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

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 :: $'a \Rightarrow 'a \Rightarrow bool$ — Step relation
    and a_0 :: $'a$ — Start state
    and F :: $'a \Rightarrow bool$ — Final states
    and *subsumes* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\preceq$ *50*) — Subsumption preorder
begin

  sublocale *Graph-Start-Defs* E a_0 ⟨*proof*⟩

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

end

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

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

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

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

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

and *F-mono*: $a \preceq a' \Longrightarrow F\ a \Longrightarrow F\ a'$
begin

sublocale *preorder* $(\preceq)$ $(\prec)$
⟨*proof*⟩

end

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

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

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

sublocale *preorder* $(\preceq)$ $(\prec)$
⟨*proof*⟩

sublocale *search-space*:
Search-Space-Nodes $\lambda\ x\ y.\ E\ x\ y \wedge \neg\ empty\ y\ a_0\ F\ (\preceq)\ \lambda\ v.\ V\ v \wedge \neg\ empty\ v$
⟨*proof*⟩

end

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

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

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

and *empty-E*: *reachable* $x \Longrightarrow$ *empty* $x \Longrightarrow E\ x\ x' \Longrightarrow$ *empty* x'
and *F-mono*: $a \preceq a' \Longrightarrow F\ a \Longrightarrow F\ a'$
begin

sublocale *preorder* $(\preceq)$ $(\prec)$
⟨*proof*⟩

sublocale *Search-Space-Nodes-Empty* E a_0 F $(\preceq)$ *reachable empty*
including *graph-automation*
⟨*proof*⟩

end

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

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

sublocale *Search-Space-finite-strict* ⊆ *Search-Space-finite*
⟨*proof*⟩

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

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

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

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

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

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

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

sublocale *Search-Space″-finite* ⊆ *Search-Space′-finite* ⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

sublocale *Worklist-Map2-finite* ⊆ *Search-Space''-finite* ⟨*proof*⟩

locale *Worklist4-Impl-Defs* = *Worklist3-Defs* +
fixes A :: $'a \Rightarrow 'ai \Rightarrow$ *assn*
fixes *succsi* :: $'ai \Rightarrow 'ai$ *list Heap*
fixes $a_0 i$ :: $'ai$ *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* $a_0 i$, *uncurry0* (*RETURN* (*PR-CONST* a_0))) $\in$ *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* ($\trianglelefteq$))) $\in A^k *_a A^k \rightarrow_a$ *bool-assn*
assumes [*sepref-fr-rules*]: (*succsi*,*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* ⟨*proof*⟩

locale *Worklist-Map2-Impl-Defs* =
Worklist4-Impl-Defs - - - - - - - - *A* + *Worklist-Map2-Defs* a_0 - - - - - *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* (λ *x y. P x* $\vee$ *y*) *False*
⟨*proof*⟩

lemma (**in** *Search-Space-finite*) *finitely-branching*:
assumes *reachable a*
shows *finite* ({a'. *E a* a' $\wedge$ $\neg$ *empty* a'})

⟨*proof*⟩

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

end

1.2 Miscellaneous

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

lemma *mset-eq-empty-iff*:
  M = {#} ⟷ *set-mset M* = {}
  ⟨*proof*⟩

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

lemma *mset-remove-member*:
  x ∈# *A* − *B* **if** *x* ∈# *A x* ∉# *B*
  ⟨*proof*⟩

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

datatype *message* = *ExploredState*

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

code-printing code-module *Tracing* ⇀ (*SML*)
‹
structure Tracing : *sig*
  val count-up : *unit* −> *unit*

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

code-reserved *SML Tracing*

code-reserved *OCaml Tracing*

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

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

lemma *trace-alt-def*[*simp*]:
  trace m x = (λ -. *x*) (*write-msg x*)
  ⟨*proof*⟩

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

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

lemma *TRACE-bind*[*simp*]:
  do {*TRACE m*; *c*} = *c*
  ⟨*proof*⟩

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

⟨*proof*⟩

sepref-definition *TRACE-impl* **is**
TRACE :: $id\text{-}assn^k \rightarrow_a unit\text{-}assn$
⟨*proof*⟩

