \{a. \text{reachable } a \wedge \neg \text{empty } a\}$ 

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

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

locale Search-Space' = Search-Space +
  assumes final-non-empty:  $F a \implies \neg \text{empty } a$ 

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

locale Search-Space''-Defs = Search-Space-Defs-Empty +
  fixes subsumes' ::  $'a \Rightarrow 'a \Rightarrow \text{bool}$  (infix  $\trianglelefteq$  50) — Subsumption preorder

locale Search-Space''-pre = Search-Space''-Defs +
  assumes empty-subsumes':  $\neg \text{empty } a \implies a \preceq b \iff a \trianglelefteq b$ 

locale Search-Space''-start = Search-Space''-pre +
  assumes start-non-empty [simp]:  $\neg \text{empty } a_0$ 

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 **for** *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*]: $a \leq b \implies \text{key } a = \text{key } b$
locale *Worklist0-Defs* = *Search-Space-Defs* +
fixes *succs* :: '*a* \Rightarrow '*a list*
locale *Worklist0* = *Worklist0-Defs* + *Search-Space* +
assumes *succs-correct*: $\text{reachable } a \implies \text{set } (\text{succs } a) = \text{Collect } (E \ a)$
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*: $F \ a \longleftrightarrow \neg \text{empty } a \wedge F' \ a$
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* \subseteq *Search-Space''-finite* ..

locale *Worklist4-Impl-Defs* = *Worklist3-Defs* +

fixes *A* :: 'a \Rightarrow 'ai \Rightarrow *assn*

fixes *sucsci* :: 'ai \Rightarrow 'ai *list Heap*

fixes *a₀i* :: 'ai *Heap*

fixes *Fi* :: 'ai \Rightarrow *bool Heap*

fixes *Lei* :: 'ai \Rightarrow 'ai \Rightarrow *bool Heap*

fixes *emptyi* :: 'ai \Rightarrow *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 a₀i*, *uncurry0 (RETURN (PR-CONST a₀)))*) \in *unit-assn^k \rightarrow_a A*

assumes [*sepref-fr-rules*]: (*Fi, RETURN o PR-CONST F'*) \in *A^k \rightarrow_a bool-assn*

assumes [*sepref-fr-rules*]: (*uncurry Lei, uncurry (RETURN oo PR-CONST (\triangleleft)))*) \in *A^k *_a A^k \rightarrow_a bool-assn*

assumes [*sepref-fr-rules*]: (*sucsci, RETURN o PR-CONST succs*) \in *A^k \rightarrow_a list-assn A*

assumes [*sepref-fr-rules*]: (*emptyi, RETURN o PR-CONST empty*) \in *A^k \rightarrow_a bool-assn*

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

sublocale *Worklist4-Impl-finite-strict* \subseteq *Search-Space''-finite-strict* ..

locale *Worklist-Map2-Impl-Defs* =

Worklist4-Impl-Defs - - - - - *A* + *Worklist-Map2-Defs a₀* - - - - - *key*

for *A* :: 'a \Rightarrow 'ai :: {*heap*} \Rightarrow - **and** *key* :: 'a \Rightarrow 'k +

fixes *keyi* :: 'ai \Rightarrow 'ki :: {*hashable, heap*} *Heap*

fixes *copyi* :: 'ai \Rightarrow 'ai *Heap*

fixes *tracei* :: *string* \Rightarrow 'ai \Rightarrow *unit Heap*

end

theory *Worklist-Common*

imports *Worklist-Locales*

begin

lemma *list-ex-foldli*:
list-ex P $xs = foldli$ xs *Not* $(\lambda x y. P x \vee y)$ *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* $\{a'. E a a' \wedge \neg empty a'\}$
proof –
 have $\{a'. E a a' \wedge \neg empty a'\} \subseteq \{a'. reachable a' \wedge \neg empty a'\}$
 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 =
 $\{(m, s).$
 $\bigcup (ran\ m) = s \wedge (\forall k. \forall x. m\ k = Some\ x \longrightarrow (\forall v \in x. key\ v = k))$
 \wedge
 $finite\ (dom\ m) \wedge (\forall k\ S. m\ k = Some\ S \longrightarrow finite\ S)$
 $\}$
end

1.2 Miscellaneous

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

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

lemma *filter-mset-eq-empty-iff*:
 $\{\#x \in \# A. P\ x\ \#\} = \{\#\} \longleftrightarrow (\forall x \in set\text{-}mset\ A. \neg P\ x)$
by (*auto simp: mset-eq-empty-iff*)

lemma *mset-remove-member*:

```

     $x \in\# A - B$  if  $x \in\# A$   $x \notin\# B$ 
    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  $\Rightarrow$  unit where
  write-msg m = ()

code-printing code-module Tracing  $\rightarrow$  (SML)
<
structure Tracing : sig
  val count-up : unit  $\rightarrow$  unit
  val get-count : unit  $\rightarrow$  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  $\rightarrow$  unit
  val get-count : unit  $\rightarrow$  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  $\rightarrow$  (SML) (fn x => Tracing.count'-up ()) -
    and (OCaml) (fun x  $\rightarrow$  Tracing.count'-up ()) -

```

definition *trace where*

$trace\ m\ x = (let\ a = write\text{-}msg\ m\ in\ x)$

lemma *trace-alt-def[simp]:*

$trace\ m\ x = (\lambda\ -. x)\ (write\text{-}msg\ x)$

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*]:

$(trace,\ trace) \in \langle Id, \langle Id, Id \rangle fun\text{-}rel \rangle fun\text{-}rel$

by *simp*

sepref-definition *TRACE-impl is*

$TRACE :: id\text{-}assn^k \rightarrow_a\ unit\text{-}assn$

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*]:

$(trace',\ trace') \in \langle Id, Id \rangle fun\text{-}rel$

by *simp*

sepref-definition *TRACE'-impl is*

$uncurry0\ TRACE' :: unit\text{-}assn^k \rightarrow_a\ unit\text{-}assn$

unfolding *TRACE'-alt-def*

by *sepref*

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

end

2 Subsumption Graphs

theory *Worklist-Algorithms-Subsumption-Graphs*

imports

Probabilistic-Timed-Automata.Graphs

Probabilistic-Timed-Automata.More-List

begin

2.1 Preliminaries

Transitive Closure context

fixes $R :: 'a \Rightarrow 'a \Rightarrow \text{bool}$

assumes $R\text{-trans}[\text{intro}]: \bigwedge x y z. R x y \Longrightarrow R y z \Longrightarrow R x z$

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 $P :: 'a \Rightarrow \text{bool}$ **and** $R :: 'a \Rightarrow 'a \Rightarrow \text{bool}$

assumes *reachable-finite*: $\text{finite } \{x. R^{**} a x\}$

assumes $R\text{-irrefl}$: $\bigwedge x. \neg R x x$ **and** $R\text{-trans}[\text{intro}]: \bigwedge x y z. R x y \Longrightarrow R y z \Longrightarrow R x z$

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  $\langle R^{**} a y \rangle$  show ?thesis by auto
  next
    assume  $\exists z. R y z$ 
    then obtain z where  $R y z$  by safe
    let  $?T = \{y. R^{**} z y\}$ 
    from  $\langle R y z \rangle \langle R^{**} a y \rangle$  have  $\neg R^{**} z a$ 
      by (auto simp: R-irrefl dest!: rtranclp-transitive-compress2[of R,
rotated])
    then have  $a \notin ?T$  by auto
    moreover have  $?T \subseteq ?S$ 
      using  $\langle R^{**} a y \rangle \langle R y z \rangle$  by auto
    ultimately have  $?T \subset ?S$ 
      by auto
    have  $P x \vee \exists x (R x)$  if  $R^{**} z x$  for x
      using that  $\langle R y z \rangle \langle R^{**} a y \rangle$  by (auto intro!: psubset.prems)
      from psubset.hyps(2)[OF <?T < ?S> this] psubset.prems  $\langle R y z \rangle$ 
 $\langle R^{**} a y \rangle$  obtain w
      where  $R^{**} z w$   $P w$  by auto
      with  $\langle R^{**} a y \rangle \langle R y z \rangle$  have  $R^{**} a w$  by auto
      with  $\langle P w \rangle$  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  $P Q :: 'a \Rightarrow bool$ 
  assumes Q-finite: finite  $\{x. Q x\}$  and Q-witness: Q a
  assumes R-irrefl:  $\bigwedge x. \neg R x x$  and R-trans[intro]:  $\bigwedge x y z. R x y \implies$ 
R y z  $\implies R x z$ 
  assumes step:  $\bigwedge x. Q x \implies P x \vee (\exists y. R x y \wedge Q y)$ 

```

```

shows  $\exists x. P x \wedge Q x \wedge R^{**} a x$ 
proof –
  let  $?R = \lambda x y. R x y \wedge Q x \wedge Q y$ 
  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  $\langle Q a \rangle$  by (auto elim: rtranclp.cases)
  have  $\{x. ?R^{**} a x\} \subseteq \{x. Q x\}$  by auto
  with  $\langle \text{finite } \rightarrow \rangle$  have finite  $\{x. ?R^{**} a x\}$  by – (rule finite-subset)
  then have  $\exists x. P x \wedge ?R^{**} a x$ 
  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 ::  $'a \Rightarrow 'a \Rightarrow \text{bool}$  (infix  $\preceq 50$ ) and less (infix
 $\prec 50$ ) +
  fixes  $E :: 'a \Rightarrow 'a \Rightarrow \text{bool}$  — The full edge set
begin

```

```

sublocale Graph-Defs  $E$  .

```

```

end

```

```

locale Subsumption-Graph-Pre-Nodes-Defs = Subsumption-Graph-Pre-Defs
+
  fixes  $V :: 'a \Rightarrow \text{bool}$ 
begin

```

```

sublocale Subgraph-Node-Defs-Notation .

```

```

end

```

```

locale Subsumption-Graph-Defs = Subsumption-Graph-Pre-Defs +
  fixes  $s_0 :: 'a$  — Start state
  fixes  $RE :: 'a \Rightarrow 'a \Rightarrow \text{bool}$  — 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* $\lambda x y. RE x y \vee (x \prec y \wedge G.reachable y)$
 s_0 .

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

notation RE $(- \rightarrow_G - [100, 100] 40)$

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

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

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

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

end

locale *Subsumption-Graph-Pre* = *Subsumption-Graph-Defs* + *preorder less-eq less* +

assumes *mono*:

$a \preceq b \implies E a a' \implies reachable a \implies reachable b \implies \exists b'. E b b' \wedge a' \preceq b'$

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*:

$a \preceq b \implies a \rightarrow a' \implies V a \implies V b \implies \exists b'. b \rightarrow b' \wedge a' \preceq b'$

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*:

$a \preceq b \implies E a a' \implies \text{reachable } a \vee G.\text{reachable } a \implies \text{reachable } b \vee G.\text{reachable } b$

$\implies \exists b'. E b b' \wedge a' \preceq b'$

assumes *reachability-compatible*:

$\forall s. G.\text{reachable } s \longrightarrow (\forall s'. E s s' \longrightarrow RE s s') \vee (\exists t. s \prec t \wedge G.\text{reachable } t)$

assumes *finite-reachable*: *finite* $\{a. G.\text{reachable } a\}$

locale *Reachability-Compatible-Subsumption-Graph* =

Subsumption-Graph-Defs + *Subsumption-Graph-Pre* +

assumes *reachability-compatible*:

$\forall s. G.\text{reachable } s \longrightarrow (\forall s'. E s s' \longrightarrow RE s s') \vee (\exists t. s \prec t \wedge G.\text{reachable } t)$

assumes *subgraph*: $\forall s s'. RE s s' \longrightarrow E s s'$

assumes *finite-reachable*: *finite* $\{a. G.\text{reachable } a\}$

locale *Subsumption-Graph-View-Defs* = *Subsumption-Graph-Defs* +

fixes *SE* :: $'a \Rightarrow 'a \Rightarrow \text{bool}$ — *Subsumption edges*

and *covered* :: $'a \Rightarrow \text{bool}$

locale *Reachability-Compatible-Subsumption-Graph-View* =

Subsumption-Graph-View-Defs + *Subsumption-Graph-Pre* +

assumes *reachability-compatible*:

$\forall s. G.\text{reachable } s \longrightarrow$

$(\text{if } \text{covered } s \text{ then } (\exists t. SE s t \wedge G.\text{reachable } t) \text{ else } (\forall s'. E s s' \longrightarrow RE s s'))$

assumes *subsumption*: $\forall s s'. SE s s' \longrightarrow s \prec s'$

assumes *subgraph*: $\forall s s'. RE s s' \longrightarrow E s s'$

assumes *finite-reachable*: *finite* $\{a. G.\text{reachable } a\}$

begin

sublocale *Reachability-Compatible-Subsumption-Graph* $(\preceq) (\prec) E s_0 RE$

proof *unfold-locales*

have $(\forall s s'. E s s' \longrightarrow RE s s') \vee (\exists t. s \prec t \wedge G.\text{reachable } t)$ **if** *G.reachable s* **for** *s*

using *that reachability-compatible subsumption by (cases covered s; fast-force)*

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 *s₀* :: '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 s₀* .

sublocale *G: Graph-Start-Defs* *RE s₀* .

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  $P\text{-}s_0: P\ s_0$ 
assumes  $Q\text{-}post: RE\ a\ b \implies Q\ b$ 
assumes  $Q\text{-}s_0: Q\ s_0$ 
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  $[simp] = close\text{-}def$ 
supply  $[intro] = P\text{-}pre\ P\text{-}post\ Q\text{-}post$ 
proof (standard, goal-cases)
  case prems: (1 a b a')
  then obtain  $x\ y$  where  $[simp]: x \rightarrow y\ a = closure\ x\ a' = closure\ y$ 
    by auto
  then have  $P\ x\ P\ y$ 
    by blast+
  from prems(4)  $P\text{-}s_0$  obtain  $x'$  where  $[simp]: b = closure\ x'\ P\ x'$ 
    unfolding Graph-Start-Defs.reachable-def by cases auto
  from  $mono[OF\ \prec\ \preceq\ \rightarrow][simplified]\ \langle x \rightarrow y\rangle\ \langle P\ x\rangle\ \langle P\ x'\rangle$  obtain  $b'$  where
     $x' \rightarrow b'\ closure\ y \preceq closure\ b'$ 
    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  $Graph\text{-}Start\text{-}Defs.reachable\ (close\ RE)\ (closure\ s_0)\ x$ 
    then obtain  $x'$  where  $[simp]: x = closure\ x'\ Q\ x'\ P\ x'$ 
      using  $Q\text{-}s_0\ P\text{-}s_0\ subgraph$  unfolding Graph-Start-Defs.reachable-def
by cases auto
    have  $(\forall\ s'. close\ E\ x\ s' \longrightarrow close\ RE\ x\ s')$ 
       $\vee (\exists\ t. x \prec t \wedge Graph\text{-}Start\text{-}Defs.reachable\ (close\ RE)\ (closure\ s_0)\ t)$ 
    proof (cases covered x')
      case True
      with reachability-compatible  $\langle Q\ x'\rangle$  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  $\langle Q \ x' \rangle$  have  $\forall s'. x' \rightarrow s' \longrightarrow RE \ x' \ s'$ 
      by auto
    then show ?thesis
      unfolding close-def using closure-eq[OF - - -  $\langle P \ x' \rangle$ ] 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  $\{a. \text{Graph-Start-Defs.reachable } (close \ RE) \ (closure \ s_0) \ a\} \subseteq \text{closure}$ 
    ‘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  $\Rightarrow$  bool — Final states
  assumes F-mono[intro]:  $F \ a \Longrightarrow a \preceq b \Longrightarrow F \ b$ 

```

```

locale Liveness-Compatible-Subsumption-Graph = Reachability-Compatible-Subsumption-Graph-Final
+

```

```

  assumes no-subsumption-cycle:
     $G'.reachable \ x \Longrightarrow x \rightarrow_{G^{+'}} x \Longrightarrow x \rightarrow_{G^+} x$ 

```

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  $\langle G.reachable s \rangle$  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  $\langle ((\prec)^{**}) s x \rangle$  have  $s \prec x \vee s = x$ 
  by induction auto
ultimately show ?thesis

```

by *auto*
qed

lemma *reachable-has-surrogate*:

$\exists t. G.\text{reachable } t \wedge s \preceq t \wedge (\forall s'. E t s' \longrightarrow RE t s')$ **if** *reachable s*
using *that*

proof *induction*

case *start*

have $G.\text{reachable } s_0$

by *auto*

then show *?case*

by (*rule G-reachable-has-surrogate*)

next

case (*step a b*)

then obtain t **where** $*$: $G.\text{reachable } t \wedge a \preceq t \wedge (\forall s'. t \rightarrow s' \longrightarrow t \rightarrow_G s')$

by *auto*

from $\text{mono}[OF \langle a \preceq t \rangle \langle a \rightarrow b \rangle] \langle \text{reachable } a \rangle \langle G.\text{reachable } t \rangle$ **obtain** b'

where

$t \rightarrow b' \wedge b \preceq b'$

by *auto*

with $G.\text{reachable-has-surrogate}[\text{of } b']$ *** show** *?case*

by (*auto intro: preorder-intros G-reachable-step*)

qed

context

fixes $F :: 'a \Rightarrow \text{bool}$ — Final states

assumes $F\text{-mono}[\text{intro}]$: $F a \Longrightarrow a \preceq b \Longrightarrow F b$

begin

corollary *reachability-correct*:

$\nexists s'. \text{reachable } s' \wedge F s'$ **if** $\nexists s'. G.\text{reachable } s' \wedge F s'$
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*:

$\exists t. G.\text{reachable } t \wedge s \preceq t \wedge (\forall s'. E t s' \longrightarrow RE t s')$ **if** *G.reachable s*

proof –

note [*intro*] = *preorder-intros*

from *finite-reachable* $\langle G.\text{reachable } s \rangle$ **obtain** *x* **where**

$\forall s'. E x s' \longrightarrow RE x s' G.\text{reachable } x ((\prec)^{**}) s x$

apply *atomize-elim*

apply (*induction rule: rtranclp-ev-induct2*)

using *reachability-compatible* **by** *auto*

moreover from $\langle ((\prec)^{**}) s x \rangle$ **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 $\langle G.reachable s \rangle$ **have** $\langle G.reachable s \rangle$ **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 $\langle ((\prec)^{**}) s x \rangle$ **have** $s \prec x \vee s = x$

by *induction auto*

then show *?thesis*

proof

assume $s \prec x$

with *real-edges* $\langle G.reachable x \rangle$ **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* $\langle a \preceq a' \rangle \langle E a b \rangle \langle reachable a \rangle$] $\langle G.reachable a' \rangle$ **obtain** b' **where** $E a' b' b \preceq b'$

by *auto*

from *reachable-has-surrogate*[*OF* $\langle G.reachable a' \rangle$] **obtain** a''

where $a' \preceq a''$ $G.reachable a''$ **and** $*$: $\forall s'. E a'' s' \longrightarrow RE a'' s'$

by *auto*

from *mono*[*OF* $\langle a' \preceq a'' \rangle \langle E a' b' \rangle$] $\langle G.reachable a' \rangle \langle G.reachable a'' \rangle$ **obtain** b'' **where**

$E a'' b'' b' \preceq b''$

by *auto*

with $*$ $\langle a' \preceq a'' \rangle \langle b \preceq b' \rangle \langle G.reachable a'' \rangle$ **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 ≤ a'> <E a b> <reachable a>]* *<G'.reachable a'>* **obtain**
b' **where**
 $b \preceq b' a' \rightarrow b'$
by *auto*
from *reachable-has-surrogate'[of a'] <G'.reachable a'>* **obtain** *a''* **where**
 $*$:
 $a' \preceq a'' a' \rightarrow_{G*'} a'' \forall s'. a'' \rightarrow s' \longrightarrow a'' \rightarrow_G s'$
by *auto*
with *<G'.reachable a'>* **have** *G'.reachable a''*
by *blast*
with *mono[OF <a' ≤ a''> <E a' b'>]* *<G'.reachable a'>* **obtain** *b''* **where**
 $b' \preceq b'' a'' \rightarrow b''$
by *auto*
with $*$ *<b ≤ b'> <b' ≤ b''> <G'.reachable a''>* **show** *?thesis*
by (*auto simp: G'.reaches1-reaches-iff2*)
qed

theorem *reachability-complete'*:

$\exists s'. s \preceq s' \wedge G.\text{reachable } s'$ **if** $a \rightarrow^* s$ *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 ≤ s'>]*

obtain *s'' t'* **where**

$s' \preceq s'' t \preceq t' s'' \rightarrow_G t' G.\text{reachable } s''$

by *atomize-elim*

with *<G.reachable s'>* **show** *?case*

by *auto*

qed

corollary *reachability-complete*:

$\exists s'. s \preceq s' \wedge G.\text{reachable } s'$ **if** *reachable s*

using *reachability-complete'*[of s_0 s] that **unfolding** *reachable-def* by *auto*

corollary *reachability-correct*:

$(\exists s'. s \preceq s' \wedge \text{reachable } s') \longleftrightarrow (\exists s'. s \preceq s' \wedge G.\text{reachable } s')$
 by (*blast dest: reachability-complete intro: preorder-intros*)

lemma *steps-G'-steps*:

$\exists ys\ ns.\ \text{list-all2 } (\preceq)\ xs\ (\text{nths } ys\ ns) \wedge G'.\text{steps } (b \# ys)$ **if**
 $\text{steps } (a \# xs)\ \text{reachable } a\ a \preceq b\ G'.\text{reachable } b$
 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* $\langle \text{reachable } x \rangle \langle E\ x\ y \rangle - \langle x \preceq b \rangle$] $\langle G'.\text{reachable } b \rangle$ **obtain** b' **where**

$y \preceq b' \ b \rightarrow_{G^{+'}} b'$

by *auto*

with $\langle \text{reachable } x \rangle$ *Cons.hyps*(1) *Cons.prem*s(3) **obtain** $ys\ ns$ **where**

list-all2 $(\preceq)\ xs\ (\text{nths } ys\ ns)\ G'.\text{steps } (b' \# ys)$

by *atomize-elim* (*blast intro: Cons.hyps*(3)[*OF* - $\langle y \preceq b' \rangle$] *intro: graphI-aggressive*)

from $\langle b \rightarrow_{G^{+'}} b' \rangle$ *this*(2) **obtain as** **where**

$G'.\text{steps } (b \# as\ @\ b' \# ys)$

by (*fastforce intro: G'.graphI-aggressive1*)

with $\langle y \preceq b' \rangle$ **show** *?case*

apply (*inst-existentials as @ b' # ys {length as} \cup {n + length as + 1*
 $| n.\ n \in ns\}$)

subgoal

apply (*subst nth*s-split, *force*)

apply (*subst nth*s-nth, (*simp; fail*))

apply *simp*

apply (*subst nth*s-shift, *force*)

subgoal *premises* *prems*

proof –

have

$\{x - \text{length } as \mid x.\ x \in \{\text{Suc } (n + \text{length } as) \mid n.\ n \in ns\}\} = \{n + 1 \mid n.\ n \in ns\}$

by *force*

with $\langle \text{list-all2} - - \rightarrow \rangle$ **show** *?thesis*

by (*simp add: nth*s-Cons)

qed

done

by *assumption*

qed

lemma *cycle-G'-cycle''*:

assumes *steps* ($s_0 \# ws @ x \# xs @ [x]$)

shows $\exists x' xs' ys'. x \preceq x' \wedge G'.steps (s_0 \# xs' @ x' \# ys' @ [x'])$

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 $\langle steps\ ?xs \rangle$ **show** *?thesis*

by (*fastforce intro: graphI-aggressive1*)

qed

from *steps-G'-steps[OF this, of s₀]* **obtain** *ys ns* **where** *ys*:

list-all2 (\preceq) ($ws @ x \# concat (replicate\ ?n\ (xs @ [x]))$) (*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 (\preceq) ($concat (replicate\ ?n\ (xs @ [x]))$) (*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  $G'.steps (s_0 \# ys_4 @ ys_6 @ [z])$ 
  by (auto intro: G'.graphI-aggressive2)
ultimately show ?thesis
  by (inst-existentials ys_4 @ ys_6) auto
qed
by force
done
let  $?ys = filter ((\preceq) x) ys'$ 
have  $length ?ys \geq ?n$ 
  using list-all2-replicate-elem-filter[OF ys'(3), of x]
  using filter-nths-length[of ((\preceq) x) ys' ns']
  by auto
from  $\langle G'.steps (s_0 \# ws' @ [x']) \rangle$  have  $G'.reachable x'$ 
  by  $-$  (rule G'.reachable-reaches, auto)
have  $set ?ys \subseteq set ys'$ 
  by auto
also have  $\dots \subseteq \{x. G'.reachable x\}$ 
  using  $\langle G'.steps (x' \# -) \rangle \langle G'.reachable x' \rangle$ 
  by clarsimp (rule G'.reachable-steps-elem[rotated], assumption, auto)
finally have  $\neg distinct ?ys$ 
  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  $as\ bs\ cs$  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  $\langle G'.steps (x' \# -) \rangle \langle ys' = - \rangle$  show ?thesis
    by (force intro: G'.graphI-aggressive2)
qed
moreover have  $G'.steps (s_0 \# ws' @ x' \# as' @ [y])$ 
proof  $-$ 
  from  $\langle G'.steps (x' \# ys') \rangle \langle ys' = - \rangle$  have  $G'.steps (x' \# as' @ [y])$ 
    by (force intro: G'.graphI-aggressive2)

```

with $\langle G'.steps (s_0 \# ws' @ [x']) \rangle$ **show** *?thesis*
 by (*fastforce intro: G'.graphI-aggressive1*)
qed
moreover from $\langle ?ys = - \rangle$ **have** $x \preceq y$
proof –
 from $\langle ?ys = - \rangle$ **have** $y \in set ?ys$ **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 $steps (s_0 \# ws @ x \# xs @ [x])$
shows $\exists y ys. x \preceq y \wedge G'.steps (y \# ys @ [y]) \wedge G'.reachable y$
proof –
from $cycle-G'-cycle''[OF\ assms]$ **obtain** $x' xs' ys'$ **where**
 $x \preceq x' G'.steps (s_0 \# xs' @ x' \# ys' @ [x'])$
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 $reachable x x \rightarrow^+ x$
shows $\exists y ys. x \preceq y \wedge G'.reachable y \wedge y \rightarrow_G^{+'} y$
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** $steps\ ws\ hd\ ws =$
 $s_0\ last\ ws = x$
by *auto*
with $*$ **obtain** us **where** $steps (s_0 \# (us @ xs) @ x \# xs @ [x])$
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*:
 $\exists s'. s \preceq s' \wedge G'.reachable s'$ **if** $G'.reachable s$
using *reachability-complete* **that by** *auto*

end

end

corollary (in *Reachability-Compatible-Subsumption-Graph-Final*) *reachability-correct*:

$$(\exists s'. \text{reachable } s' \wedge F s') \longleftrightarrow (\exists s'. G.\text{reachable } s' \wedge F s')$$

using *reachability-complete* by *blast*

2.4 Liveness

theorem (in *Liveness-Compatible-Subsumption-Graph*) *cycle-iff*:

$$(\exists x. x \rightarrow^+ x \wedge \text{reachable } x \wedge F x) \longleftrightarrow (\exists x. x \rightarrow_{G^+} x \wedge G.\text{reachable } x \wedge F x)$$

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 $G'.\text{steps } (x \# xs) \ x \preceq y \ V \ x \ V \ y$

shows $\exists \ ys. \ G'.\text{steps } (y \# ys) \wedge \text{list-all2 } (\preceq) \ xs \ ys$

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* $\langle x \preceq x' \rangle \langle x \rightarrow y \rangle \text{Cons.prem}$] obtain *y'* where $x' \rightarrow y'$
 $y \preceq y'$

by *auto*

with *Cons.hyps*(\exists)[*OF* $\langle y \preceq y' \rangle \langle x \rightarrow y \rangle \text{Cons.prem}$] obtain *ys* where
 $G'.\text{steps } (y' \# ys) \ \text{list-all2 } (\preceq) \ xs \ ys$

by *auto*

with $\langle x' \rightarrow y' \rangle \langle y \preceq y' \rangle$ show *?case*

by *auto*
qed

lemma *steps-append-subsumption*:

assumes $G'.steps(x \# xs) G'.steps(y \# ys) y \preceq last(x \# xs) \vee x \vee y$
shows $\exists ys'. G'.steps(x \# xs @ ys') \wedge list-all2(\preceq) ys ys'$

proof –

from *assms* **have** $V(last(x \# xs))$

by – (*rule G'-steps-V-last, auto*)

from *steps-mono*[*OF* $\langle G'.steps(y \# ys) \rangle \langle y \preceq \rightarrow \langle V y \rangle$ *this*] **obtain** ys'

where

$G'.steps(last(x \# xs) \# ys') list-all2(\preceq) ys ys'$

by *auto*

with *G'.steps-append*[*OF* $\langle G'.steps(x \# xs) \rangle$ *this*(1)] **show** *?thesis*

by *auto*

qed

lemma *steps-replicate-subsumption*:

assumes $x \preceq last(x \# xs) G'.steps(x \# xs) n > 0 \vee x$

notes [*intro*] = *preorder-intros*

shows $\exists ys. G'.steps(x \# ys) \wedge list-all2(\preceq) (concat(replicate n xs)) ys$

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.prem*s **show** *?thesis*

by (*inst-existentials xs*) (*auto intro: list-all2-refl*)

next

case *prems: (Suc n')*

with *Suc* $\langle n = \rightarrow$ **obtain** ys **where** ys :

$list-all2(\preceq) (concat(replicate n xs)) ys G'.steps(x \# ys)$

by *auto*

with $\langle n = \rightarrow$ **have** $list-all2(\preceq) (concat(replicate n' xs) @ xs) ys$

by (*metis append-Nil2 concat.simps(1,2) concat-append replicate-Suc replicate-append-same*)

with $\langle x \preceq \rightarrow$ **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* $\langle G'.steps(x \# ys) \rangle \langle G'.steps(x \# xs) \rangle$ *this*] $\langle V x \rangle$ **obtain**

```

    ys' where G'.steps (x # ys @ ys') list-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:  $\bigwedge x y. x \preceq y \implies y \preceq x \implies x = y$ 
  shows wf {(x, y). y <x  $\wedge$  V x  $\wedge$  V y} (is wf ?S)
proof (rule finite-acyclic-wf)
  have ?S  $\subseteq$  {(x, y). V x  $\wedge$  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 ( $\lambda x y. y <x \wedge V x \wedge V y$ )
  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:  $x \preceq x' \wedge G'.steps (x \# xs @ [x']) \wedge V x$ 
  shows  $\exists x'' ys. x' \preceq x'' \wedge G'.steps (x'' \# ys @ [x'']) \wedge V x''$ 
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  $\preceq$  x'>
obtain ys where
  G'.steps (x # ys) list-all2 ( $\preceq$ ) ?xs ys
by auto

```

```

let ?ys = filter (( $\preceq$ ) x^) ys
have length ?ys  $\geq$  ?n
  using list-all2-replicate-elem-filter[OF <list-all2 ( $\preceq$ ) ?xs ys>, of x^]
  by auto
have set ?ys  $\subseteq$  set ys
  by auto
also have ...  $\subseteq$  {x.  $\forall$  x}
  using G'-steps-V-all[OF <G'.steps (x # ys)>] <V x> unfolding list-all-iff
by auto
finally have  $\neg$  distinct ?ys
  using distinct-card[of ?ys] <-  $\geq$  ?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  $\forall$  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'  $\preceq$  y

```

proof –
from $\langle ?ys = - \rangle$ **have** $y \in \text{set } ?ys$ **by** *auto*
then show *?thesis* **by** *auto*
qed
ultimately show *?thesis*
by *auto*
qed

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

lemma *pre-cycle-cycle-reachable*:
 $(\exists x x'. a_0 \rightarrow^* x \wedge V x \wedge x \rightarrow^+ x' \wedge x \preceq x') \longleftrightarrow (\exists x. a_0 \rightarrow^* x \wedge V x \wedge x \rightarrow^+ x)$

proof –
interpret *interp: Subsumption-Graph-Pre-Nodes - - E* $\lambda x. a_0 \rightarrow^* x \wedge V x$
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* \longleftrightarrow $x \rightarrow^+ y$ **if** $a_0 \rightarrow^* x$ **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 \preceq y reachable x reachable y*

shows \exists *ys. steps (y # ys) \wedge list-all2 (\preceq) xs ys*

using *assms steps-mono* **by** (*simp add: reachable-steps-equiv*)

lemma *steps-append-subsumption:*

assumes *steps (x # xs) steps (y # ys) y \preceq last (x # xs) reachable x reachable y*

shows \exists *ys'. steps (x # xs @ ys') \wedge list-all2 (\preceq) ys ys'*

using *assms steps-append-subsumption* **by** (*simp add: reachable-steps-equiv*)

lemma *steps-replicate-subsumption:*

assumes *x \preceq last (x # xs) steps (x # xs) n > 0 reachable x*

notes [*intro*] = *preorder-intros*

shows \exists *ys. steps (x # ys) \wedge list-all2 (\preceq) (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: \bigwedge x y. x \preceq y \implies y \preceq x \implies x = y*

shows *wf {(x, y). y \prec x \wedge reachable x \wedge reachable y} (is wf ?S)*

proof (*rule finite-acyclic-wf*)

have *?S \subseteq {(x, y). reachable x \wedge 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* ($\lambda x y. y \prec x \wedge \text{reachable } x \wedge \text{reachable } y$)
by (*rule acyclicI-order*[**where** $f = \text{id}$]) *auto*
qed

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

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

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

end

end

end

context *Subsumption-Graph-Defs*
begin

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

lemma *G''-reachable-G'*[*intro*]:
 $G'.\text{reachable } x \text{ if } G''.\text{reachable } x$
using *that*
unfolding $G'.\text{reachable-def } G''.\text{reachable-def } G-G'.\text{reachable-iff } \text{Graph-Start-Defs}.\text{reachable-def}$
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  $\langle RE\ z'\ z \rangle$  show ?case
  by (blast intro: rtranclp.intros(2))
qed

```

end

```

locale Reachability-Compatible-Subsumption-Graph-Total = Reachability-Compatible-Subsumption-G
+
  assumes total: reachable  $a \implies$  reachable  $b \implies a \preceq b \vee b \preceq a$ 
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:
     $\forall s. G.reachable\ s \implies (\forall s'. E\ s\ s' \implies RE\ s\ s') \vee (\exists t. s \prec t \wedge G.reachable\ t)$ 
  assumes subgraph:  $\forall s\ s'. RE\ s\ s' \implies E\ s\ s'$ 

```

```

assumes finite-reachable: finite {a. G.reachable a}
assumes mono:
   $a \preceq b \implies E a a' \implies \text{reachable } a \implies G.\text{reachable } b \implies \exists b'. E b b' \wedge a' \preceq b'$ 
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:
   $\exists t. G.\text{reachable } t \wedge s \preceq t \wedge (\forall s'. E t s' \longrightarrow RE t s')$  if G.reachable s
  using that
proof –
  from finite-reachable  $\langle G.\text{reachable } s \rangle$  obtain x where
     $\forall s'. E x s' \longrightarrow RE x s' G.\text{reachable } x ((\prec)^{**}) s x$ 
  apply atomize-elim
  apply (induction rule: rtranclp-ev-induct2)
  using reachability-compatible by auto
  moreover from  $\langle ((\prec)^{**}) s x \rangle$  have  $s \prec x \vee s = x$ 
  by induction auto
  ultimately show ?thesis by auto
qed

lemma subsumption-step:

```

$\exists a'' b'. a' \preceq a'' \wedge b \preceq b' \wedge RE\ a''\ b' \wedge G.\text{reachable}\ a''$ **if**
 $\text{reachable}\ a\ E\ a\ b\ G.\text{reachable}\ a'\ a \preceq a'$

proof –

from $\text{mono}[OF\ \langle a \preceq a' \rangle\ \langle E\ a\ b \rangle\ \langle \text{reachable}\ a \rangle\ \langle G.\text{reachable}\ a' \rangle]$ **obtain**
 b' **where** $E\ a'\ b'\ b \preceq b'$

by *auto*

from $\text{reachable-has-surrogate}[OF\ \langle G.\text{reachable}\ a' \rangle]$ **obtain** a''

where $a' \preceq a''\ G.\text{reachable}\ a''$ **and** $*$: $\forall s'. E\ a''\ s' \longrightarrow RE\ a''\ s'$

by *auto*

from $\text{mono}[OF\ \langle a' \preceq a'' \rangle\ \langle E\ a'\ b' \rangle]\ \langle G.\text{reachable}\ a' \rangle\ \langle G.\text{reachable}\ a'' \rangle$
obtain b'' **where**

$E\ a''\ b''\ b' \preceq b''$

by *auto*

with $*$ $\langle a' \preceq a'' \rangle\ \langle b \preceq b' \rangle\ \langle G.\text{reachable}\ a'' \rangle$ **show** *?thesis* **by** *auto*

qed

theorem *reachability-complete'*:

$\exists s'. s \preceq s' \wedge G.\text{reachable}\ s'$ **if** $E^{**}\ a\ s\ G.\text{reachable}\ a$

using *that*

proof (*induction*)

case *base*

then show *?case* **by** *auto*

next

case (*step* $s\ t$)

then obtain s' **where** $s \preceq s'\ G.\text{reachable}\ s'$

by *auto*

with $\text{step}(4)$ **have** $\text{reachable}\ a\ G.\text{reachable}\ s'$

by *auto*

with $\text{step}(1)$ **have** $\text{reachable}\ s$

by (*auto simp: reachable-def*)

from $\text{subsumption-step}[OF\ \langle \text{reachable}\ s \rangle\ \langle E\ s\ t \rangle\ \langle G.\text{reachable}\ s' \rangle\ \langle s \preceq s' \rangle]$

obtain $s''\ t'$ **where**

$s' \preceq s''\ t \preceq t'\ s'' \rightarrow_G\ t'\ G.\text{reachable}\ s''$

by *atomize-elim*

with $\langle G.\text{reachable}\ s' \rangle$ **show** *?case*

by (*auto simp: reachable-def*)

qed

corollary *reachability-complete*:

$\exists s'. s \preceq s' \wedge G.\text{reachable}\ s'$ **if** $\text{reachable}\ s$

using $\text{reachability-complete}'[\text{of}\ s_0\ s]$ **that** *unfolding reachable-def* **by** *auto*

corollary *reachability-correct*:

$(\exists s'. s \preceq s' \wedge \text{reachable}\ s') \longleftrightarrow (\exists s'. s \preceq s' \wedge G.\text{reachable}\ s')$

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*:
 $\exists s'. s \preceq s' \wedge G.\text{reachable } s' \text{ if } G'.\text{reachable } s$
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 $s = \text{ASSERT } (s \neq \{\}) \gg \text{SPEC } (\lambda (x, s'). x \in s \wedge s' = s - \{x\})$

lemma *take-from-set-correct*:

assumes $s \neq \{\}$

shows *take-from-set* $s \leq \text{SPEC } (\lambda (x, s'). x \in s \wedge s' = s - \{x\})$

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 $s = \text{ASSERT } (s \neq \{\#\}) \gg \text{SPEC } (\lambda (x, s'). x \in\# s \wedge s' = s - \{\#x\#})$

lemma *take-from-mset-correct*:

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

lemmas [*refine-vcg*] = $take\text{-}from\text{-}mset\text{-}correct[THEN\ order.trans]$

lemma *set-mset-mp*: $set\text{-}mset\ m \subseteq s \implies n < count\ m\ x \implies x \in s$
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 (
 $\{(b, b').\ b \wedge \neg\ b'\}$
 $\langle *lex* \rangle$
 $\{(passed', passed).\$
 $passed' \subseteq \{a.\ reachable\ a \wedge \neg\ empty\ a\} \wedge passed \subseteq \{a.\ reachable\ a$
 $\wedge \neg\ empty\ a\} \wedge$
 $reachable\text{-}subsumed\ passed \subset reachable\text{-}subsumed\ passed'\}$
 $\langle *lex* \rangle$
measure size
 \rangle
 $(\lambda\ (a, b, c).\ (c, a, b))$

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

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

definition *pw-inv* $\equiv \lambda\ (passed, wait, brk).$

$(brk \longrightarrow (\exists\ f.\ reachable\ f \wedge F\ f)) \wedge$

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

definition $add\text{-pw-spec } passed \text{ wait } a \equiv SPEC (\lambda(passed', wait', brk).$

$\text{if } \exists a'. E a a' \wedge F a' \text{ then}$

brk

else

$\neg brk \wedge set\text{-mset } wait' \subseteq set\text{-mset } wait \cup \{a'. E a a'\} \wedge$

$(\forall s \in set\text{-mset } wait. \exists s' \in set\text{-mset } wait'. s \preceq s') \wedge$

$(\forall s \in \{a'. E a a' \wedge \neg \text{empty } a'\}. \exists s' \in set\text{-mset } wait' \cup passed. s$
 $\preceq s') \wedge$

$(\forall s \in passed \cup \{a\}. \exists s' \in passed'. s \preceq s') \wedge$

$(passed' \subseteq passed \cup \{a\} \cup \{a'. E a a' \wedge \neg \text{empty } a'\} \wedge$

$((\exists x \in passed'. \neg (\exists x' \in passed. x \preceq x')) \vee wait' \subseteq\# wait \wedge passed$

$= passed')$

$)$

$)$

definition

$init\text{-pw-spec} \equiv$

$SPEC (\lambda (passed, wait).$

$\text{if } \text{empty } a_0 \text{ then } passed = \{\} \wedge wait \subseteq\# \{\#a_0\# \} \text{ else } passed \subseteq \{a_0\}$

$\wedge wait = \{\#a_0\# \})$

abbreviation $subsumed\text{-elem} :: 'a \Rightarrow 'a \text{ set} \Rightarrow bool$

where $subsumed\text{-elem } a M \equiv \exists a'. a' \in M \wedge a \preceq a'$

notation

$subsumed\text{-elem} ((-/ \in'' -) [51, 51] 50)$

definition $pw\text{-inv-frontier}' passed \text{ wait} =$

$(\forall a. a \in passed \longrightarrow$

$(a \in' set\text{-mset } wait)$

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

lemma $pw\text{-inv-frontier-frontier}'$:

$pw\text{-inv-frontier}' passed \text{ wait}$ **if**

$pw\text{-inv-frontier } passed \text{ wait} passed \subseteq Collect \text{ reachable}$

using that unfolding $pw\text{-inv-frontier}'\text{-def } pw\text{-inv-frontier}\text{-def}$ **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 a0 then RETURN (True, {})
    else if empty a0 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*
 $\neg b \in' \text{set-mset } wait$
 $E b b'$
 $\neg \text{empty } b \neg \text{empty } b'$
 $b \in' \text{passed}$
 $\text{reachable } b \text{ passed} \subseteq \{a. \text{reachable } a \wedge \neg \text{empty } a\}$
shows $b' \in' \text{passed} \cup \text{set-mset } wait$

proof –

from $\langle b \in' \rightarrow \rangle$ **obtain** *b1* **where** $b1: b \preceq b1$ $b1 \in \text{passed}$
by *blast*

with *mono[OF this(1) $\langle E b b' \rangle$ $\langle \text{passed} \subseteq \rightarrow \rangle$ $\langle \text{reachable } b \rangle$ $\langle \neg \text{empty } b \rangle$]*

obtain $b1'$ **where**

$E b1 b1' b' \preceq b1'$

by *auto*

moreover then have $\neg \text{empty } b1'$

using *assms(5) empty-mono* **by** *blast*

moreover from *assms b1* **have** $\neg b1 \in' \text{set-mset } wait$

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-lem:*

assumes *pw-inv-frontier passed wait passed* $\subseteq \{a. \text{reachable } a \wedge \neg \text{empty } a\}$

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 \subseteq *Collect reachable*

$a \in \#$ *wait*

$\forall s \in \text{set-mset } (\text{wait} - \{\#a\# \}). \exists s' \in \text{set-mset } \text{wait}'. s \preceq s'$

$\forall s \in \{a'. E a a' \wedge \neg \text{empty } a'\}. \exists s' \in \text{passed} \cup \text{set-mset } \text{wait}'. s \preceq s'$

$\forall s \in \text{passed} \cup \{a\}. \exists s' \in \text{passed}'. s \preceq s'$

passed' \subseteq *passed* \cup $\{a\} \cup \{a' . E a a' \wedge \neg \text{empty } a\}$

pw-inv-frontier passed wait

passed \subseteq $\{a. \text{reachable } a \wedge \neg \text{empty } a\}$

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 \subseteq *Collect reachable*

$a \in \#$ *wait*

$\forall s. s \in' \text{set-mset } (\text{wait} - \{\#a\# \}) \longrightarrow s \in' \text{set-mset } \text{wait}'$

$\forall s \in \{a'. E a a' \wedge \neg \text{empty } a'\}. s \in' \text{passed} \cup \text{set-mset } \text{wait}'$

$\forall s. s \in' \text{passed} \cup \{a\} \longrightarrow s \in' \text{passed}'$

passed' \subseteq *passed* \cup $\{a\} \cup \{a' . E a a'\}$

pw-inv-frontier' passed wait

passed \subseteq $\{a. \text{reachable } a \wedge \neg \text{empty } a\}$

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** $b \in \text{passed} \mid a = b \mid E a b$

by *auto*

note *cases = this*

from *cases* $\langle \neg b \in' \text{set-mset } \text{wait}' \rangle$ *assms'(4)* $\langle \text{reachable } a \rangle$ $\langle \text{passed}$
 $\subseteq \rightarrow$ **have** *reachable b*

by *cases (auto intro: reachable-step)*

with *A(3,4)* **have** $\neg \text{empty } b$ **by** (*auto simp: empty-E*)

from *cases this* $\langle \text{reachable } b \rangle$ **consider** $a = b \mid a \neq b \mid b \in' \text{passed}$
reachable b

```

    using  $\langle \neg b \in' \text{set-mset wait}' \rangle \text{ assms}'(4)$  by cases (fastforce intro:
reachable-step)+
  then consider  $b \preceq a$  reachable b |  $\neg b \preceq a$   $b \in'$  passed reachable b
    using  $\langle \neg b \in' \text{set-mset wait}' \rangle \text{ assms}'(4)$   $\langle \text{reachable } a \rangle$  by fastforce+
  then show ?case
proof cases
  case 1
    with  $A(3,4)$  have  $\neg \text{empty } b$ 
      by (auto simp: empty-E)
    with  $\text{mono}[OF\ 1(1)\ \langle E\ b\ b' \rangle\ 1(2)\ \langle \text{reachable } a \rangle]$  obtain  $b1'$  where
       $E\ a\ b1'\ b' \preceq b1'$ 
      by auto
    with  $\langle \neg \text{empty } b' \rangle$  have  $b1' \in'$  passed  $\cup \text{set-mset wait}'$ 
      using  $\text{assms}'(4)$  by (auto dest: empty-mono)
    with  $\langle b' \preceq \rightarrow \text{assms}'(5) \rangle$  show ?thesis
      by (auto intro: trans)
  next
  case 2
    with  $A(3,4)$  have  $\neg \text{empty } b$ 
      by (auto simp: empty-E)
    from  $2\ \langle \neg b \in' \text{set-mset wait}' \rangle \text{ assms}'(2,3)$  have  $\neg b \in'$  set-mset
wait
      by (metis insert-DiffM2 mset-right-cancel-elem)
    from
       $\text{aux3-aux}[OF$ 
         $\text{assms}'(7)\ \text{this}\ \langle E\ b\ b' \rangle\ \langle \neg \text{empty } b \rangle\ \langle \neg \text{empty } b' \rangle\ \langle b \in' \text{passed} \rangle$ 
 $\langle \text{reachable } b \rangle \text{ assms}'(8)$ 
      ]
      have  $b' \in'$  passed  $\cup \text{set-mset wait}$  .
      with  $\text{assms}'(2,3,5)$  show ?thesis
      by auto (metis insert-DiffM insert-noteq-member)
  qed
qed
done
qed

```

```

private lemma aux6:
assumes
   $a \in\# \text{wait}$ 
  start-subsumed passed wait
   $\forall s \in \text{set-mset} (\text{wait} - \{\#a\}) \cup \{a'. E\ a\ a' \wedge \neg \text{empty } a'\}. \exists s' \in$ 
set-mset wait'. s  $\preceq s'$ 
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  $\langle x \in \# \text{ wait} \rangle$  have  $x \in \text{set-mset} (\text{wait} - \{\#a\# \})$ 
        by (metis insert-DiffM insert-noteq-member prems(1))
      with prems(2,4)  $\langle - \preceq x \rangle$  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
   $\{ \# \}$ 
     $\text{passed} \subseteq \{ a. \text{reachable } a \wedge \neg \text{empty } a \} \neg \text{empty } x$ 
  shows  $\exists x' \in \text{passed}. x \preceq x'$ 
  proof –
    from  $\langle \text{reachable } x \rangle$  have  $E^{**} a_0 x$  by (simp add: reachable-def)
    have  $\neg \text{empty } a_0$  using  $\langle E^{**} a_0 x \rangle$  assms(5) empty-E-star by blast
    with assms(3) obtain b where  $a_0 \preceq b$   $b \in \text{passed}$  unfolding start-subsumed-def
  by auto
    have  $\exists x'. \exists x''. E^{**} b x' \wedge x \preceq x' \wedge x' \preceq x'' \wedge x'' \in \text{passed}$  if
       $E^{**} a x \ a \preceq b \ b \preceq b' \ b' \in \text{passed}$ 
      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.premis 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.premis step.hyps have empty x by – (rule empty-E-star,
auto simp: reachable-def)
    with step.premis 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.premis 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

```

$a \in \# \text{ wait}$
 $\text{set-mset wait} \subseteq \text{Collect reachable}$
 $\text{set-mset wait}' \subseteq \text{set-mset (wait - \{\#a\})} \cup \text{Collect (E a)}$
 $x \in \# \text{ wait}'$
shows *reachable x*
using *assms by (auto dest: in-diffD)*

private lemma *aux8:*

$x \in \text{reachable-subsumed } S'$ **if** $x \in \text{reachable-subsumed } S \forall s \in S. \exists x \in S'. s \preceq x$
using *that unfolding reachable-subsumed-def by (auto intro: trans)*

private lemma *aux9:*

assumes
 $\text{set-mset wait}' \subseteq \text{set-mset (wait - \{\#a\})} \cup \text{Collect (E a)}$
 $x \in \# \text{ wait}' \forall a'. E a a' \longrightarrow \neg F a' F x$
 $\forall a \in \text{passed} \cup \text{set-mset wait}. \neg F a$
shows *False*

proof –

from *assms(1,2)* **have** $x \in \text{set-mset wait} \vee x \in \text{Collect (E a)}$
by *(meson UnE in-diffD subsetCE)*
with *assms(3,4,5)* **show** *?thesis*
by *auto*

qed

private lemma *aux10:*

assumes $\forall a \in \text{passed}' \cup \text{set-mset wait}. \neg F a F x x \in \# \text{ wait} - \{\#a\}$
shows *False*
by *(meson UnI2 assms in-diffD)*

lemma *aux12:*

$\text{size wait}' < \text{size wait}$ **if** $\text{wait}' \subseteq \# \text{ wait} - \{\#a\} a \in \# \text{ wait}$
using *that*
by *(metis*
 $\text{Diff-eq-empty-iff-mset add-diff-cancel-left' add-mset-add-single add-mset-not-empty}$
 $\text{insert-subset-eq-iff mset-le-add-mset-decr-left1 mset-subset-size sub-}$
 $\text{set-mset-def})$

lemma *aux13:*

assumes
 $\text{passed} \subseteq \{a. \text{reachable } a \wedge \neg \text{empty } a\}$
 $\text{passed}' \subseteq \text{insert } a (\text{passed} \cup \{a'. E a a' \wedge \neg \text{empty } a'\})$
 $\neg \text{empty } a$
 $\text{reachable } a$

```

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

method solve-vc =
    rule aux3 aux5 aux7 aux10 aux11 pw-inv-frontier-empty-elim; 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:
     $\text{pw-algo} \leq \text{SPEC } (\lambda (brk, \text{passed}).$ 
     $(brk \longleftrightarrow F\text{-reachable})$ 
     $\wedge (\neg brk \longrightarrow$ 
     $(\forall a. \text{reachable } a \wedge \neg \text{empty } a \longrightarrow (\exists b \in \text{passed}. a \preceq b))$ 
     $\wedge \text{passed} \subseteq \{a. \text{reachable } a \wedge \neg \text{empty } a\})$ 
     $)$ 
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: fi-
nal-non-empty)+
qed

lemmas (in Search-Space'-finite) [refine-vcg] = pw-algo-correct[THEN Or-
derings.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  $\equiv$  SPEC ( $\lambda$ (passed',wait',brk)).
  if  $\exists x \in \text{set } (\text{succs } a)$ . F x then brk
  else  $\text{passed}' \subseteq \text{passed} \cup \{x \in \text{set } (\text{succs } a) . \neg (\exists y \in \text{passed} . x \preceq y)\}$ 
     $\wedge \text{passed} \subseteq \text{passed}'$ 
     $\wedge \text{wait} \subseteq \# \text{wait}'$ 
     $\wedge \text{wait}' \subseteq \# \text{wait} + \text{mset } ([x \leftarrow \text{succs } a . \neg (\exists y \in \text{passed} . x \preceq y)])$ 
     $\wedge (\forall x \in \text{set } (\text{succs } a) . \exists y \in \text{passed}' . x \preceq y)$ 
     $\wedge (\forall x \in \text{set } (\text{succs } a) . \neg (\exists y \in \text{passed} . x \preceq y) \longrightarrow (\exists y \in \# \text{wait}' .$ 
x  $\preceq y))$ 
     $\wedge \neg \text{brk}$ 

```

```

definition add-pw passed wait a  $\equiv$ 
  nfoldli (succs a) ( $\lambda$ (-, -, brk).  $\neg \text{brk}$ )
  ( $\lambda a$  (passed, wait, brk). RETURN (
    if F a then
      (passed, wait, True)
    else if  $\exists x \in \text{passed} . a \preceq x$  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 ≤ add-pw-spec passed wait a
if reachable a a ∈ 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 ≤ ↓ Id (add-pw-unified-spec passed wait a)
unfolding add-pw-def add-pw-unified-spec-def
apply (refine-vcg
  nfoldli-rule where I =
     $\lambda l1 l2 (passed', wait', brk).$ 
    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* \equiv
 $\text{ifoldli (succs } a) (\lambda(-, -, brk). \neg brk)$
 $(\lambda a (\text{passed}, \text{wait}, brk). \text{RETURN (}$
 if F a then
 $(\text{passed}, \text{wait}, \text{True})$
 else if empty a then
 $(\text{passed}, \text{wait}, \text{False})$
 else if $\exists x \in \text{passed}. a \sqsubseteq x$ then
 $(\text{passed}, \text{wait}, \text{False})$
 else (insert a passed, add-mset a wait, False)
 $)$
 $(\text{passed}, \text{wait}, \text{False})$

definition *pw-algo-unified where*

pw-algo-unified = *do*
 {
 if F a₀ then RETURN (True, {})
 else if empty a₀ then RETURN (False, {})
 else do {
 $(\text{passed}, \text{wait}) \leftarrow \text{RETURN} (\{a_0\}, \{\#a_0\#});$
 $(\text{passed}, \text{wait}, brk) \leftarrow \text{WHILEIT } pw\text{-inv } (\lambda (\text{passed}, \text{wait}, brk). \neg$
brk \wedge wait \neq {})
 $(\lambda (\text{passed}, \text{wait}, brk). \text{do}$
 {
 $(a, \text{wait}) \leftarrow \text{take-from-mset } \text{wait};$
 $\text{ASSERT (reachable } a);$
 if empty a then RETURN (passed, wait, brk) else add-pw'
 passed wait a
 }
)
 $(\text{passed}, \text{wait}, \text{False});$
 $\text{RETURN (brk, passed)}$
 }
 }

end — Worklist 2 Defs

context *Worklist2*

begin

lemma *empty-subsumes'2:*

empty $x \vee x \leq y \longleftrightarrow x \preceq y$
using *empty-subsumes'* *empty-subsumes* **by** *auto*

lemma *bex-or*:

$P \vee (\exists x \in S. Q x) \longleftrightarrow (\exists x \in S. P \vee Q x)$ **if** $S \neq \{\}$
using *that* **by** *auto*

lemma *add-pw'-ref'*:

add-pw' *passed* *wait* $a \leq \Downarrow (Id \cap \{((p, w, -), -). p \neq \{\} \wedge \text{set-mset } w \subseteq p\})$ (*add-pw* *passed* *wait* a)
if $\text{passed} \neq \{\}$ *set-mset* *wait* \subseteq *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
 $\leq \Downarrow (Id \cap \{((p, w, -), -). p \neq \{\} \wedge \text{set-mset } w \subseteq p\})$ (*add-pw-spec* *passed'*
wait' a')
if $\text{passed} \neq \{\}$ *set-mset* *wait* \subseteq *passed* *reachable* $a \in$ *passed*
and [*simp*]: $\text{passed} = \text{passed}'$ $\text{wait} = \text{wait}'$ $a = a'$

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* $a \leq \Downarrow Id$ (*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*:

$p \leq \Downarrow R$ $p' \text{ if } p \leq \Downarrow S$ $p' S \subseteq R$
using *that* **by** (*simp* *add: pw-le-iff* *refine-pw-simps; blast*)

lemma *add-pw'-ref*:

add-pw' *passed* *wait* $a \leq$
 $\Downarrow (\{((p, w, b), (p', w', b')). p \neq \{\} \wedge p = p' \cup \text{set-mset } w \wedge w = w' \wedge$

$b = b'$)
 (add-pw-spec passed' wait' a')
if $passed \neq \{\}$ set-mset wait \subseteq passed reachable $a \in passed$
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 $s = \text{ASSERT } (s \neq []) \gg \text{SPEC } (\lambda (x, s'). s = x \# s')$

lemma take-from-list-correct:

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

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

context *Worklist-Map-Defs*
begin

definition

add-pw'-map passed wait a \equiv
nfoldli (succs a) ($\lambda(-, -, brk). \neg brk$)
(λa (*passed, wait, -*).
do {
RETURN (
if *F a* then (*passed, wait, True*) else
let *k = key a*; *passed' = (case passed k of Some passed' \Rightarrow passed' |*
None \Rightarrow {})
in
if empty *a* then
(*passed, wait, False*)
else if $\exists x \in$ *passed'*. *a \sqsubseteq x* then
(*passed, wait, False*)
else
(*passed(k \mapsto (insert a passed')), a # wait, False*)
)
}
)
(*passed, wait, False*)

definition

pw-map-inv $\equiv \lambda$ (*passed, wait, brk*).
 \exists *passed' wait'*.
(*passed, passed'*) \in *map-set-rel* \wedge (*wait, wait'*) \in *list-mset-rel* \wedge
pw-inv (passed', wait', brk)

definition *pw-algo-map where*

pw-algo-map = do
{
if *F a₀* then RETURN (*True, Map.empty*)
else if empty *a₀* then RETURN (*False, Map.empty*)
else do {
(*passed, wait*) \leftarrow RETURN (*[key a₀ \mapsto {a_{0}}}*], [*a₀*]);
(*passed, wait, brk*) \leftarrow WHILEIT *pw-map-inv* (λ (*passed, wait, brk*).
 \neg *brk \wedge wait \neq []*)
(λ (*passed, wait, brk*). do
{
(*a, wait*) \leftarrow *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*:

$(x \in \text{ran } m) \vee (x = y)$ **if** $x \in \text{ran } (m(a \mapsto y))$
using that unfolding *ran-def* **by** (*auto split: if-split-asm*)

lemma *ran-upd-cases2*:

$(\exists k. m k = \text{Some } x \wedge k \neq a) \vee (x = y)$ **if** $x \in \text{ran } (m(a \mapsto y))$
using that unfolding *ran-def* **by** (*auto split: if-split-asm*)

context *Worklist-Map*

begin

lemma *add-pw'-map-ref[refine]*:

$\text{add-pw'-map } \text{passed } \text{wait } a \leq \Downarrow (\text{map-set-rel } \times_r \text{ list-mset-rel } \times_r \text{ bool-rel})$
 $(\text{add-pw'} \text{ passed}' \text{ wait}' a')$

if $(\text{passed}, \text{passed}') \in \text{map-set-rel } (\text{wait}, \text{wait}') \in \text{list-mset-rel } (a, a') \in \text{Id}$
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** $a \ a' \ - \ - \ \text{passed}' \ - \ \text{wait}' \ f' \ \text{passed} \ - \ \text{wait } f$

proof —

from *assms* **have** [*simp*]: $a' = a \ f = f'$ **by** *simp+*

from *assms* **have** *rel-passed*: $(\text{passed}, \text{passed}') \in \text{map-set-rel}$ **by** *simp*

then have *union*: $\text{passed}' = \bigcup (\text{ran } \text{passed})$

unfolding *map-set-rel-def* **by** *auto*

from *assms* **have** *rel-wait*: $(\text{wait}, \text{wait}') \in \text{list-mset-rel}$ **by** *simp*

from *rel-passed* **have** *keys[simp]*: $\text{key } v = k$ **if** $\text{passed } k = \text{Some } xs \ v \in xs$ **for** $k \ xs \ v$

using that unfolding *map-set-rel-def* **by** *auto*