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′* ())
⟨*proof*⟩

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

sepref-definition *TRACE′-impl* **is**
uncurry0 TRACE′ :: $unit\text{-}assn^k \rightarrow_a unit\text{-}assn$
⟨*proof*⟩

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

lemma *rtranclp-transitive-compress1*: $R\ a\ c$ **if** $R\ a\ b\ R^{**}\ b\ c$
⟨*proof*⟩

lemma *rtranclp-transitive-compress2*: $R\ a\ c$ **if** $R^{**}\ a\ b\ R\ b\ c$
⟨*proof*⟩

end

lemma *rtranclp-ev-induct*[*consumes 1*, *case-names irrefl trans step*]:
fixes $P :: 'a \Rightarrow bool$ **and** $R :: 'a \Rightarrow 'a \Rightarrow bool$
assumes *reachable-finite*: *finite* $\{x.\ R^{**}\ a\ x\}$
assumes *R-irrefl*: $\bigwedge x.\ \neg\ R\ x\ x$ **and** *R-trans*[*intro*]: $\bigwedge x\ y\ z.\ R\ x\ y \Longrightarrow R\ y\ z \Longrightarrow R\ x\ z$
assumes *step*: $\bigwedge x.\ R^{**}\ a\ x \Longrightarrow P\ x \vee (\exists\ y.\ R\ x\ y)$
shows $\exists\ x.\ P\ x \wedge R^{**}\ a\ x$
⟨*proof*⟩

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*]: $\bigwedge x\ y\ z.\ R\ x\ y \Longrightarrow R\ y\ z \Longrightarrow R\ x\ z$
assumes *step*: $\bigwedge x.\ Q\ x \Longrightarrow P\ x \vee (\exists\ y.\ R\ x\ y \wedge Q\ y)$
shows $\exists\ x.\ P\ x \wedge Q\ x \wedge R^{**}\ a\ x$
⟨*proof*⟩

2.2 Definitions

locale *Subsumption-Graph-Pre-Defs* =
ord less-eq less **for** *less-eq* $:: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\preceq$ *50*) **and** *less* (**infix** $\prec$ *50*) +
fixes $E :: 'a \Rightarrow 'a \Rightarrow bool$ — The full edge set
begin

sublocale *Graph-Defs E* ⟨*proof*⟩

end

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

sublocale *Subgraph-Node-Defs-Notation* ⟨*proof*⟩

end

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

sublocale *Graph-Start-Defs E* s_0 ⟨*proof*⟩

sublocale *G*: *Graph-Start-Defs RE* s_0 ⟨*proof*⟩

sublocale *G'*: *Graph-Start-Defs* λ *x y*. *RE x y* ∨ (*x* ≺ *y* ∧ *G.reachable y*) s_0 ⟨*proof*⟩

abbreviation *G'-E* (- $\rightarrow_{G'}$ - [*100*, *100*] *40*) **where**
G'-E x y ≡ *RE x y* ∨ (*x* ≺ *y* ∧ *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*:
$a \preceq b \Longrightarrow E\ a\ a' \Longrightarrow reachable\ a \Longrightarrow reachable\ b \Longrightarrow \exists\ b'.\ E\ b\ b' \wedge a' \preceq b'$
begin

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

end

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

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

end

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

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

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

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

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

assumes *subgraph*: $\forall\ s\ s'.\ RE\ s\ s' \longrightarrow E\ s\ s'$
assumes *finite-reachable*: $finite\ \{a.\ G.reachable\ a\}$
begin

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

end

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

sublocale *Graph-Start-Defs* $E\ s_0$ $\langle proof \rangle$

sublocale G: *Graph-Start-Defs* $RE\ s_0$ $\langle proof \rangle$

end

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

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

lemma *Simulation-close*:
$Simulation\ A\ (close\ A)\ (\lambda\ a\ b.\ b = closure\ a)$
⟨*proof*⟩

sublocale *view*: *Reachability-Compatible-Subsumption-Graph*
$(\preceq)\ (\prec)\ close\ E\ closure\ s_0\ close\ RE$
⟨*proof*⟩