```

define  $k$  where  $k \equiv \text{key } a$ 
define  $xs$  where  $xs \equiv \text{case passed } k \text{ of } \text{None} \Rightarrow \{\} \mid \text{Some } p \Rightarrow p$ 
have  $xs\text{-ran}$ :  $x \in \bigcup (\text{ran passed})$  if  $x \in xs$  for  $x$ 
  using that unfolding xs-def ran-def by (auto split: option.split-asm)
have  $*$ :  $(\exists x \in xs. a \trianglelefteq x) \longleftrightarrow (\exists x \in \text{passed}'. a' \trianglelefteq x)$ 
proof standard
  assume  $\exists x \in xs. a \trianglelefteq x$ 
  with  $\text{rel-passed}$  show  $\exists x \in \text{passed}'. a' \trianglelefteq x$ 
    unfolding  $xs\text{-def union}$  by (auto intro: ranI split: option.split-asm)
  next
  assume  $\exists x \in \text{passed}'. a' \trianglelefteq x$ 
  with  $\text{rel-passed}$  show  $\exists x \in xs. a \trianglelefteq x$  unfolding  $xs\text{-def union ran-def}$ 
 $k\text{-def map-set-rel-def}$ 
  using empty-subsumes'2 by force
qed
have  $(\text{passed}(k \mapsto \text{insert } a \text{ } xs), \text{insert } a' \text{ } \text{passed}') \in \text{map-set-rel}$ 
  using  $\langle (\text{passed}, \text{passed}') \in \text{map-set-rel} \rangle$ 
  unfolding  $\text{map-set-rel-def}$ 
  apply safe
  subgoal
    unfolding  $\text{union}$  by (auto dest!: ran-upd-cases xs-ran)
  subgoal
    unfolding  $\text{ran-def}$  by auto
  subgoal for  $a''$ 
    unfolding  $\text{union ran-def}$ 
    apply clarsimp
    subgoal for  $k'$ 
      unfolding  $xs\text{-def}$  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  $\text{rel-wait rel-passed}$  show  $?thesis$ 
  unfolding  $*[\text{symmetric}]$ 
  unfolding  $xs\text{-def } k\text{-def } \text{Let-def}$ 
  unfolding  $\text{list-mset-rel-def } \text{br-def}$ 
  by auto
qed
done

```

```

lemma  $\text{init-map-ref}[\text{refine}]$ :
   $(([\text{key } a_0 \mapsto \{a_0\}], [a_0]), \{a_0\}, \{\#a_0\}) \in \text{map-set-rel} \times_r \text{list-mset-rel}$ 
  unfolding  $\text{map-set-rel-def list-mset-rel-def } \text{br-def}$  by auto

```

```

lemma  $\text{take-from-list-ref}[\text{refine}]$ :

```

$take-from-list\ xs \leq \Downarrow (Id \times_r list-mset-rel) (take-from-mset\ ms)$ **if** $(xs, ms) \in list-mset-rel$
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 $pw-algo-map-ref:$
 $pw-algo-map \leq \Downarrow (Id \times_r map-set-rel) pw-algo-unified$
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

$add-pw'-map2\ passed\ wait\ a \equiv$
 $nfoldli\ (succs\ a)\ (\lambda(-, -, brk).\ \neg brk)$
 $(\lambda a\ (passed, wait, -).$
 $\ do\ \{$
 $\ \ \ \ RETURN\ ($
 $\ \ \ \ \ \ if\ empty\ a\ then$
 $\ \ \ \ \ \ \ \ (passed, wait, False)$
 $\ \ \ \ \ \ else\ if\ F'\ a\ then\ (passed, wait, True)$
 $\ \ \ \ \ \ else$
 $\ \ \ \ \ \ \ \ let\ k = key\ a;\ passed' = (case\ passed\ k\ of\ Some\ passed' \Rightarrow passed' \mid$
 $\ \ \ \ \ \ \ \ None \Rightarrow \{\})$
 $\ \ \ \ \ \ \ \ in$
 $\ \ \ \ \ \ \ \ \ \ if\ \exists\ x \in passed'.\ a \trianglelefteq x\ then$
 $\ \ \ \ \ \ \ \ \ \ \ \ (passed, wait, False)$
 $\ \ \ \ \ \ \ \ \ \ else$
 $\ \ \ \ \ \ \ \ \ \ \ \ (passed(k \mapsto (insert\ a\ passed')), a \# wait, False)$
 $\ \ \ \ \ \ \ \ \ \)$
 $\ \ \ \ \ \ \ \ \ \ }$
 $\ \ \ \ \ \ \ \ \ \)$
 $\ \ \ \ \ \ \ \ \ \ (passed, wait, False)$

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 \leq *SPEC* (λ (*brk*, *passed*).
 (*brk* \longleftrightarrow *F-reachable*) \wedge
 (\neg *brk* \longrightarrow
 (\exists *p*.
 (*passed*, *p*) \in *map-set-rel* \wedge (\forall *a*. *reachable a* \wedge \neg *empty a* \longrightarrow (\exists *b* \in *p*.
a \preceq *b*))
 \wedge *p* \subseteq {*a*. *reachable a* \wedge \neg *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* \Rightarrow *assn* (\wedge^* - [60] 90) **where**
big-star S \equiv *fold-mset* (*) *emp S*

interpretation *comp-fun-commute-mult*:

comp-fun-commute (*) :: (*'a* :: *ab-semigroup-mult* \Rightarrow - \Rightarrow -)

by *standard* (*auto simp: ab-semigroup-mult-class.mult.left-commute*)

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

lemma *sep-big-star-union* [*simp*]: \wedge^* (*S* + *T*) = (\wedge^* *S*) * (\wedge^* *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.prem_s mset-add mset-subset-eq-add-mset-cancel)

with add.IH[of *S'*] add.prem_s have $\bigwedge^* T \Longrightarrow_t \bigwedge^* S'$

by *auto*

then show ?thesis

unfolding $\langle S = \rightarrow \rangle$ by (sep-auto intro: entt-star-mono)

next

case *False*

with add.prem_s 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*:

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

lemma *in-mset-dom-union-split*:

$mset\text{-}set\ (dom\ m) = mset\text{-}set\ (dom\ m - \{k\}) + \{\#k\#}$ **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*:

$map\text{-}assn\ A\ m\ mh \implies_A$
 $map\text{-}assn\ A\ (m(k := None))\ (mh(k := None)) * option\text{-}assn\ A\ (m\ k)$
 $(mh\ k)$
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]*:

$z \in \# mset\text{-}set\ S \iff z \in S$ **if** *finite* S
using *that* **by** *auto*

lemma *ent-refl'*:

$a = b \implies a \implies_A b$
by *auto*

lemma *map-assn-update-aux*:

$map\text{-}assn\ A\ m\ mh * A\ v\ vi \implies_A map\text{-}assn\ A\ (m(k \mapsto v))\ (mh(k \mapsto vi))$
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* \implies_A

*map-assn A (m(k \mapsto v)) (mh(k \mapsto 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 Chargueraud.

definition (**in** *imp-map*) *hms-assn A m mi* $\equiv \exists_A mh. is-map mh mi * map-assn A m mh$

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* $\equiv empty$

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 \mapsto 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 \mapsto 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 $mh\ k = None \longleftrightarrow m\ k = None$ *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
 $\langle \lambda (vi, mi').\ option-assn\ A\ (m\ k)\ vi * hms-assn\ A\ (m(k := None))\ mi' \rangle_t$
unfolding *hms-extract-def hms-assn-def*
apply (*sep-auto intros 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

$\langle hms\text{-}assn\ A\ m\ mi \rangle$

hms-lookup lookup copy k mi

$\langle \lambda\ vi.\ hms\text{-}assn\ A\ m\ mi\ * \ option\text{-}assn\ A\ (m\ k)\ vi \rangle_t$

proof –

have 0:

$\langle is\text{-}map\ mh\ mi\ * \ map\text{-}assn\ A\ (m(k := None))\ (mh(k := None))\ * \ option\text{-}assn\ A\ (m\ k)\ (mh\ k)\ * \ true \rangle$

copy x'

$\langle \lambda r.\ \exists_A x.$

*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:

$\langle is\text{-}map\ mh\ mi\ * \ map\text{-}assn\ A\ m\ mh\ * \ true \rangle$

copy x'

$\langle \lambda r.\ \exists_A x.\ is\text{-}map\ mh\ mi\ * \ map\text{-}assn\ A\ m\ mh\ * \ A\ x\ r\ * \ \uparrow(m\ k = Some\ x)\ * \ true \rangle$

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 ↦ x) = m)
  apply (subgoal-tac mh(k ↦ 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-assnk *a Ad *a (hms-assn A)d →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-assnk
  →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* \equiv *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*:

$\lambda k m. (m\ k, m(k := None)) :: K \rightarrow \langle K, V \rangle \text{map-rel} \rightarrow \langle V \rangle \text{option-rel} \times_r \langle K, V \rangle \text{map-rel}$

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)*)
 \in

$id\text{-assn}^k *_a (hms\text{-assn } A)^d \rightarrow_a \text{prod-assn } (option\text{-assn } A) (hms\text{-assn } A)$

by *sepref-to-hoare sep-auto*

lemma *hms-lookup-hnr*:

(*uncurry (hms-lookup lookup copy), uncurry (RETURN oo op-map-lookup)*)
 \in

$id\text{-assn}^k *_a (hms\text{-assn } A)^k \rightarrow_a option\text{-assn } A$ **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) ( $\lambda(-, -, brk). \neg brk$ )
  ( $\lambda a$  (passed, wait, -).
    do {
      RETURN (
        if empty a then
          (passed, wait, False)
        else if F' a then (passed, wait, True)
        else
          let
            k = key a;
            (v, passed) = op-map-extract k passed
          in
            case v of
              None  $\Rightarrow$  (passed(k  $\mapsto$  {COPY a}), a # wait, False) |
              Some passed'  $\Rightarrow$ 
                if  $\exists x \in$  passed'. a  $\trianglelefteq$  x then
                  (passed(k  $\mapsto$  passed'), wait, False)
                else
                  (passed(k  $\mapsto$  (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) ( $\lambda(-, -, brk). \neg brk$ )
  ( $\lambda a$  (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) ∈ $A^k \rightarrow_a K$

assumes [sepref-fr-rules]: (copyi, RETURN o COPY) ∈ $A^k \rightarrow_a A$

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 a₀ PR-CONST F' PR-CONST (\trianglelefteq) PR-CONST succs
PR-CONST empty PR-CONST key
PR-CONST F trace

lemma [*def-pat-rules*]:
a₀ \equiv UNPROTECT a₀ F' \equiv UNPROTECT F' (\trianglelefteq) \equiv UNPROTECT
(\trianglelefteq) succs \equiv UNPROTECT succs
empty \equiv UNPROTECT empty key_i \equiv UNPROTECT key_i F \equiv UN-
PROTECT 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 \implies 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* {a₀} *lso-fold-custom-empty*)

unfolding *hm.hms-fold-custom-empty*

apply (*rewrite in* [a₀] *HOL-list.fold-custom-empty*)

apply (*rewrite in* {} *lso-fold-custom-empty*)

unfolding *F-split*

by *sepref*

concrete-definition (*in* -) *pw-impl*

for *Lei a₀i 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* (λbrk. 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 a₀i Fi succsi emptyi*), *uncurry0* (*RETURN F-reachable*))

$\in \text{unit-assn}^k \rightarrow_a \text{bool-assn}$

using

pw-impl.refine[

OF Worklist-Map2-Impl-axioms,

FCOMP pw-algo-map2-correct'[*THEN Id-SPEC-refine*, *THEN nres-rell*]

]

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 a₀i* ($\lambda a_0i. (\text{uncurry0 } a_0i, \text{uncurry0 } (\text{RETURN } a_0)) \in \text{unit-assn}^k \rightarrow_a A$)

assumes *GEN-ALGO Fi* ($\lambda Fi. (Fi, \text{RETURN } o F) \in A^k \rightarrow_a \text{bool-assn}$)

assumes *GEN-ALGO Lei* ($\lambda Lei. (\text{uncurry } Lei, \text{uncurry } (\text{RETURN } oo (\trianglelefteq))) \in A^k *_a A^k \rightarrow_a \text{bool-assn}$)

assumes *GEN-ALGO succsi* ($\lambda succsi. (\text{succsi}, \text{RETURN } o \text{succs}) \in A^k \rightarrow_a \text{list-assn } A$)

assumes *GEN-ALGO emptyi* ($\lambda Fi. (Fi, \text{RETURN } o \text{empty}) \in A^k \rightarrow_a \text{bool-assn}$)

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 a₀i Fi succsi emptyi*),

uncurry0 (*RETURN (PR-CONST op-F-reachable)*))

$\in \text{unit-assn}^k \rightarrow_a \text{bool-assn}$

proof –

from *assms interpret*

```

    Worklist-Map2-Impl
    E a0 F (≼) succs empty (≼) F' A succsi a0i 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 ≠ {}) ≫ SPEC (λ (x, s'). x ∈ s ∧ s' = s
- {x})

```

```

lemma take-from-set-correct:

```

```

  assumes s ≠ {}

```

```

  shows take-from-set s ≤ SPEC (λ (x, s'). x ∈ s ∧ 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 $s = \text{ASSERT } (s \neq \{\#\}) \gg \text{SPEC } (\lambda (x, s'). x \in\# s \wedge s' = s - \{\#x\#})$

lemma *take-from-mset-correct*:

assumes $s \neq \{\#\}$

shows $\text{take-from-mset } s \leq \text{SPEC } (\lambda (x, s'). x \in\# s \wedge s' = s - \{\#x\#})$

using *assms* **unfolding** *take-from-mset-def* **by** *simp*

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

lemma *set-mset-mp*: $\text{set-mset } m \subseteq s \implies n < \text{count } m \ x \implies x \in s$

by (*meson count-greater-zero-iff le-less-trans subsetCE zero-le*)

lemma *pred-not-lt-is-zero*: $(\neg n - \text{Suc } 0 < n) \longleftrightarrow n=0$ **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* $\text{passed } \text{wait} = (\exists a \in \text{passed} \cup \text{set-mset } \text{wait}. a_0 \preceq a)$

definition

worklist-var =

$\text{inv-image } (\text{finite-psupset } (\text{Collect } \text{reachable}) <*\text{lex}*> \text{measure size}) (\lambda (a, b, c). (a, b))$

definition *worklist-inv-frontier* $\text{passed } \text{wait} =$

$(\forall a \in \text{passed}. \forall a'. E a a' \wedge \neg \text{empty } a' \longrightarrow (\exists b' \in \text{passed} \cup \text{set-mset } \text{wait}. a' \preceq b'))$

definition *worklist-inv* $\equiv \lambda (\text{passed}, \text{wait}, \text{brk}).$

$\text{passed} \subseteq \text{Collect } \text{reachable} \wedge$

$(\text{brk} \longrightarrow (\exists f. \text{reachable } f \wedge F f)) \wedge$

$(\neg brk \longrightarrow$
 $\quad worklist-inv-frontier\ passed\ wait$
 $\wedge (\forall a \in passed \cup set-mset\ wait. \neg F\ a)$
 $\wedge worklist-start-subsumed\ passed\ wait$
 $\wedge set-mset\ wait \subseteq Collect\ reachable)$

definition *add-succ-spec* $wait\ a \equiv SPEC\ (\lambda(wait',brk).$

$\quad if\ \exists a'. E\ a\ a' \wedge F\ a'$ then
 $\quad\quad brk$
 $\quad else$
 $\quad\quad \neg brk \wedge set-mset\ wait' \subseteq set-mset\ wait \cup \{a' . E\ a\ a'\} \wedge$
 $\quad\quad (\forall s \in set-mset\ wait \cup \{a' . E\ a\ a' \wedge \neg empty\ a'\}. \exists s' \in set-mset$
 $\quad\quad wait'. s \preceq s')$
 $\quad)$

definition *worklist-algo* **where**

$worklist-algo = do$
 $\quad \{$
 $\quad\quad if\ F\ a_0$ then *RETURN* *True*
 $\quad\quad else\ do\ \{$
 $\quad\quad\quad let\ passed = \{\};$
 $\quad\quad\quad let\ wait = \{\#a_0\#};$
 $\quad\quad\quad (passed, wait, brk) \leftarrow WHILEIT\ worklist-inv\ (\lambda (passed, wait, brk).$
 $\quad\quad\quad \neg brk \wedge wait \neq \{\#\})$
 $\quad\quad\quad (\lambda (passed, wait, brk). do$
 $\quad\quad\quad\quad \{$
 $\quad\quad\quad\quad\quad (a, wait) \leftarrow take-from-mset\ wait;$
 $\quad\quad\quad\quad\quad ASSERT\ (reachable\ a);$
 $\quad\quad\quad\quad\quad if\ (\exists a' \in passed. a \preceq a')$ then *RETURN* $(passed, wait, brk)$
 $\quad\quad\quad\quad\quad do$
 $\quad\quad\quad\quad\quad\quad \{$
 $\quad\quad\quad\quad\quad\quad\quad (wait,brk) \leftarrow add-succ-spec\ wait\ a;$
 $\quad\quad\quad\quad\quad\quad\quad let\ passed = insert\ a\ passed;$
 $\quad\quad\quad\quad\quad\quad\quad RETURN\ (passed, wait, brk)$
 $\quad\quad\quad\quad\quad\quad\quad \}$
 $\quad\quad\quad\quad\quad \}$
 $\quad\quad\quad\quad)$
 $\quad\quad\quad\quad (passed, wait, False);$
 $\quad\quad\quad\quad RETURN\ brk$
 $\quad\quad\quad \}$
 $\quad \}$