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-Fina*
+
assumes *no-subsumption-cycle*:
$G'.reachable\ x \Longrightarrow x \to_G{}^{+\prime}\ x \Longrightarrow x \to_G{}^{+}\ x$

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*
⟨*proof*⟩

lemma *G-reachable-G′-reachable*[*intro*]:
  G'.*reachable a* **if** *G.reachable a*
  ⟨*proof*⟩

lemma *G-G′-reachable-iff*:
  G.reachable a ⟷ G'.*reachable a*
  ⟨*proof*⟩

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*}
  ⟨*proof*⟩

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*⟩

lemma *reachable-has-surrogate*:
  $\exists\ t.\ G.reachable\ t \wedge s \preceq t \wedge (\forall\ s'.\ E\ t\ s' \longrightarrow RE\ t\ s')$ **if** *reachable s*
  ⟨*proof*⟩

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

corollary *reachability-correct*:
  $\nexists\ s'.\ reachable\ s' \wedge F\ s'$ **if** $\nexists\ s'.\ G.reachable\ s' \wedge F\ s'$
  ⟨*proof*⟩

end

end

context *Reachability-Compatible-Subsumption-Graph*

begin

Setup for automation

context
  includes *graph-automation*
begin

lemma *subgraph′*[*intro*]:
  $E\ s\ s'$ **if** $RE\ s\ s'$
  ⟨*proof*⟩

lemma *G-reachability-sound*[*intro*]:
  reachable a **if** *G.reachable a*
  ⟨*proof*⟩

lemma *G-steps-sound*[*intro*]:
  steps xs **if** *G.steps xs*
  ⟨*proof*⟩

lemma *G-run-sound*[*intro*]:
  run xs **if** *G.run xs*
  ⟨*proof*⟩

lemma *G′-reachability-sound*[*intro*]:
  reachable a **if** G'*.reachable a*
  ⟨*proof*⟩

lemma *G′-finite-reachable*: *finite* $\{a.\ G'.reachable\ a\}$
  ⟨*proof*⟩

lemma *G-steps-G′-steps*[*intro*]:
  $G'.steps\ as$ **if** $G.steps\ as$
  ⟨*proof*⟩

lemma *reachable-has-surrogate*:
  $\exists\ t.\ G.reachable\ t \wedge s \preceq t \wedge (\forall\ s'.\ E\ t\ s' \longrightarrow RE\ t\ s')$ **if** $G.reachable\ s$
⟨*proof*⟩

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*⟩

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*⟩

lemma *subsumption-step′*:
∃ *b′*. *b* $\preceq$ *b′* ∧ *a′* $\rightarrow_G{}^{+}{}'$ *b′* **if** *reachable a a* → *b G′.reachable a′ a* $\preceq$ *a′*
⟨*proof*⟩

theorem *reachability-complete′*:
∃ *s′*. *s* $\preceq$ *s′* ∧ *G.reachable s′* **if** *a* →∗ *s G.reachable a*
⟨*proof*⟩

corollary *reachability-complete*:
∃ *s′*. *s* $\preceq$ *s′* ∧ *G.reachable s′* **if** *reachable s*
⟨*proof*⟩

corollary *reachability-correct*:
(∃ *s′*. *s* $\preceq$ *s′* ∧ *reachable s′*) ⟷ (∃ *s′*. *s* $\preceq$ *s′* ∧ *G.reachable s′*)
⟨*proof*⟩

lemma *steps-G′-steps*:
∃ *ys ns*. *list-all2* ($\preceq$) *xs* (*nths ys ns*) ∧ *G′.steps* (*b* # *ys*) **if**
steps (*a* # *xs*) *reachable a a* $\preceq$ *b G′.reachable b*
⟨*proof*⟩

lemma *cycle-G′-cycle″*:
assumes *steps* (s_0 # *ws* @ *x* # *xs* @ [*x*])
shows ∃ *x′ xs′ ys′*. *x* $\preceq$ *x′* ∧ *G′.steps* (s_0 # *xs′* @ *x′* # *ys′* @ [*x′*])
⟨*proof*⟩