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 $\forall x \in \text{passed}. \neg a \preceq x$

and $\text{passed} \subseteq \text{Collect reachable}$

and *reachable a*

shows

(*insert a passed, wait', brk'*),

passed, wait, brk)

\in *worklist-var*

proof –

from *assms* **have** $a \notin \text{passed}$ **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

$a' \in \text{passed}$

$a \preceq a'$

$a \in \# \text{wait}$

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

$a' \in \text{passed}$

$a \preceq a'$

$a \in \# \text{ wait}$

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 \subseteq *Collect reachable*

$a \in \# \text{ wait}$

$\forall s \in \text{set-mset} (\text{wait} - \{\#a\# \}) \cup \{a'. E a a' \wedge \neg \text{empty } a'\}. \exists s' \in \text{set-mset } \text{wait}'. s \preceq s'$

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 $A: E b b' \neg \text{empty } b' b \in \text{passed}$

with *assms* **obtain** b'' **where** $b'': b'' \in \text{passed} \cup \text{set-mset } \text{wait } b' \preceq$

b''

unfolding *worklist-inv-frontier-def* **by** *blast*

from *this(1)* **show** *?thesis*

apply *standard*

subgoal

using $\langle b' \preceq b'' \rangle$ **by** *auto*

subgoal

apply (*cases a = b''*)

subgoal

using $b''(2)$ **by** *blast*

subgoal *premises* *prems*

```

proof –
  from prems have  $b'' \in \# \text{ wait} - \{\#a\# \}$ 
    by (auto simp: mset-remove-member)
  with assms prems  $\langle b' \preceq b'' \rangle$  show ?thesis
    by (blast intro: local.trans)
  qed
done
done
qed
done
qed

```

```

private lemma aux6:
  assumes
     $a \in \# \text{ wait}$ 
    worklist-start-subsumed passed wait
     $\forall s \in \text{set-mset} (\text{wait} - \{\#a\# \}) \cup \{a'\}. E a a' \wedge \neg \text{empty } a'. \exists s' \in$ 
    set-mset wait'.  $s \preceq s'$ 
  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  $\langle x \in \# \text{ wait} \rangle$  have  $x \in \text{set-mset} (\text{wait} - \{\#a\# \})$ 
        by (metis insert-DiffM insert-noteq-member prems(1))
      with prems(2,4)  $\langle - \preceq x \rangle$  show ?thesis
        by (auto dest: trans)
  qed
done

```

```

lemma aux4:
  assumes worklist-inv-frontier passed  $\{\#\}$  reachable x worklist-start-subsumed
  passed  $\{\#\}$ 
     $\text{passed} \subseteq \text{Collect reachable}$ 
  shows  $\exists x' \in \text{passed}. x \preceq x'$ 
proof –
  from  $\langle \text{reachable } x \rangle$  have  $E^{**} a_0 x$  by (simp add: reachable-def)
  from assms(3) obtain  $b$  where  $a_0 \preceq b$   $b \in \text{passed}$  unfolding work-

```

list-start-subsumed-def **by** *auto*
have $\exists x'. \exists x''. E^{**} b x' \wedge x \preceq x' \wedge x' \preceq x'' \wedge x'' \in \text{passed}$ **if**
 $E^{**} a x a \preceq b \quad b \preceq b' \quad b' \in \text{passed}$
 $\text{reachable } a \text{ reachable } b$ **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.prem*s *step.hyps* **have** *empty x* **by** – (*rule empty-E-star, auto*)
with *step.prem*s **show** *?thesis* **by** (*auto intro: empty-subsumes*)
next
case *False*
with $\langle E a a1 \rangle \langle a \preceq b \rangle \langle \text{reachable } a \rangle \langle \text{reachable } b \rangle$ **obtain** *b1* **where**
 $E b b1 a1 \preceq b1$
using *mono* **by** *blast*
show *?thesis*
proof (*cases empty b1*)
case *True*
with *empty-mono* $\langle a1 \preceq b1 \rangle$ **have** *empty a1* **by** *blast*
with *step.prem*s *step.hyps* **have** *empty x* **by** – (*rule empty-E-star, auto simp: reachable-def*)
with *step.prem*s **show** *?thesis* **by** (*auto intro: empty-subsumes*)
next
case *False*
from $\langle E b b1 \rangle \langle a1 \preceq b1 \rangle$ **obtain** *b1'* **where** $E b' b1' b1 \preceq b1'$
using $\langle \neg \text{empty } a \rangle$ *empty-mono* *assms(4)* *mono* *step.prem*s **by**
blast
from *empty-mono*[*OF* $\langle \neg \text{empty } b1 \rangle \langle b1 \preceq b1' \rangle$] **have** $\neg \text{empty } b1'$
by *auto*
with $\langle E b' b1' \rangle \langle b' \in \text{passed} \rangle$ *assms(1)* **obtain** *b1''* **where** $b1'' \in \text{passed}$
 $b1' \preceq b1''$
unfolding *worklist-inv-frontier-def* **by** *auto*
with $\langle b1 \preceq - \rangle$ **have** $b1 \preceq b1''$ **using** *trans* **by** *blast*
with *step.IH*[*OF* $\langle a1 \preceq b1 \rangle$ *this* $\langle b1'' \in \text{passed} \rangle$] $\langle \text{reachable } a \rangle \langle E a a1 \rangle \langle \text{reachable } b \rangle \langle E b b1 \rangle$
obtain $x' x''$ **where**
 $E^{**} b1 x' x \preceq x' x' \preceq x'' x'' \in \text{passed}$
by *auto*
moreover from $\langle E b b1 \rangle \langle E^{**} b1 x' \rangle$ **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
 $set-mset\ aca \subseteq set-mset\ (aaa - \{\#aba\}) \cup Collect\ (E\ aba)$
 $set-mset\ aaa \subseteq Collect\ reachable$
 $x \in \#\ aca$
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* =
 $(\forall a \in passed. \forall a'. E\ a\ a' \wedge \neg empty\ a' \longrightarrow (\exists b' \in passed \cup set-mset\ wait. a' \preceq b'))$

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

definition *worklist-inv'* $\equiv \lambda (passed, wait, brk).$
 $worklist-inv\ (passed, wait, brk) \wedge (\forall a \in passed. \neg empty\ a) \wedge (\forall a \in set-mset\ wait. \neg empty\ a)$

definition *add-succ-spec'* *wait a* $\equiv SPEC\ (\lambda (wait', brk).$

(

```

if  $\exists a'. E a a' \wedge F a'$  then
  brk
else
   $\neg brk \wedge set-mset\ wait' \subseteq set-mset\ wait \cup \{a'. E a a'\} \wedge$ 
   $(\forall s \in set-mset\ wait \cup \{a'. E a a' \wedge \neg empty\ a'\}. \exists s' \in set-mset$ 
wait'.  $s \preceq s'$ )
)  $\wedge (\forall s \in set-mset\ wait'. \neg empty\ s)$ 
)

```

definition *worklist-algo'* **where**

```

worklist-algo' = do
{
  if  $F a_0$  then RETURN True
  else do {
    let passed = {};
    let wait = {#a_0#};
     $(passed, wait, brk) \leftarrow WHILEIT\ worklist-inv' (\lambda (passed, wait, brk).$ 
 $\neg brk \wedge wait \neq \{\#\})$ 
     $(\lambda (passed, wait, brk). do$ 
    {
       $(a, wait) \leftarrow take-from-mset\ wait;$ 
      ASSERT (reachable a);
      if  $(\exists a' \in passed. a \preceq a')$  then 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'' \equiv

if empty a₀ then RETURN False else worklist-algo'

end — Search Space'' Defs

context *Search-Space''-finite-strict*

begin

lemma *worklist-algo''-correct*:

worklist-algo'' \leq SPEC (λ brk. brk \longleftrightarrow F-reachable)

proof (*cases empty a₀*)

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 $\langle \neg$ empty \rightarrow)*

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**

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

lemma *take-from-list-correct*:

assumes $s \neq []$

shows $take\text{-from-list } s \leq \text{SPEC } (\lambda (x, s'). s = x \# s')$

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* $\equiv \lambda (passed, wait, brk).$

$passed \subseteq \text{Collect reachable} \wedge$

$(brk \longrightarrow (\exists f. reachable f \wedge F f)) \wedge$

$(\neg brk \longrightarrow$

$worklist\text{-inv-frontier-list passed wait}$

$\wedge (\forall a \in passed \cup set wait. \neg F a)$

$\wedge start\text{-subsumed-list passed wait}$

$\wedge set\ wait \subseteq \text{Collect reachable}$)

$\wedge (\forall a \in passed. \neg empty a) \wedge (\forall a \in set\ wait. \neg empty a)$

definition *add-succ-spec-list wait a* $\equiv \text{SPEC } (\lambda(wait', brk).$

$($
 $if \exists a'. E a a' \wedge F a' then$
 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)  $\leftarrow$  WHILEIT worklist-inv-list ( $\lambda$  (passed, wait,
brk).  $\neg$  brk  $\wedge$  wait  $\neq$  [])
            ( $\lambda$  (passed, wait, brk). do
              {
                (a, wait)  $\leftarrow$  take-from-list wait;
                ASSERT (reachable a);
                if ( $\exists a' \in$  passed.  $a \leq a'$ ) then RETURN (passed, wait, brk)
              }
            )
          else
            do
              {
                (wait, brk)  $\leftarrow$  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

$\neg F a_0 \neg F a_0$

$(x, x') \in \{((passed, wait, brk), (passed', wait', brk')). passed = passed' \wedge$
 $mset\ wait = wait' \wedge brk = brk'\}$

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)
(λa (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'))))
  )
  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 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)
      }
    )
    (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]*: $worklist-algo1' \leq \Downarrow Id$ *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 a0 then RETURN False else worklist-algo1'

```

```

lemma worklist-algo1-ref[refine]: worklist-algo1  $\leq$   $\Downarrow$ Id worklist-algo''
proof –
  have worklist-algo1'  $\leq$  worklist-algo' if  $\neg$  empty a0
  proof –
    note worklist-algo1-list-ref
    also note worklist-algo-list-ref
    finally show worklist-algo1'  $\leq$  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 \equiv
nfoldli (*succs a*) ($\lambda(-,brk). \neg brk$)
($\lambda a (wait,brk).$
if empty a then RETURN (wait, False)
else if F' a then RETURN (wait, True)
else RETURN (
 if $\exists x \in set\ wait. a \sqsubseteq x \wedge \neg x \sqsubseteq a$
 then [*x* \leftarrow *wait. \neg x \sqsubseteq a*]
 else *a # [x* \leftarrow *wait. \neg x \sqsubseteq a], False*)
)
(*wait, False*)

definition
filter-insert-wait wait a \equiv
if $\exists x \in set\ wait. a \sqsubseteq x \wedge \neg x \sqsubseteq a$
then [*x* \leftarrow *wait. \neg x \sqsubseteq a*]
else *a # [x* \leftarrow *wait. \neg x \sqsubseteq a]*

end

context *Worklist3*
begin

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

```

filter-insert-wait wait a = (
  let
    (f, xs) =
      fold ( $\lambda x (f, xs). \text{if } x \sqsubseteq a \text{ then } (f, xs) \text{ else } (f \vee a \sqsubseteq x, x \# xs)$ )
wait (False, [])
    in
      if f then rev xs else a # rev xs
  )

```

proof –
have
 $fold (\lambda x (f, xs). \text{if } x \sqsubseteq a \text{ then } (f, xs) \text{ else } (f \vee a \sqsubseteq x, x \# xs)) \text{ wait } (f, xs)$
 $= (f \vee (\exists x \in \text{set wait. } a \sqsubseteq x \wedge \neg x \sqsubseteq a), \text{rev } [x \leftarrow \text{wait. } \neg x \sqsubseteq a] @ xs)$ **for** $f \text{ xs}$
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  $\equiv$ 
  nfoldli (succs a) ( $\lambda(-, brk). \neg brk$ )
  ( $\lambda a (wait, brk).$ 
    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  $\leq \Downarrow (Id \times_r \text{bool-rel}) (add-succ1' \text{wait}' a')$ 
if ( $\text{wait}, \text{wait}' \in Id$ ) ( $a, a' \in Id$ )
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 a₀
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 a₀ 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 ← xs. P x] = rev (foldli xs (λ x. True) (λ 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**

sempref-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

sempref-register *PR-CONST* a_0 *PR-CONST* F' *PR-CONST* (\trianglelefteq) *PR-CONST*
succs PR-CONST empty

lemma [*def-pat-rules*]:

$a_0 \equiv UNPROTECT\ a_0\ F' \equiv UNPROTECT\ F' (\trianglelefteq) \equiv UNPROTECT$

(\triangleleft) *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 \implies 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 \sqsupset - lso-fold-custom-empty*)
unfolding *fold-lso-bex*
unfolding *take-from-list-alt-def*
apply (*rewrite in [a₀] HOL-list.fold-custom-empty*)
by *sepref*

concrete-definition (**in** *-*) *worklist-algo2-impl*

for *Lei a₀i Fi succsi emptyi*

uses *Worklist4-Impl.worklist-algo2-impl.refine-raw* **is** $(\text{uncurry0 } ?f, -) \in -$
end — *Worklist4 Impl*

context *Worklist4-Impl-finite-strict*
begin

lemma *worklist-algo2-impl-hnr-F-reachable*:

$(\text{uncurry0 } (\text{worklist-algo2-impl } \text{Lei } a_0 i \text{ } Fi \text{ } \text{succsi } \text{emptyi}), \text{uncurry0 } (\text{RETURN } F\text{-reachable}))$

$\in \text{unit-assn}^k \rightarrow_a \text{bool-assn}$

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* \equiv *op-F-reachable* **by** *simp*

lemma *hnr-op-F-reachable*:

assumes *GEN-ALGO* $a_0 i$ $(\lambda a_0 i. (\text{uncurry0 } a_0 i, \text{uncurry0 } (\text{RETURN } a_0))) \in \text{unit-assn}^k \rightarrow_a A$

assumes *GEN-ALGO* Fi $(\lambda Fi. (Fi, \text{RETURN } o F')) \in A^k \rightarrow_a \text{bool-assn}$

assumes *GEN-ALGO* Lei $(\lambda Lei. (\text{uncurry } Lei, \text{uncurry } (\text{RETURN } oo (\leq)))) \in A^k *_a A^k \rightarrow_a \text{bool-assn}$

assumes *GEN-ALGO* succsi $(\lambda \text{succsi}. (\text{succsi}, \text{RETURN } o \text{succs})) \in A^k \rightarrow_a \text{list-assn } A$

assumes *GEN-ALGO* emptyi $(\lambda Fi. (Fi, \text{RETURN } o \text{empty})) \in A^k \rightarrow_a \text{bool-assn}$

shows

$(\text{uncurry0 } (\text{worklist-algo2-impl } \text{Lei } a_0 i \text{ } Fi \text{ } \text{succsi } \text{emptyi}), \text{uncurry0 } (\text{RETURN } (\text{PR-CONST } \text{op-F-reachable})))$

$\in \text{unit-assn}^k \rightarrow_a \text{bool-assn}$

proof –

from *assms interpret Worklist4-Impl* $E a_0 F (\leq) \text{succs empty } (\leq) F' A \text{succsi } a_0 i \text{ } Fi \text{ } Lei \text{emptyi}$

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* ($- \rightarrow -$ [100, 100] 40)

notation *G.E'* ($- \rightarrow -$ [100, 100] 40)

no-notation *reaches* ($- \rightarrow^* -$ [100, 100] 40)

notation *G.G'.reaches* ($- \rightarrow^* -$ [100, 100] 40)

no-notation *reaches1* ($- \rightarrow^+ -$ [100, 100] 40)

notation *G.G'.reaches1* ($- \rightarrow^+ -$ [100, 100] 40)

Plain set membership is also an option.

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

definition *dfs* :: $'a \text{ set} \Rightarrow (\text{bool} \times 'a \text{ set}) \text{ nres}$ **where**

dfs P \equiv *do* {

$(P, ST, r) \leftarrow \text{RECT } (\lambda \text{dfs } (P, ST, v).$

do {

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 $ST = v \# ST;$

$(P, ST', r) \leftarrow$

FOREACHcd $\{v'. v \rightarrow v'\} (\lambda(-, -, b). \neg b) (\lambda v' (P, ST, -). \text{dfs}$

(P, ST, v') $(P, ST, \text{False});$

ASSERT $(ST' = ST);$

let $ST = \text{tl } ST';$

let $P = \text{insert } v P;$

```

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

```

definition *liveness-compatible* **where** *liveness-compatible* $P \equiv$
 $(\forall x x' y. x \rightarrow y \wedge x' \in P \wedge x \preceq x' \longrightarrow (\exists y' \in P. y \preceq y')) \wedge$
 $(\forall s' \in P. \forall s. s \preceq s' \wedge V s \longrightarrow$
 $\neg (\lambda x y. x \rightarrow y \wedge (\exists x' \in P. \exists y' \in P. x \preceq x' \wedge y \preceq y'))^{++} s s)$

definition *dfs-spec* \equiv
 $SPEC (\lambda (r, P).$
 $(r \longrightarrow (\exists x. a_0 \rightarrow^* x \wedge x \rightarrow^+ x))$
 $\wedge (\neg r \longrightarrow \neg (\exists x. a_0 \rightarrow^* x \wedge x \rightarrow^+ x))$
 $\wedge \textit{liveness-compatible } P \wedge P \subseteq \{x. V x\}$
 $)$
 $)$

end

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

lemma *check-loop-loop*: $\exists v' \in \textit{set } ST. v' \preceq v$ **if** *check-loop* v *ST*
using *that* **unfolding** *check-loop-def* **by** *blast*

lemma *check-loop-no-loop*: $v \notin \textit{set } ST$ **if** $\neg \textit{check-loop } v$ *ST*
using *that* **unfolding** *check-loop-def* **by** *blast*

lemma *mono*:
 $a \preceq b \implies a \rightarrow a' \implies V b \implies \exists b'. b \rightarrow b' \wedge a' \preceq b'$
by (*auto dest: mono simp: G.E'-def*)

context

fixes $P :: 'a \textit{ set}$ **and** $E1 E2 :: 'a \Rightarrow 'a \Rightarrow \textit{bool}$ **and** $v :: 'a$
defines [*simp*]: $E1 \equiv \lambda x y. x \rightarrow y \wedge (\exists x' \in P. x \preceq x') \wedge (\exists x \in P. y \preceq x)$
defines [*simp*]: $E2 \equiv \lambda x y. x \rightarrow y \wedge (x \preceq v \vee (\exists xa \in P. x \preceq xa)) \wedge (y \preceq v \vee (\exists x \in P. y \preceq x))$

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

$\forall s\ V\ v\ s\ \preceq\ v$

liveness-compatible P

$\forall va.\ v \rightarrow va \longrightarrow (\exists x \in P.\ va \preceq x)$

$E2^{++}\ s\ s$

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** $a \rightarrow b$ $a \preceq a'$ $a' \in P\ \forall a$ **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** $G'.reaches\ a\ b$ $a \preceq a'$ $a' \in P\ \forall a$ **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 $\langle V\ a \rangle\ \langle G.reaches\ a\ b \rangle$ **have** $V\ b$

including *subgraph-automation by blast*

from $\langle G.reaches\ a\ b \rangle\ \langle a \preceq a' \rangle\ \langle a' \in P \rangle$ **obtain** b' **where** $b' \in P\ b \preceq b'$

by (*fastforce dest: 1*)
 with $\langle E2\ b\ c \rangle \langle V\ b \rangle$ have $E1\ b\ c$
 by (*fastforce intro: 2*)
 with $\langle G.reaches\ a\ b \rangle$ show *?case*
 by *blast*
qed
 from $\langle E2^{++}\ s\ s \rangle$ obtain x where $E2\ s\ x\ G'.reaches\ x\ s$
 by *blast*
 then have $s \rightarrow x$
 by *auto*
 with $\langle s \preceq v \rangle \langle V\ s \rangle \langle V\ v \rangle$ obtain x' where $v \rightarrow x'\ x \preceq x'$
 by (*auto dest: mono*)
 with *assms(5)* obtain x'' where $x \preceq x''\ x'' \in P$
 by (*auto intro: order-trans*)
 from $4[OF\ \langle G'.reaches\ x\ s \rangle \langle x \preceq x'' \rangle \langle x'' \in P \rangle] \langle s \rightarrow x \rangle \langle V\ s \rangle$ have
 $G.reaches\ x\ s$
 including *subgraph-automation* by *blast*
 with $\langle x \preceq x'' \rangle \langle x'' \in P \rangle$ obtain s' where $s \preceq s'\ s' \in P$
 by (*blast dest: 1*)
 with $\langle s \rightarrow x \rangle \langle x \preceq x'' \rangle \langle x'' \in P \rangle$ have $E1\ s\ x$
 by *auto*
 with $\langle G.reaches\ x\ s \rangle$ have $G.reaches1\ s\ s$
 by *blast*
 with *assms(4)* $\langle s' \in P \rangle \langle s \preceq s' \rangle \langle V\ s \rangle$ show *False*
 unfolding *liveness-compatible-def* by *auto*
qed

lemma *liveness-compatible-extend'*:

assumes

$V\ s\ V\ v\ s \preceq s'\ s' \in P$

$\forall va. v \rightarrow va \longrightarrow (\exists x \in P. va \preceq x)$

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  $\neg$  E1 c d G'.reaches d s
  by auto
from  $\langle E2 c d \rangle \langle \neg E1 c d \rangle$  have  $c \preceq v \vee d \preceq v$ 
  by auto
with  $\langle V s \rangle$  ** obtain v' where G'.reaches1 v' v' v'  $\preceq$  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  $\rightarrow^*$  x x  $\rightarrow^+$  x a  $\preceq$  s s  $\in$  P
  shows False
proof -
  include graph-automation-aggressive
  have *:  $\exists y \in P. x \preceq y$  if G.G'.reaches a x for x
    using that assms unfolding liveness-compatible-def by induction auto
  have **:
     $(\lambda x y. x \rightarrow y \wedge (\exists x' \in P. x \preceq x') \wedge (\exists x \in P. y \preceq x))^{++} a' b \longleftrightarrow a' \rightarrow^+ b$ 
  if a  $\rightarrow^*$  a' for a' b
  apply standard
  subgoal
    by (induction rule: tranclp-induct; blast)
  subgoal
    using  $\langle a \rightarrow^* a' \rangle$ 
    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* $P \forall va. v \rightarrow va \longrightarrow (\exists x \in P. va \preceq x)$

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* (\preceq) (\prec) $E V$
by *standard* (*drule mono, auto simp: Subgraph-Node-Defs.E'-def*)

lemma *pre-cycle-cycle*:

$(\exists x x'. a_0 \rightarrow^* x \wedge x \rightarrow^+ x' \wedge x \preceq x') \longleftrightarrow (\exists x. a_0 \rightarrow^* x \wedge x \rightarrow^+ x)$

by (*meson G.E'-def G.G'.reaches1-reaches-iff1 subsumption.pre-cycle-cycle-reachable finite-V*)

lemma *reachable-alt*:

$V v$ **if** $V a_0 a_0 \rightarrow^* v$

using *that(2,1)* **by** *cases* (*auto simp: G.E'-def*)

lemma *dfs-correct*:

dfs P \leq *dfs-spec* **if** $V a_0$ *liveness-compatible P* $P \subseteq \{x. V x\}$

proof –

define *rpre* **where** $rpre \equiv \lambda(P, ST, v).$

$a_0 \rightarrow^* v$

$\wedge V v$

$\wedge P \subseteq \{x. V x\}$

$\wedge \text{set } ST \subseteq \{x. a_0 \rightarrow^* x\}$

$\wedge \text{set } ST \subseteq \{x. V x\}$

\wedge *liveness-compatible P*

$\wedge (\forall s \in \text{set } ST. s \rightarrow^+ v)$

\wedge *distinct ST*

define *rpost* **where** $rpost \equiv \lambda(P, ST, v) (P', ST', r).$

$(r \longrightarrow (\exists x x'. a_0 \rightarrow^* x \wedge x \rightarrow^+ x' \wedge x \preceq x') \wedge ST' = ST) \wedge$

```

( $\neg$   $r \longrightarrow$ 
   $P \subseteq P'$ 
   $\wedge P' \subseteq \{x. V x\}$ 
   $\wedge \text{set } ST \subseteq \{x. a_0 \rightarrow^* x\}$ 
   $\wedge ST' = ST$ 
   $\wedge (\forall s \in \text{set } ST. s \rightarrow^* v)$ 
   $\wedge \text{liveness-compatible } P'$ 
   $\wedge (\exists v' \in P'. v \preceq v')$ 
   $\wedge \text{distinct } ST$ 
)

```

define *inv* **where** $inv \equiv \lambda P ST v (- :: 'a \text{ set}) \text{ it } (P', ST', r).$
 $(r \longrightarrow (\exists x x'. a_0 \rightarrow^* x \wedge x \rightarrow^+ x' \wedge x \preceq x') \wedge ST' = ST) \wedge$
 $(\neg r \longrightarrow$
 $P \subseteq P'$
 $\wedge P' \subseteq \{x. V x\}$
 $\wedge \text{set } ST \subseteq \{x. a_0 \rightarrow^* x\}$
 $\wedge ST' = ST$
 $\wedge (\forall s \in \text{set } ST. s \rightarrow^* v)$
 $\wedge \text{set } ST \subseteq \{x. V x\}$
 $\wedge \text{liveness-compatible } P'$
 $\wedge (\forall v \in \{v'. v \rightarrow v'\} - \text{it. } (\exists v' \in P'. v \preceq v'))$
 $\wedge \text{distinct } ST$
 $)$

define *Termination* $:: (('a \text{ set} \times 'a \text{ list} \times 'a) \times 'a \text{ set} \times 'a \text{ list} \times 'a) \text{ set}$
where

$Termination = \text{inv-image } (\text{finite-psupset } \{x. V x\}) (\lambda (a,b,c). \text{set } b)$

have *rpre-init*: $rpre (P, [], a_0)$

unfolding *rpre-def* **using** $\langle V a_0 \rangle \langle \text{liveness-compatible } P \rangle \langle P \subseteq \rightarrow \rangle$ **by**
auto

have *wf*: $wf \text{ Termination}$

unfolding *Termination-def* **by** $(\text{blast intro: finite-V})$

have *successors-finite*: $\text{finite } \{v'. v \rightarrow v'\}$ **for** v

using *finite-V* **unfolding** *G.E'-def* **by** *auto*

show *?thesis*

unfolding *dfs-def* *dfs-spec-def*

apply $(\text{refine-rcg } \text{refine-vcg})$

```

apply (rule Orderings.order.trans)
apply(rule RECT-rule[where
  pre = rpre and
  V = Termination and
  M = λx. 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  $I = \text{inv } P (v \# ST) v$ ]
)
apply clarsimp-all

```

```

subgoal
  by (auto intro: successors-finite)

```

```

subgoal
  using  $\langle V a_0 \rangle$ 
  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  $\langle V a_0 \rangle$  have  $v \rightarrow v'$ 
      by auto
    with prems show ?thesis
      by (auto intro: G.E'-V2)
  qed
done

```

```

subgoal
  using  $\langle V a_0 \rangle$  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  $\langle V a_0 \rangle$  by (subst rpost-def, subst (asm) inv-def, auto)

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

subgoal for  $P' ST'$ 
  using  $\langle V a_0 \rangle$ 
  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 ( $\lambda$ 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:  $V a \implies \text{set} (\text{succs } a) = \{x. a \rightarrow x\}$ 
  assumes finite-V:  $\text{finite } \{a. V a\}$ 
begin

```

— The following complications only arise because we add the assertion in this refinement step.

```

lemma succs-ref[refine]:
   $(\text{succs } a, \text{succs } b) \in \langle \text{Id} \rangle \text{list-rel}$  if  $(a, b) \in \text{Id}$ 
  using that by auto

```

```

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

```

```

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

```

```

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

```

```

(dfs x^)  $\implies$ 
  (x, x')  $\in$  Id  $\times_r$  br id ( $\lambda xs. \text{set } xs \subseteq \{x. V x\}$ )  $\times_r$  br id V  $\implies$ 
  x2 = (x1a, x2a)  $\implies$ 
  x' = (x1, x2)  $\implies$ 
  x2b = (x1c, x2c)  $\implies$ 
  x = (x1b, x2b)  $\implies$ 
  nfoldli (succs x2c) ( $\lambda(-, -, b). \neg b$ ) ( $\lambda v' (P, ST, -). \text{dfs}' (P, ST, v')$ ) (x1b,
x2c # x1c, False)
   $\leq \Downarrow$  (Id  $\times_r$  br id ( $\lambda xs. \text{set } xs \subseteq \{x. V x\}$ )  $\times_r$  bool-rel)
    (FOREACHcd {v'. x2a  $\rightarrow$  v'} ( $\lambda(-, -, b). \neg b$ )
      ( $\lambda v' (P, ST, -). \text{dfs}$  (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  $\langle$ auto simp: br-def $\rangle$ )
apply (erule nfoldli-refine)
apply (solves  $\langle$ auto simp: br-def $\rangle$ )+
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 a0
  using that unfolding dfs1-def dfs-def
  apply refine-rcg
    apply (solves  $\langle$ auto simp: br-def $\rangle$ )+
    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 ⇒ 'v ⇒ bool (**infix** ≤ 50)
begin

sublocale *Search-Space-Key-Defs* .

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

definition *insert-map-set* v S ≡

let k = key v; S' = (case S k of Some S ⇒ S | None ⇒ {}) in S(k ↦
(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 ↦ v #
S')

definition *check-subsumption-map-set* v S ≡

let k = key v; S' = (case S k of Some S ⇒ S | None ⇒ {}) in (∃ x ∈ S'.
v ≤ x)

definition *check-subsumption-map-list* v S ≡

let k = key v; S' = (case S k of Some S ⇒ S | None ⇒ []) in (∃ x ∈ set
S'. x ≤ v)

definition *pop-map-list* v S ≡

let k = key v; S' = (case S k of Some S ⇒ tl S | None ⇒ []) in S(k ↦ S')

definition *dfs-map* :: ('k → 'v set) ⇒ (bool × ('k → 'v set)) nres **where**

dfs-map P ≡ do {
(P,ST,r) ← RECT (λdfs (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) (λv' (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]:  $a \trianglelefteq b \implies \text{key } a = \text{key } b$ 
  assumes V-subsumes':  $V a \implies a \preceq b \longleftrightarrow a \trianglelefteq b$ 

```

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

```

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

lemma *empty-map-list-rel*:

```

  (Map.empty, []) ∈ map-list-rel
  unfolding map-list-rel-def irrefl-trans-on-def by auto

```

lemma *rel-start[refine]*:

```

  ((P, Map.empty, a0), P', [], a0) ∈ map-set-rel ×r map-list-rel ×r Id if
  (P, P') ∈ map-set-rel
  using that unfolding map-set-rel-def by (auto intro: empty-map-list-rel)

```

lemma *refine-True*:

```

  (x1b, x1) ∈ map-set-rel  $\implies$  (x1c, x1a) ∈ map-list-rel

```

$\implies ((x1b, x1c, True), x1, x1a, True) \in \text{map-set-rel} \times_r \text{map-list-rel} \times_r \text{Id}$
by *simp*

lemma *check-subsumption-ref[refine]*:

$V x2a \implies (x1b, x1) \in \text{map-set-rel} \implies \text{check-subsumption-map-set } x2a$
 $x1b = (\exists x \in x1. x2a \preceq x)$

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]*:

$\text{set } xs \subseteq \{x. V x\} \implies (m, xs) \in \text{map-list-rel}$

$\implies \text{check-subsumption-map-list } x m = \text{check-loop } x xs$

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*:

$x \notin \text{set } xs \text{ if } \neg \text{check-loop-list } x xs$

using that **unfolding** *check-loop-list-def* **by** *auto*

lemma *insert-ref[refine]*:

$(x1b, x1) \in \text{map-set-rel} \implies$

$(x1c, x1a) \in \langle \text{Id} \rangle \text{list-set-rel} \implies$

$\neg \text{check-loop-list } x2a x1c \implies$

$((x1b, x2a \# x1c, False), x1, \text{insert } x2a x1a, False) \in \text{map-set-rel} \times_r$
 $\text{list-set-hd-rel } x2a \times_r \text{Id}$

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*:

$(m, S) \in \text{map-set-rel} \implies (\text{insert-map-set } x m, \text{insert } x S) \in \text{map-set-rel}$

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 $(m, xs) \in \text{map-list-rel } x \in \text{set } xs$
 obtains xs' where $x \in \text{set } xs' \ m \ (\text{key } x) = \text{Some } xs'$
 using *assms* unfolding *map-list-rel-def ran-def* by (*auto 4 3 dest: sym*)

lemma *map-list-rel-memI*:
 $(m, xs) \in \text{map-list-rel} \implies m \ k = \text{Some } xs' \implies x' \in \text{set } xs' \implies x' \in \text{set } xs$
 unfolding *map-list-rel-def ran-def* by *auto*

lemma *map-list-rel-grouped-by-key*:
 $x' \in \text{set } xs' \implies (m, xs) \in \text{map-list-rel} \implies m \ k = \text{Some } xs' \implies \text{key } x' = k$
 unfolding *map-list-rel-def* by *auto*

lemma *map-list-rel-not-memI*:
 $k \neq \text{key } x \implies m \ k = \text{Some } xs' \implies (m, xs) \in \text{map-list-rel} \implies x \notin \text{set } xs'$
 unfolding *map-list-rel-def* by *auto*

lemma *map-list-rel-not-memI2*:
 $x \notin \text{set } xs' \ \text{if } m \ a = \text{Some } xs' \ (m, xs) \in \text{map-list-rel } x \notin \text{set } xs$
 using *that* unfolding *map-list-rel-def ran-def* by *auto*

lemma *push-map-list-ref*:
 $x \notin \text{set } xs \implies (m, xs) \in \text{map-list-rel} \implies (\text{push-map-list } x \ m, x \ \# \ xs) \in \text{map-list-rel}$
 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  $A: x \notin \text{set } xs \ (m, xs) \in \text{map-list-rel}$ 
  then obtain  $R$  where *:
     $x \notin \text{set } xs$ 
     $(\bigcup x \in \text{ran } m. \text{set } x) = \text{set } xs$ 
     $\forall k x. m k = \text{Some } x \longrightarrow (\forall v \in \text{set } x. \text{key } v = k)$ 
    distinct  $xs$ 
     $\forall k x. m k = \text{Some } x \longrightarrow \text{sorted-wrt } R x$ 
    sorted-wrt  $R xs$ 
    irrefl-trans-on  $R (\text{set } xs)$ 
  unfolding map-list-rel-def by auto
  have **: sorted-wrt  $(\lambda a b. a = x \wedge b \neq x \vee a \neq x \wedge b \neq x \wedge R a b) xs$ 
if
  sorted-wrt  $R xs$   $x \notin \text{set } xs$  for  $xs$ 
  using that by (induction  $xs$ ) auto

show ?thesis
  apply (inst-existentials  $\lambda a b. a = x \wedge b \neq x \vee a \neq x \wedge b \neq x \wedge R a b$ )
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*]:
 $(x1b, x1) \in \text{map-set-rel} \implies$
 $(x1c, x1a) \in \text{map-set-rel} \implies$
 $\neg \text{check-subsumption}' x2a x1c \implies$
 $((x1b, \text{insert-map-set } x2a x1c, \text{False}), x1, \text{insert } x2a x1a, \text{False}) \in \text{map-set-rel}$
 $\times_r \text{map-set-rel} \times_r \text{Id}$
by (*auto intro: insert-map-set-ref*)

lemma *map-list-rel-check-subsumption-map-list*:
 $\text{set } xs \subseteq \{x. V x\} \implies (m, xs) \in \text{map-list-rel} \implies \neg \text{check-subsumption-map-list}$
 $x m \implies x \notin \text{set } xs$
unfolding *check-subsumption-map-list-def* **by** (*auto 4 3 elim!: map-list-rel-memD*
dest: V-subsumes')

lemma *push-map-list-ref*[*refine*]:
 $\text{set } x1a \subseteq \{x. V x\} \implies$
 $(x1b, x1) \in \text{map-set-rel} \implies$
 $(x1c, x1a) \in \text{map-list-rel} \implies$
 $\neg \text{check-subsumption-map-list } x2a x1c \implies$
 $((x1b, \text{push-map-list } x2a x1c, \text{False}), x1, x2a \# x1a, \text{False}) \in \text{map-set-rel}$
 $\times_r \text{map-list-rel} \times_r \text{Id}$
by (*auto intro: push-map-list-ref dest: map-list-rel-check-subsumption-map-list*)

lemma *sorted-wrt-tl*:
 $\text{sorted-wrt } R (tl xs) \text{ if } \text{sorted-wrt } R xs$
using that by (*induction xs*) *auto*

lemma *irrefl-trans-on-mono*:
 $\text{irrefl-trans-on } R S \text{ if } \text{irrefl-trans-on } R S' S \subseteq S'$
using that unfolding *irrefl-trans-on-def* **by** *blast*

lemma *pop-map-list-ref*[*refine*]:
 $(\text{pop-map-list } v m, S) \in \text{map-list-rel} \text{ if } (m, v \# S) \in \text{map-list-rel}$
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 *:  $x \in \text{set } S$  if  $x \in \text{set } (tl\ xs)\ m\ (\text{key } v) = \text{Some } xs$ 
  proof -
    from prems have sorted-wrt  $R\ xs$ 
      by auto
    from  $\langle m\ (\text{key } v) = - \rangle$  have  $v \in \text{set } xs$ 
      using map-list-rel-memD[OF  $\langle (m, v \# S) \in \text{map-list-rel} \rangle$ , of  $v$ ] by
auto
    have *:  $R\ v\ x$  if  $x \in \text{set } xs\ v \neq x$  for  $x$ 
    using that prems by - (drule map-list-rel-memI[OF  $\langle - \in \text{map-list-rel} \rangle$ ],
auto)
    have  $v \neq x$ 
    proof (rule ccontr)
      assume  $\neg v \neq x$ 
      with that obtain  $a\ as\ bs$  where
         $xs = a \# as @ v \# bs$ 
      unfolding in-set-conv-decomp by (cases  $xs$ ) auto
      with  $\langle \text{sorted-wrt } R\ xs \rangle$  have  $a \in \text{set } xs\ R\ a\ v\ a \in \text{insert } v\ (\text{set } S)$ 
        using map-list-rel-memI[OF  $\langle - \in \text{map-list-rel} \rangle\ \langle m - = - \rangle$ , of  $a$ ]
        by auto
      with *  $\langle \text{irrefl-trans-on } - \ - \rangle$  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  $\langle - \in \text{map-list-rel} \rangle$ ])
    qed
    moreover have  $x \in \text{set } S$  if  $m\ a = \text{Some } xs'\ a \neq \text{key } v$  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  $\langle - \in \text{map-list-rel} \rangle$ , of  $x$ ]  $\langle x \in - \rangle$  obtain
 $xs'$  where  $xs'$ :
     $x \in \text{set } xs'\ m\ (\text{key } x) = \text{Some } xs'$ 
    by auto
  show ?thesis
  proof (cases  $\text{key } x = \text{key } v$ )
    case True
      with prems  $xs'$  have [simp]:  $xs' = xs$ 
        by auto
      from prems have  $x \neq v$ 

```