lemma *cycle-G′-cycle′*:
assumes *steps* (s_0 # *ws* @ *x* # *xs* @ [*x*])
shows ∃ *y ys*. *x* $\preceq$ *y* ∧ *G′.steps* (*y* # *ys* @ [*y*]) ∧ *G′.reachable y*
⟨*proof*⟩

lemma *cycle-G′-cycle*:
assumes *reachable x x* $\rightarrow^{+}$ *x*
shows ∃ *y ys*. *x* $\preceq$ *y* ∧ *G′.reachable y* ∧ *y* $\rightarrow_G{}^{+}{}'$ *y*
⟨*proof*⟩

corollary *G′-reachability-complete*:
∃ *s′*. *s* $\preceq$ *s′* ∧ *G.reachable s′* **if** *G′.reachable s*
⟨*proof*⟩

end

end

corollary (**in** *Reachability-Compatible-Subsumption-Graph-Final*) *reachability-correct*:
  $(\exists\ s'.\ reachable\ s' \wedge F\ s') \longleftrightarrow (\exists\ s'.\ G.reachable\ s' \wedge F\ s')$
  ⟨*proof*⟩

2.4 Liveness

theorem (**in** *Liveness-Compatible-Subsumption-Graph*) *cycle-iff*:
  $(\exists\ x.\ x \rightarrow^{+} x \wedge reachable\ x \wedge F\ x) \longleftrightarrow (\exists\ x.\ x \rightarrow_{G}{}^{+} x \wedge G.reachable\ x \wedge F\ x)$
  ⟨*proof*⟩

2.5 Appendix

context *Subsumption-Graph-Pre-Nodes*
begin

Setup for automation

context
  includes *graph-automation*
begin

lemma *steps-mono*:
  assumes $G'.steps\ (x\ \#\ xs)\ x \preceq y\ V\ x\ V\ y$
  shows $\exists\ ys.\ G'.steps\ (y\ \#\ ys) \wedge list\text{-}all2\ (\preceq)\ xs\ ys$
  ⟨*proof*⟩ **including** *subgraph-automation*
⟨*proof*⟩

lemma *steps-append-subsumption*:
  assumes $G'.steps\ (x\ \#\ xs)\ G'.steps\ (y\ \#\ ys)\ y \preceq last\ (x\ \#\ xs)\ V\ x\ V\ y$
  shows $\exists\ ys'.\ G'.steps\ (x\ \#\ xs\ @\ ys') \wedge list\text{-}all2\ (\preceq)\ ys\ ys'$
⟨*proof*⟩

lemma *steps-replicate-subsumption*:
  assumes $x \preceq last\ (x\ \#\ xs)\ G'.steps\ (x\ \#\ xs)\ n > 0\ V\ x$
  notes [*intro*] = *preorder-intros*
  shows $\exists\ ys.\ G'.steps\ (x\ \#\ ys) \wedge list\text{-}all2\ (\preceq)\ (concat\ (replicate\ n\ xs))\ ys$
  ⟨*proof*⟩

context
  assumes *finite-V*: $finite\ \{x.\ V\ x\}$

begin

lemma *wf-less-on-reachable-set*:
  assumes *antisym*: $\bigwedge x\ y.\ x \preceq y \Longrightarrow y \preceq x \Longrightarrow x = y$
  shows $wf\ \{(x,\ y).\ y \prec x \wedge V\ x \wedge V\ y\}$ (**is** $wf\ ?S$)
⟨*proof*⟩

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

lemma *pre-cycle-cycle′*:
  assumes A: $x \preceq x'\ G'.steps\ (x \# xs\ @\ [x'])\ V\ x$
  shows $\exists\ x''\ ys.\ x' \preceq x'' \wedge G'.steps\ (x'' \# ys\ @\ [x'']) \wedge V\ x''$
⟨*proof*⟩

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* ⟨*proof*⟩

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*⟩
    including *graph-automation-aggressive*
    ⟨*proof*⟩

end

end

end

context *Subsumption-Graph-Pre*
begin

Setup for automation

context
  includes *graph-automation*
begin

interpretation *Subsumption-Graph-Pre-Nodes - - E reachable*
  ⟨*proof*⟩

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*
⟨*proof*⟩

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'*
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *wf-less-on-reachable-set*:
assumes *antisym*: $\bigwedge x\ y.\ x \preceq y \Longrightarrow y \preceq x \Longrightarrow x = y$
shows *wf* $\{(x, y).\ y \prec x \wedge$ *reachable* $x \wedge$ *reachable* $y\}$ (**is** *wf ?S*)
⟨*proof*⟩

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

lemma *pre-cycle-cycle'*:
assumes *A*: $x \preceq x'$ *steps* $(x \ \# \ xs \ @ \ [x'])$ *reachable* x
shows $\exists$ x'' *ys*. $x' \preceq x'' \wedge$ *steps* $(x'' \ \# \ ys \ @ \ [x'']) \wedge$ *reachable* x''
⟨*proof*⟩

lemma *pre-cycle-cycle*:
$(\exists\ x\ x'.$ *reachable* $x \wedge$ *reaches* $x\ x' \wedge x \preceq x') \longleftrightarrow (\exists\ x.$ *reachable* $x \wedge$ *reaches* $x\ x)$
including *reaches-steps-iff* ⟨*proof*⟩

end

end

end

context *Subsumption-Graph-Defs*
begin

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

lemma G''-*reachable*-G'[*intro*]:
  G'.*reachable* x **if** G''.*reachable* x
  ⟨*proof*⟩

end

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

sublocale G''-*pre*: *Subsumption-Graph-Pre* $(\preceq)$ $(\prec)$ λ x y. $\exists$ z. *G.reachable* $z \wedge x \preceq z \wedge$ *RE* z y
⟨*proof*⟩

end

2.6 Old Material

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

Setup for automation

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

lemma *subgraph'*[*intro*]:

$E\ s\ s'$ **if** $RE\ s\ s'$
⟨*proof*⟩

lemma *G-reachability-sound*[*intro*]:
reachable a **if** *G.reachable a*
⟨*proof*⟩

lemma *G-steps-sound*[*intro*]:
steps xs **if** *G.steps xs*
⟨*proof*⟩

lemma *G-run-sound*[*intro*]:
run xs **if** *G.run xs*
⟨*proof*⟩

lemma *reachable-has-surrogate*:
$\exists\ t.\ G.reachable\ t \wedge s \preceq t \wedge (\forall\ s'.\ E\ t\ s' \longrightarrow RE\ t\ s')$ **if** $G.reachable\ s$
⟨*proof*⟩

lemma *subsumption-step*:
$\exists\ a''\ b'.\ a' \preceq a'' \wedge b \preceq b' \wedge RE\ a''\ b' \wedge G.reachable\ a''$ **if**
$reachable\ a\ E\ a\ b\ G.reachable\ a'\ a \preceq a'$
⟨*proof*⟩

theorem *reachability-complete'*:
$\exists\ s'.\ s \preceq s' \wedge G.reachable\ s'$ **if** $E^{**}\ a\ s\ G.reachable\ a$
⟨*proof*⟩

corollary *reachability-complete*:
$\exists\ s'.\ s \preceq s' \wedge G.reachable\ s'$ **if** $reachable\ s$
⟨*proof*⟩

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

lemma *G'-reachability-sound*[*intro*]:
reachable a **if** *G'.reachable a*
⟨*proof*⟩

corollary *G'-reachability-complete*:
$\exists\ s'.\ s \preceq s' \wedge G.reachable\ s'$ **if** $G'.reachable\ s$
⟨*proof*⟩

end

end

end

3 Unified Passed-Waiting-List

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

hide-const *wait*

3.1 Utilities

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

lemma *take-from-set-correct*:
 assumes $s \neq \{\}$
 shows *take-from-set s* $\leq$ *SPEC* $(\lambda\ (x, s').\ x \in s \wedge s' = s - \{x\})$
⟨*proof*⟩

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

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

lemma *take-from-mset-correct*:
 assumes $s \neq \{\#\}$
 shows *take-from-mset s* $\leq$ *SPEC* $(\lambda\ (x, s').\ x \in\# s \wedge s' = s - \{\#x\#\})$
⟨*proof*⟩