```

    by auto
  have  $x \in \text{set } (\text{tl } xs)$ 
  proof (rule ccontr)
    assume  $x \notin \text{set } (\text{tl } xs)$ 
    with  $xs'$  have  $\text{hd } xs = x$ 
      by (cases  $xs$ ) auto
    from prems have  $\text{sorted-wrt } R \ xs$ 
      by auto
    from  $\langle m \text{ (key } v) = - \rangle$  have  $v \in \text{set } xs$ 
      using  $\text{map-list-rel-memD}[OF \ \langle (m, v \# S) \in \text{map-list-rel} \rangle, \text{ of } v]$ 
  by auto
  have  $*: R \ v \ x$  if  $x \in \text{set } xs \ v \neq x$  for  $x$ 
    using that prems by  $- \ (\text{drule } \text{map-list-rel-memI}[OF \ \langle - \in \text{map-list-rel} \rangle], \text{ auto})$ 
  with  $\langle x \in \text{set } xs' \ \langle x \neq v \rangle$  have  $R \ v \ x$ 
    by auto
  from  $\langle v \in \text{set } xs \ \langle \text{hd } xs = x \rangle \ \langle x \neq v \rangle$  have  $R \ x \ v$ 
    unfolding  $\text{in-set-conv-decomp}$ 
    apply  $\text{clarsimp}$ 
    using  $\langle \text{sorted-wrt } R \ xs \rangle$ 
    subgoal for  $ys$ 
      by (cases  $ys$ ) auto
    done
  with  $\langle R \ v \ x \rangle \ \langle \text{irrefl-trans-on } - \ - \rangle \ \langle x \in \text{set } S \rangle$  show  $\text{False}$ 
    unfolding  $\text{irrefl-trans-on-def}$  by blast
  qed
  with  $xs'$  show  $?thesis$ 
    unfolding  $\langle xs' = - \rangle \ \text{ran-def}$  by auto
  next
  case  $\text{False}$ 
  with  $xs'$  show  $?thesis$ 
    unfolding  $\text{ran-def}$  by auto
  qed
  subgoal
    by ( $\text{meson in-set-tlD}$ )
  subgoal for  $R$ 
    by ( $\text{blast intro: irrefl-trans-on-mono sorted-wrt-tl}$ )
  done
done

```

lemma tl-list-set-ref :

$$(m, S) \in \text{map-set-rel} \implies$$

$$(st, ST) \in \text{list-set-hd-rel } x \implies$$

```

    (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-recg
  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 a0i :: '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 a0i, uncurry0 (RETURN (PR-CONST a0))) ∈ unit-assnk
→a A
    (uncurry Lei, uncurry (RETURN oo PR-CONST (≤))) ∈ Ak *a Ak →a
bool-assn
    (succsi, RETURN o PR-CONST succs) ∈ Ak →a list-assn A
    (keyi, RETURN o PR-CONST key) ∈ Ak →a K
    (copyi, RETURN o COPY) ∈ Ak →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*]:
 $\langle \text{hms-assn } A \ m \ mi \rangle \text{ hms-delete } k \ mi \langle \text{hms-assn } A \ (m(k := \text{None})) \rangle_t$
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*:
 $(\text{uncurry } \text{hms-delete}, \text{uncurry } (\text{RETURN } oo \ \text{op-map-delete})) \in$
 $\text{id-assn}^k *_a (\text{hms-assn } A)^d \rightarrow_a \text{hms-assn } A$
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*:
 $\text{fold insert } xs \ S = \text{set } xs \cup \ S$
by (*induction xs arbitrary: S*) *auto*

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

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

sepref-register
 $\text{PR-CONST } a_0 \ \text{PR-CONST } (\trianglelefteq) \ \text{PR-CONST } \text{succs} \ \text{PR-CONST } \text{key}$

lemma [*def-pat-rules*]:
 $a_0 \equiv \text{UNPROTECT } a_0 \ (\trianglelefteq) \equiv \text{UNPROTECT } (\trianglelefteq) \ \text{succs} \equiv \text{UNPROTECT}$
 $\text{succs } \text{key} \equiv \text{UNPROTECT } \text{key}$
by *simp-all*

abbreviation *passed-assn* $\equiv \text{hm.hms-assn}' \ K \ (\text{lso-assn } A)$

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*} \rightarrow_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*)))
 \in *unit-assn*^{*k*} \rightarrow_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*) \leftarrow *dfs-map* *Map.empty*; *RETURN* *r*}) :: *unit-assn*^{*k*}
 \rightarrow_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*} \rightarrow_a *bool-assn* \times_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** $(\text{uncurry0 } ?f, -) \in -$

lemma (**in** *Liveness-Search-Space-Key*) *dfs-map-empty*:
 $\text{dfs-map } \text{Map.empty} \leq \Downarrow (\text{bool-rel} \times_r \text{map-set-rel}) \text{dfs-spec}$ **if** $V a_0$

proof $-$

have *liveness-compatible* $\{\}$
unfolding *liveness-compatible-def* **by** *auto*
have $(\text{Map.empty}, \{\}) \in \text{map-set-rel}$
unfolding *map-set-rel-def* **by** *auto*
note $\text{dfs-map-dfs-refine}[OF \text{this } \langle V a_0 \rangle]$
also note $\text{dfs-correct}[OF \langle V a_0 \rangle \langle \text{liveness-compatible } \{\} \rangle]$
finally show *?thesis*
by *auto*

qed

lemma (**in** *Liveness-Search-Space-Key*) *dfs-map-empty-correct*:
 $\text{do } \{(r, p) \leftarrow \text{dfs-map } \text{Map.empty}; \text{RETURN } r\} \leq \text{SPEC } (\lambda r. r \longleftrightarrow (\exists x. a_0 \rightarrow^* x \wedge x \rightarrow^+ x))$
if $V a_0$
supply $\text{dfs-map-empty}[OF \langle V a_0 \rangle, \text{THEN } \text{Orderings.order.trans}, \text{refine-vcg}]$
apply *refine-vcg*
unfolding *dfs-spec-def pw-le-iff* **by** $(\text{auto simp: refine-pw-simps})$

lemma *dfs-map-impl'-hnr*:

$(\text{uncurry0 } (\text{dfs-map-impl}' \text{succsi } a_0 i \text{Lei keyi copyi}),$
 $\text{uncurry0 } (\text{SPEC } (\lambda r. r = (\exists x. a_0 \rightarrow^* x \wedge x \rightarrow^+ x))))$
 $\in \text{unit-assn}^k \rightarrow_a \text{bool-assn}$ **if** $V a_0$

using

$\text{dfs-map-impl}'.\text{refine}[$
 $OF \text{Liveness-Search-Space-Key-Impl-axioms},$
 $\text{FCOMP } \text{dfs-map-empty-correct}[\text{THEN } \text{Id-SPEC-refine}, \text{THEN } \text{nres-relI}],$
 $OF \langle V a_0 \rangle$
 $].$

lemma *dfs-map-impl'-hoare-triple*:

$\langle \uparrow(V a_0) \rangle$

```

    dfs-map-impl' succsi a0i Lei keyi copyi
  <λr. ↑(r ←→ (∃ x. a0 →* x ∧ x →+ 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. ∃Axa. 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  $m k = \text{Some } v (x, k) \in \text{the-pure } K (v',$ 
 $v) \in \text{the-pure } V$ 
    apply atomize-elim
    by (meson map-rel-obtain2)
  with  $prems \langle \text{is-pure } K \rangle$  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  $(\text{hm-it-next-key}, \text{next-key}) \in (\text{hm.assn } K V)^k \rightarrow_a K$ 
  using  $assms$  by sepref-to-hoare (sep-auto intro!: hm-it-next-key-next-key-ax)

```

```

lemma hm-it-next-key-next-key':
   $(\text{hm-it-next-key}, \text{next-key}) \in (\text{hm.hms-assn } V)^k \rightarrow_a \text{id-assn}$ 
  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
     $P' = \text{is-hashmap } mi m' * \text{map-assn } V m mi * \uparrow(mi \neq \text{Map.empty})$ 
    and  $Q = \lambda x. \text{is-hashmap } mi m' * \uparrow(x \in \text{dom } mi) * \text{map-assn } V$ 
 $m mi]$ 
  )
  apply (solves  $\langle \text{sep-auto}; \text{unfold map-assn-def}; \text{auto simp: refine-pw-simps} \rangle$ ) +
  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*) \longleftrightarrow *m* \neq *Map.empty*
unfolding *next-key-def* **by** *auto*

context
fixes *mi m K*
assumes *map-rel*: (*mi*, *m*) \in $\langle K, Id \rangle$ *map-rel*
begin

private lemma *k-aux*:
assumes *k* \in *dom mi* (*mi*, *m*) \in $\langle K, Id \rangle$ *map-rel*
shows \exists *k'*. (*k*, *k'*) \in *K*
using *assms* **unfolding** *map-rel-def* **by** *auto*

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

private lemma *map-empty-iff*: *mi* \neq *Map.empty* \longleftrightarrow *m* \neq *Map.empty*
using *map-rel* **by** *auto*

private lemma *aux*:
assumes *RETURN k* \leq *next-key mi*
shows *RETURN (SOME k'. (k, k') \in K)* \leq *next-key m*
using *map-rel* *assms*
unfolding *next-key-def*
apply (*cases m* \neq *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* \leq *next-key mi* *nofail* (*next-key m*)
shows (*k*, *SOME k'. (k, k') \in K*) \in *K*
using *map-rel* *assms*
unfolding *next-key-def*
apply (*cases m* \neq *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  $(hm-it-next-key, next-key) \in (hm.hms-assn' K V)^k \rightarrow_a K$ 
  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  $\langle is-pure K \rangle$ 
    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
done

```

6.2 Computing the Range of a Map

definition *ran-of-map* where

```

ran-of-map m  $\equiv$  do
  {
    (xs, m)  $\leftarrow$  WHILEIT
      ( $\lambda (xs, m'). finite (dom m') \wedge ran m = ran m' \cup set xs$ ) ( $\lambda (-, m).$ 
Map.empty  $\neq m$ )
    ( $\lambda (xs, m).$  do
      {
        k  $\leftarrow$  next-key m;
        let (x, m) = op-map-extract k m;
        ASSERT (x  $\neq$  None);
        RETURN (the x # xs, m)
      }
    )
  }

```

```

      ([], m);
    RETURN xs
  }

```

context
begin

private definition

ran-of-map-var = (*inv-image* (*measure* (*card o dom*)) ($\lambda (a, b). b$))

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 \leq *SPEC* ($\lambda r. \text{set } r = \text{ran } m$) **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* \Rightarrow *'b nres*)

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

lemma (**in** *imp-map-is-empty*) *hms-empty-rule* [*sep-heap-rules*]:

$\langle \text{hms-assn } A \text{ } m \text{ } mi \rangle \text{ hms-is-empty } mi \langle \lambda r. \text{hms-assn } A \text{ } m \text{ } mi * \uparrow(r \longleftrightarrow m = \text{Map.empty}) \rangle_t$

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*]:

$(\text{hms-is-empty}, \text{RETURN } o \text{ op-map-is-empty}) \in (\text{hms-assn } A)^k \rightarrow_a \text{bool-assn}$

```

    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 $G'.steps (x \# ws @ y \# xs @ [y]) x \preceq x' V x V x'$

shows $\exists y' xs' ys'. y \preceq y' \wedge G'.steps (x' \# xs' @ y' \# ys' @ [y'])$

proof –

let $?n = \text{card } \{x. V x\} + 1$

let $?xs = y \# \text{concat } (\text{replicate } ?n (xs @ [y]))$

from *assms(1)* **have** $G'.steps (x \# ws @ [y]) G'.steps (y \# xs @ [y])$

by (*auto intro: Graph-Start-Defs.graphI-aggressive2*)

with $G'.steps\text{-replicate}[of\ y \# xs @ [y]\ ?n]$ **have** $G'.steps\ ?xs$

by *auto*

from *steps-mono*[*OF* $\langle G'.steps (x \# ws @ [y]) \rangle \langle x \preceq x' \rangle \langle V x \rangle \langle V x' \rangle$]

obtain ys **where**

$G'.steps (x' \# ys) \text{list-all2 } (\preceq) (ws @ [y]) ys$

by *auto*

then obtain $y' ws'$ **where** $G'.steps (x' \# ws' @ [y']) y \preceq y'$

unfolding *list-all2-append1 list-all2-Cons1* **by** *auto*

with $\langle G'.steps (x \# ws @ [y]) \rangle$ **have** $V y V y'$

subgoal

using *G'-steps-V-last assms(3)* **by** *fastforce*

using *G'-steps-V-last* $\langle G'.steps (x' \# ws' @ [y']) \rangle$ *assms(4)* **by** *fastforce*

with *steps-mono*[*OF* $\langle G'.steps\ ?xs \rangle \langle y \preceq y' \rangle$] **obtain** ys **where** ys :

```

    list-all2 ( $\preceq$ ) (concat (replicate ?n (xs @ [y]))) ys G'.steps (y' # ys)
  by auto
let ?ys = filter (( $\preceq$ ) y) ys
have length ?ys  $\geq$  ?n
  using list-all2-replicate-elem-filter[OF ys(1), of y]
  by auto
have set ?ys  $\subseteq$  set ys
  by auto
also have ...  $\subseteq$  {x. V x}
  using  $\langle G'.steps (y' \# -) \rangle \langle V y' \rangle$  by (auto simp: list-all-iff dest: G'-steps-V-all)
finally have  $\neg$  distinct ?ys
  using distinct-card[of ?ys]  $\langle - \rangle = ?n$ 
  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  $\langle G'.steps (y' \# -) \rangle$  have  $G'.steps (y' \# as' @ y'' \# bs' @ [y''])$ 
  unfolding  $\langle ys = - \rangle$  by (force intro: Graph-Start-Defs.graphI-aggressive2)
moreover from  $\langle ?ys = - \rangle$  have  $y \preceq y''$ 
proof -
  from  $\langle ?ys = - \rangle$  have  $y'' \in$  set ?ys by auto
  then show ?thesis by auto
qed
ultimately show ?thesis
  using  $\langle G'.steps (x' \# ws' @ [y']) \rangle$ 
  by (inst-existentials y'' ws' @ y' # as' bs';
fastforce intro: Graph-Start-Defs.graphI-aggressive1
)
qed

```

lemma reaches-cycle-mono:

```

  assumes  $G'.reaches\ x\ y\ y \rightarrow^+ y\ x \preceq x' \ V\ x\ V\ x'$ 
  shows  $\exists\ y'. y \preceq y' \wedge G'.reaches\ x'\ y' \wedge y' \rightarrow^+ y'$ 
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 **:  $\exists as. G'.steps (x \# as @ last list \# ys @ [last list])$ 
  if  $a1: G'.steps (x \# a \# list @ ys @ [last list]) list \neq []$ 
  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 \preceq x'> <V x> <V x'>] obtain  $y' xs' ys'$ 
where
   $y \preceq y'$ 
   $G'.steps (x' \# xs' @ y' \# ys' @ [y'])$ 
  by safe
then have  $G'.steps (x' \# xs' @ [y']) G'.steps (y' \# ys' @ [y'])$ 
  by (force intro: Graph-Start-Defs.graphI-aggressive2) +
with  $\langle y \preceq y' \rangle$  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*: $a \preceq a' \Longrightarrow \neg \text{empty } a \Longrightarrow P \ a \Longrightarrow P \ a'$

assumes *Q-mono*: $a \preceq a' \Longrightarrow \neg \text{empty } a \Longrightarrow Q \ a \Longrightarrow Q \ a'$

fixes *succs-Q* :: '*a* \Rightarrow '*a* *list*

assumes *succs-Q-correct*: $A.\text{reachable } a \Longrightarrow \text{set } (\text{succs-Q } a) = \{y. E \ a \ y \wedge Q \ y \wedge \neg \text{empty } y\}$

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 $\neg \text{empty } y$ *a*₀ λ -. *False* (\preceq) λ *x*. *A.reachable* *x* \wedge $\neg \text{empty } x$

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* \langle *auto* *intro*: *A.finite-reachable* \rangle)

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

$\lambda x y. E x y \wedge Q y \wedge \neg \text{empty } y \text{ a}_1 \lambda -. \text{False } (\preceq) \lambda x. A.\text{reachable } x \wedge \neg$
 $\text{empty } x \text{ succs-}Q$

by *standard*

definition *has-cycle* where

$\text{has-cycle} = B'.\text{dfs}$

end

definition *leadsto* :: *bool nres* where

$\text{leadsto} = \text{do } \{$

$(r, \text{passed}) \leftarrow A'.\text{pw-algo};$

$\text{let } P = \{x. x \in \text{passed} \wedge P x \wedge Q x\};$

$(r, -) \leftarrow$

$\text{FOREACH}_C P (\lambda(b,-). \neg b) (\lambda v' (-,P). \text{has-cycle } v' P) (\text{False}, \{\});$

$\text{RETURN } r$

$\}$

definition

$\text{reaches-cycle } a =$

$(\exists b. (\lambda x y. E x y \wedge Q y \wedge \neg \text{empty } y)^{**} a b \wedge (\lambda x y. E x y \wedge Q y \wedge$
 $\neg \text{empty } y)^{++} b b)$

definition *leadsto-spec* where

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

lemma

$\text{leadsto} \leq \text{leadsto-spec}$

proof –

define *inv* where

$\text{inv} \equiv \lambda \text{passed } it (r, \text{passed}').$

$(r \longrightarrow (\exists a. A.\text{reachable } a \wedge \neg \text{empty } a \wedge P a \wedge Q a \wedge \text{reaches-cycle}$
 $a))$

$\wedge (\neg r \longrightarrow$

$(\forall a \in \text{passed} - it. \neg \text{reaches-cycle } a)$

$\wedge B.\text{liveness-compatible } \text{passed}'$

$\wedge \text{passed}' \subseteq \{x. A.\text{reachable } x \wedge \neg \text{empty } x\}$

$)$

have [*simp*, *intro*]:

$\neg A'.F\text{-reachable}$
unfolding $A'.F\text{-reachable-def}$ **by** *simp*

have $B\text{-reaches-empty}$:
 $\neg \text{empty } b$ **if** $\neg \text{empty } a$ $B.\text{reaches } a \ b$ **for** $a \ b$
using *that(2,1)* **by** *induction auto*

interpret *Subgraph-Start* $E \ a_0 \ \lambda \ a \ x. \ E \ a \ x \ \wedge \ Q \ x \ \wedge \ \neg \ \text{empty } x$
by *standard auto*

have $B\text{-}A\text{-reaches}$:
 $A.\text{reaches } a \ b$ **if** $B.\text{reaches } a \ b$ **for** $a \ b$
using *that* **by** (*rule reaches*)

have reaches-iff : $B.\text{reaches } a \ x \ \longleftrightarrow \ B.G.G'.\text{reaches } a \ x$
if $A.\text{reachable } a \ \neg \ \text{empty } a$ **for** $a \ x$
unfolding *reaches-cycle-def*
apply *standard*
using *that*
apply (*rotate-tac 3*)
apply (*induction rule: rtrancl.induct*)
apply *blast*
apply (*rule rtrancl.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 : $B.\text{reaches1 } a \ x \ \longleftrightarrow \ B.G.G'.\text{reaches1 } a \ x$
if $A.\text{reachable } a \ \neg \ \text{empty } a$ **for** $a \ x$
unfolding *reaches-cycle-def*
apply *standard*
subgoal
using *that*
apply (*rotate-tac 3*)
apply (*induction rule: trancl.induct*)
apply (*solves <rule trancl.intros(1), auto simp: B.G.E'-def>*)
apply (
rule trancl.intros(2);
auto 4 3 simp: B.G.E'-def dest: B-reaches-empty trancl-into-rtrancl
)

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

have *reaches-cycle-iff*: *reaches-cycle a* \longleftrightarrow ($\exists x. B.G.G'.reaches\ a\ x \wedge B.G.G'.reaches1\ x\ x$)
 if *A.reachable a* \neg *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*:
 \neg *reaches-cycle x*
 if
 $\forall a. A.reachable\ a \wedge \neg\ empty\ a \longrightarrow (\exists x \in passed. a \preceq x)$
 $passed \subseteq \{a. A.reachable\ a \wedge \neg\ empty\ a\}$
 $\forall x \in passed. P\ x \wedge Q\ x \longrightarrow \neg\ reaches-cycle\ x$
 $A.reachable\ x \neg\ empty\ x \implies P\ x \wedge Q\ x$
 for *x passed*
proof (rule *ccontr, unfold not-not*)
 assume *reaches-cycle x*
 from *that* **obtain** *x'* **where** $x' \in passed\ x \preceq x'$
 by *auto*
 with *that* **have** $P\ x' \wedge Q\ x'$
 by (*auto intro: P-mono Q-mono*)
 with $\langle x' \in passed \rangle$ *that(3)* **have** $\neg\ reaches-cycle\ x'$
 by *auto*
 have $A.reachable\ x' \neg\ empty\ x'$
 using $\langle x' \in passed \rangle$ *that(2)* *A.empty-mono* $\langle x \preceq x' \rangle$ *that(5)* **by** *auto*
 note *reaches-cycle-iff' = reaches-cycle-iff*[*OF this*] *reaches-iff*[*OF this*]
reaches1-iff[*OF this*]
 from $\langle reaches-cycle\ x \rangle$ **obtain** *y* **where** $B.reaches\ x\ y \wedge B.reaches1\ y\ y$
 unfolding *reaches-cycle-def* **by** *atomize-elim*
interpret
 Subsumption-Graph-Pre-Nodes
 $(\preceq) A.subsumes-strictly\ \lambda\ x\ y. E\ x\ y \wedge Q\ y \wedge \neg\ empty\ y$
 $\lambda\ x. A.reachable\ x \wedge \neg\ empty\ x$
 by *standard* (rule *B.mono[simplified]; assumption*)
 from $\langle B.reaches\ x\ y \rangle \langle x \preceq x' \rangle \langle B.reaches1\ y\ y \rangle$ *reaches-cycle-mono*[*OF B.finite-V*] **obtain** *y'* **where**
 $y \preceq y' \wedge B.G.G'.reaches\ x'\ y' \wedge B.G.G'.reaches1\ y'\ y'$
 apply *atomize-elim*
 apply (subst (*asm*) *reaches-iff*[*rotated 2*])

```

defer
defer
apply (subst (asm) reaches1-iff)
  defer
  defer
  using  $\langle A.\text{reachable } x \rangle \langle \neg \text{empty } x \rangle \langle A.\text{reachable } x' \rangle \langle \neg \text{empty } x' \rangle$ 
 $\langle B.\text{reaches1 } y \ y \rangle$ 
  by (auto simp: B.reaches1-reaches-iff2 dest!: B.G.G'-reaches-V)
with  $\langle A.\text{reachable } x' \rangle \langle \neg \text{empty } x' \rangle$  have reaches-cycle x'
  unfolding reaches-cycle-iff'
  by auto
with  $\langle \neg \text{reaches-cycle } x' \rangle$  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  $a_1$  it σ a passed'
apply clarsimp
subgoal premises prems
proof -
  interpret  $B'$ :
    Liveness-Search-Space
     $\lambda x y. E x y \wedge Q y \wedge \neg \text{empty } y \ a_1 \ \lambda -. \text{False } (\preceq)$ 
     $\lambda x. A.\text{reachable } x \wedge \neg \text{empty } x \ \text{succs-}Q$ 
  by standard
from  $\langle \text{inv } - - \rightarrow \rangle$  have
   $B'.\text{liveness-compatible } \text{passed}' \ \text{passed}' \subseteq \{x. A.\text{reachable } x \wedge \neg$ 
empty } x\}

```

```

      unfolding inv-def by auto
    from B'.dfs-correct[OF - this] ⟨passed ⊆ -⟩ ⟨a₁ ∈ -⟩ ⟨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 ⊆ -⟩ ⟨a₁ ∈ -⟩ ⟨it ⊆ -⟩
      apply simp
      apply (inst-existentials a₁)
      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)** a₀ a ∧ ¬ empty a ∧ P a ∧ Q a ∧
reaches-cycle a)
  )

```

```

lemma E-reaches-non-empty:
   $(\lambda x y. E x y \wedge \neg \text{empty } y)^{**} a b$  if  $a \rightarrow^* b$  A.reachable  $a \neg \text{empty } b$  for
   $a b$ 
  using that
proof induction
  case base
  then show ?case by blast
next
  case (step  $y z$ )
  from  $\langle a \rightarrow^* y \rangle \langle A.\text{reachable } a \rangle$  have A.reachable  $y$ 
    by  $-$  (rule A.reachable-reaches)
  have  $\neg \text{empty } y$ 
  proof (rule ccontr, unfold not-not)
    assume empty  $y$ 
    from A.empty-E[OF  $\langle A.\text{reachable } y \rangle \langle \text{empty } y \rangle \langle E y z \rangle$ ]  $\langle \neg \text{empty } z \rangle$ 
show False by blast
  qed
  with step show ?case
    by (auto intro: rtranclp.intros(2))
qed

```

```

lemma leadsto-spec-leadsto-spec-alt:
  leadsto-spec  $\leq$  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* $:: ('b \rightarrow 'a \text{ set}) \Rightarrow 'a \text{ set}$ **where**
map-to-set $m = (\bigcup (\text{ran } m))$

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)  
  (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; 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 a0 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-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 a₀ ↦ [a₀]), *ms*(key a₀ ↦ {a₀})) ∈ *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 a₀ ↦ [a₀]], [a₀]), [key a₀ ↦ {a₀}], [a₀]) ∈ *map-list-set-rel* ×_{*r*} *Id*
by (*simp add: map-list-set-rel-single*)

lemma *pw-map-inv-ref*:
pw-map-inv (x₁, x₂, x₃) ⇒ (x_{1a}, x₁) ∈ *map-list-set-rel* ⇒ *pw-map-inv3*
 (x_{1a}, x₂, x₃)
unfolding *pw-map-inv3-def* **by** *auto*

lemma *refine-aux*[*refine*]:
 $(x1, x) \in \text{map-list-set-rel} \implies ((x1, x2, \text{False}), x, x2, \text{False}) \in \text{map-list-set-rel}$
 $\times_r \text{Id} \times_r \text{Id}$
by *simp*

lemma *map-list-set-relD*:
 $ms\ k = \text{Some}(\text{set } xs)$ **if** $(ml, ms) \in \text{map-list-set-rel}$ $ml\ k = \text{Some } xs$
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** $(ml, ms) \in \text{map-list-set-rel}$ $ml\ k = \text{Some } xs$
using *that unfolding map-list-set-rel-def by clarsimp (metis domI option.sel)*

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

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

lemma *map-list-set-rel-insert*:
 $(ml, ms) \in \text{map-list-set-rel} \implies$
 $ml(\text{key } a) = \text{Some } xs \implies$
 $ms(\text{key } a) = \text{Some}(\text{set } xs) \implies$
 $a \notin \text{set } xs \implies$
 $(ml(\text{key } a \mapsto a \# xs), ms(\text{key } a \mapsto \text{insert } a(\text{set } xs))) \in \text{map-list-set-rel}$
apply (*frule map-list-set-rel-distinct*) **unfolding** *map-list-set-rel-def* **by**
auto

lemma *add-pw'-map3-ref*:
 $\text{add-pw'-map3 } ml\ xs\ a \leq \Downarrow (\text{map-list-set-rel} \times_r \text{Id}) (\text{add-pw'-map2 } ms\ xs$
 $a)$
if $(ml, ms) \in \text{map-list-set-rel} \neg \text{empty } a$
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  $\langle (ml, ms) \in \text{map-list-set-rel} \rangle$  have  $ms (\text{key } a) = \text{None}$ 
    by auto
  with None  $\langle (ml, ms) \in \text{map-list-set-rel} \rangle$  show ?thesis
    by (auto intro: map-list-set-rel-single)
next
  case (Some xs)
from map-list-set-relD[OF  $\langle (ml, ms) \in \text{map-list-set-rel} \rangle \langle ml = - \rangle]$  have
   $ms (\text{key } a) = \text{Some } (\text{set } xs)$ 
  by auto
moreover from prems have  $a \notin \text{set } xs$ 
  by (metis Some empty-subsumes' local.eq-refl)
ultimately show ?thesis
  using Some  $\langle (ml, ms) \in \text{map-list-set-rel} \rangle$  by (auto intro: map-list-set-rel-insert)
qed
done

```

```

lemma pw-algo-map3-ref[refine]:
   $pw\text{-algo-map3} \leq \Downarrow (\text{Id} \times_r \text{map-list-set-rel}) pw\text{-algo-map2}$ 
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':
   $pw\text{-algo-map2} \leq \Downarrow (\text{bool-rel} \times_r \text{map-set-rel}) pw\text{-algo}$ 
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 ≤ ↓ (bool-rel ×r (map-list-set-rel O map-set-rel)) pw-algo
proof –
  have [simp]:
    ((bool-rel ×r map-list-set-rel) O (bool-rel ×r map-set-rel))
    = (bool-rel ×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 a0 λ -. False (≲) empty (≲) E key
succs1 λ -. False
  by (standard; blast intro!: A.succs-correct)

```

```

interpretation B:
  Liveness-Search-Space-Key

```

$\lambda x y. E x y \wedge Q y \wedge \neg \text{empty } y \ a_0 \ \lambda -. \text{False } (\preceq) \ \lambda x. A.\text{reachable } x \wedge \neg \text{empty } x$
succs-Q key
by standard (*auto intro!: A.empty-subsumes'*)

context

fixes $a_1 :: 'a$

begin

interpretation B' :

Liveness-Search-Space-Key-Defs

$a_1 \ \lambda -. \text{False } (\preceq) \ \lambda x. A.\text{reachable } x \wedge \neg \text{empty } x$

succs-Q $\lambda x y. E x y \wedge Q y \wedge \neg \text{empty } y \ \text{key} .$

definition *has-cycle-map where*

$\text{has-cycle-map} = B'.\text{dfs-map}$

context

assumes $A.\text{reachable } a_1$

begin

interpretation B' :

Liveness-Search-Space-Key

$\lambda x y. E x y \wedge Q y \wedge \neg \text{empty } y \ a_1 \ \lambda -. \text{False } (\preceq) \ \lambda x. A.\text{reachable } x \wedge \neg \text{empty } x$

succs-Q key

by standard

lemmas $\text{has-cycle-map-ref}[\text{refine}] = B'.\text{dfs-map-dfs-refine}[\text{folded } \text{has-cycle-map-def } \text{has-cycle-def}]$

end

end

definition *outer-inv where*

$\text{outer-inv passed done todo} \equiv \lambda (r, \text{passed}')$

$(r \longrightarrow (\exists a. A.\text{reachable } a \wedge \neg \text{empty } a \wedge P a \wedge Q a \wedge \text{reaches-cycle } a))$

$\wedge (\neg r \longrightarrow$

$(\forall a \in \bigcup \text{done}. P a \wedge Q a \longrightarrow \neg \text{reaches-cycle } a)$

$\wedge B.\text{liveness-compatible } \text{passed}'$

$\wedge \text{passed}' \subseteq \{x. A.\text{reachable } x \wedge \neg \text{empty } x\}$

$)$

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)
    (λv' (-,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:*

$leadsto' \leq leadsto-spec$

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*]:
 $\neg A'.F\text{-reachable}$
 unfolding $A'.F\text{-reachable-def}$ **by** *simp*

have $B\text{-reaches-empty}$:
 $\neg \text{empty } b$ **if** $\neg \text{empty } a$ $B.\text{reaches } a\ b$ **for** $a\ b$
 using *that(2,1)* **by** *induction auto*

interpret *Subgraph-Start* $E\ a_0\ \lambda\ a\ x.\ E\ a\ x\ \wedge\ Q\ x\ \wedge\ \neg\ \text{empty } x$
 by *standard auto*

have $B\text{-}A\text{-reaches}$:
 $A.\text{reaches } a\ b$ **if** $B.\text{reaches } a\ b$ **for** $a\ b$
 using *that* **by** (*rule reaches*)

have reaches-iff : $B.\text{reaches } a\ x \iff B.G.G'.\text{reaches } a\ x$
if $A.\text{reachable } a \neg \text{empty } a$ **for** $a\ x$
 unfolding reaches-cycle-def
 apply *standard*
 using *that*
 apply (*rotate-tac 3*)
 apply (*induction rule: rtrancl.induct*)
 apply *blast*
 apply (*rule rtrancl.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 : $B.\text{reaches1 } a\ x \iff B.G.G'.\text{reaches1 } a\ x$
if $A.\text{reachable } a \neg \text{empty } a$ **for** $a\ x$
 unfolding reaches-cycle-def
 apply *standard*
 subgoal
 using *that*
 apply (*rotate-tac 3*)
 apply (*induction rule: trancl.induct*)
 apply (*solves <rule trancl.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  $\longleftrightarrow$  ( $\exists x. B.G.G'.reaches a x \wedge B.G.G'.reaches1 x x$ )
  if A.reachable a  $\neg$  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:
   $\neg$  reaches-cycle x
  if
     $\forall a. A.reachable a \wedge \neg empty a \longrightarrow (\exists x \in passed. a \preceq x)$ 
     $passed \subseteq \{a. A.reachable a \wedge \neg empty a\}$ 
     $\forall y \in ran\ passed'. \forall x \in y. P x \wedge Q x \longrightarrow \neg reaches-cycle x$ 
     $A.reachable x \neg empty x \implies P x \wedge Q x$ 
     $(passed', passed) \in A'.map-set-rel$ 
  for x passed passed'
proof (rule ccontr, unfold not-not)
  assume reaches-cycle x
  from that obtain x' where  $x':x' \in passed x \preceq x'$ 
  by auto
  with  $\langle(-, -) \in -\rangle$  obtain y where  $y: y \in ran\ passed' x' \in y$ 
  unfolding A'.map-set-rel-def by auto
  from x' that have  $P x' Q x'$ 
  by (auto intro: P-mono Q-mono)
  with  $\langle x' \in passed \rangle$  that(3) y have  $\neg reaches-cycle x'$ 
  by auto
  have  $A.reachable x' \neg empty x'$ 
  using  $\langle x' \in passed \rangle$  that(2) A.empty-mono  $\langle x \preceq x' \rangle$  that(5) by auto
  note  $reaches-cycle-iff' = reaches-cycle-iff'[OF this] reaches-iff'[OF this]$ 
reaches1-iff[OF this]
  from  $\langle reaches-cycle x \rangle$  obtain y where  $B.reaches x y B.reaches1 y y$ 
  unfolding reaches-cycle-def by atomize-elim
interpret
  Subsumption-Graph-Pre-Nodes
   $(\preceq) A.subsumes-strictly \lambda x y. E x y \wedge Q y \wedge \neg empty y$ 

```

```

    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' a1 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 $\langle passed \subseteq - \rangle \langle p \subseteq - \rangle$ **have** $A.reachable\ a_1$

by *auto*

interpret *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*

from $\langle inner-inv\ -\ -\ - \rangle$ **have**

$B'.liveness-compatible\ passed'\ passed' \subseteq \{x.\ A.reachable\ x \wedge \neg\ empty\ x\}$

unfolding *inner-inv-def* **by** *auto*

from $B'.dfs-correct[OF\ -\ this]\ \langle passed \subseteq - \rangle \langle a_1 \in - \rangle \langle p \subseteq - \rangle$ **have**

$B'.dfs\ passed' \leq B'.dfs-spec$

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 $\langle passed \subseteq - \rangle \langle a_1 \in - \rangle \langle p \subseteq - \rangle$

apply *simp*

apply *(inst-existentials a1)*

by *(auto 4 3 simp: reaches-cycle-iff intro: prems)*

subgoal

using $\langle inner-inv - - - \rangle \langle passed \subseteq - \rangle \langle a_1 \in - \rangle \langle p \subseteq - \rangle$
reaches-cycle-iff
unfolding *inner-inv-def* **by** *auto*
done
done
qed
done

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

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

subgoal for $x a b S1 S2 xa \sigma aa ba \sigma'$
unfolding *inner-inv-def* *outer-inv-def* **by** *auto*

subgoal for $x a b S1 S2 \sigma aa ba$
unfolding *outer-inv-def* **by** *auto*

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

done
qed

lemma *init-ref[refine]*:
 $((False, Map.empty), False, \{\}) \in bool-rel \times_r A'.map-set-rel$
unfolding *A'.map-set-rel-def* **by** *auto*

lemma *has-cycle-map-ref'[refine]*:
assumes $(P1, P1') \in A'.map-set-rel (a, a') \in Id A.reachable a \neg empty$
 a
shows $has-cycle-map a P1 \leq \Downarrow (bool-rel \times_r A'.map-set-rel) (has-cycle a' P1')$
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*:
 $(\text{ran } ml, \text{ran } ms) \in \langle \text{br set } (\lambda-. \text{True}) \rangle \text{set-rel}$ **if** $(ml, ms) \in \text{map-list-set-rel}$
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** $ms\ a = \text{Some } x'$
unfolding *ran-def* **by** *clarsimp*
with $prems(1)$ **obtain** m' **where**
 $ml\ a = \text{Some } (m'\ a)$
by (*fastforce simp: dom-def ran-def*)
with $prems(2)$ $\langle ms\ a = \rightarrow \rangle$ **show** *?thesis*
by (*fastforce simp: in-br-conv dom-def ran-def*)
qed
done

lemma *Id-list-rel-ref*:
 $(x'a, x'a) \in \langle Id \rangle \text{list-rel}$
by *simp*

lemma *map-list-set-rel-finite-ran*:
 $\text{finite } (\text{ran } ml)$ **if** $(ml, ms) \in \text{map-list-set-rel}$
using *that unfolding map-list-set-rel-def* **by** (*auto intro: map-dom-ran-finite*)

lemma *leadsto-map3-ref[refine]*:
 $\text{leadsto-map3} \leq \Downarrow Id\ \text{leadsto}'$
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**
 $\text{leadsto-map4} = \text{do } \{$
 $(r, \text{passed}) \leftarrow A'.\text{pw-algo-map3};$
 $\text{ASSERT } (\text{finite } (\text{dom } \text{passed}));$
 $\}$

```

    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 →_a bool-assn ×_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 {a₀} lso-fold-custom-empty)*

unfolding *hm.hms-fold-custom-empty*

apply *(rewrite in [a₀] 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 a₀ F - empty - E key F' P Q succs-Q succs1 +

liveness: Liveness-Search-Space-Key-Impl a₀ F - V succs-Q λ x y. E x y

∧ ¬ empty y ∧ Q y

- key A succsi a₀i Lei keyi copyi

for *key :: 'v ⇒ 'k*

and *a₀ F F' copyi P Q V empty succs-Q succs1 E A succsi a₀i Lei keyi +*

fixes *succs1i and emptyi and Pi Qi and tracei*

assumes *succs1-impl: (succs1i, (RETURN ∘ PR-CONST) succs1) ∈ A^k →_a list-assn A*

and *empty-impl:*

(emptyi, RETURN o PR-CONST empty) ∈ A^k →_a bool-assn

assumes *[sepref-fr-rules]:*

(Pi, RETURN o PR-CONST P) ∈ A^k →_a bool-assn (Qi, RETURN o PR-CONST Q) ∈ A^k →_a bool-assn

assumes *trace-impl:*

*(uncurry tracei, uncurry (λ(- :: string) -. RETURN ())) ∈ id-assn^k *_a A^k →_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-assnk →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,-*) \in -

lemma *leadsto-impl-hnr*:

(*uncurry0* (
leadsto-impl copyi succsi a₀i Lei keyi succs1i emptyi Pi Qi tracei
),
uncurry0 leadsto-spec-alt
) \in *unit-assn*^{*k*} \rightarrow_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-rell]
] .

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.