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

lemma *set-mset-mp*: *set-mset m* $\subseteq$ *s* $\Longrightarrow$ *n* $<$ *count m x* $\Longrightarrow$ $x \in s$
 ⟨*proof*⟩

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

3.2 Generalized Worklist Algorithm

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

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

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

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

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

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

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

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

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

lemma *pw-inv-frontier-frontier'*:
pw-inv-frontier' passed wait **if**
pw-inv-frontier passed wait passed $\subseteq$ *Collect reachable*
⟨*proof*⟩

lemma
pw-inv-frontier passed wait **if** *pw-inv-frontier' passed wait*
⟨*proof*⟩

definition *pw-algo* **where**
pw-algo = *do*
{
if F a_0 *then RETURN* (*True*, {})
else if empty a_0 *then RETURN* (*False*, {})

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

end

Correctness Proof **instance** *nat* :: *preorder* ⟨*proof*⟩

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*⟩

lemma *wf-worklist-var*:
  wf pw-var
  ⟨*proof*⟩

context
begin

private lemma *aux5*:
  assumes
    a′ ∈ *passed′*
    a ∈# *wait*
    a ⪯ *a′*

start-subsumed passed wait
$\forall$ *s*$\in$*passed*. $\exists$ *x*$\in$*passed'*. *s* $\preceq$ *x*
$\forall$ *s*$\in$#*wait* $-$ {#*a*#}. *Multiset.Bex wait'* (($\preceq$) *s*)
shows *start-subsumed passed' wait'*
⟨*proof*⟩ **lemma** *aux11*:
assumes
empty a
start-subsumed passed wait
shows *start-subsumed passed* (*wait* $-$ {#*a*#})
⟨*proof*⟩

lemma *aux3-aux*:
assumes *pw-inv-frontier' passed wait*
$\neg$ *b* $\in'$ *set-mset wait*
E b b'
$\neg$ *empty b* $\neg$ *empty b'*
b $\in'$ *passed*
reachable b passed $\subseteq$ {*a*. *reachable a* $\wedge$ $\neg$ *empty a*}
shows *b'* $\in'$ *passed* $\cup$ *set-mset wait*
⟨*proof*⟩ **lemma** *pw-inv-frontier-empty-elem*:
assumes *pw-inv-frontier passed wait passed* $\subseteq$ {*a*. *reachable a* $\wedge$ $\neg$ *empty a*} *empty a*
shows *pw-inv-frontier passed* (*wait* $-$ {#*a*#})
⟨*proof*⟩ **lemma** *aux3*:
assumes
set-mset wait $\subseteq$ *Collect reachable*
a $\in$# *wait*
$\forall$ *s* $\in$ *set-mset* (*wait* $-$ {#*a*#}). $\exists$ *s'* $\in$ *set-mset wait'*. *s* $\preceq$ *s'*
$\forall$ *s* $\in$ {*a'*. *E a a'* $\wedge$ $\neg$ *empty a'*}. $\exists$ *s'* $\in$ *passed* $\cup$ *set-mset wait'*. *s* $\preceq$ *s'*
$\forall$ *s* $\in$ *passed* $\cup$ {*a*}. $\exists$ *s'* $\in$ *passed'*. *s* $\preceq$ *s'*
passed' $\subseteq$ *passed* $\cup$ {*a*} $\cup$ {*a'* . *E a a'* $\wedge$ $\neg$ *empty a*}
pw-inv-frontier passed wait
passed $\subseteq$ {*a*. *reachable a* $\wedge$ $\neg$ *empty a*}
shows *pw-inv-frontier passed' wait'*
⟨*proof*⟩ **lemma** *aux6*:
assumes
a $\in$# *wait*
start-subsumed passed wait
$\forall$ *s* $\in$ *set-mset* (*wait* $-$ {#*a*#}) $\cup$ {*a'*. *E a a'* $\wedge$ $\neg$ *empty a'*}. $\exists$ *s'* $\in$ *set-mset wait'*. *s* $\preceq$ *s'*
shows *start-subsumed* (*insert a passed*) *wait'*
⟨*proof*⟩

lemma *empty-E-star*:

empty x' **if** E^{**} *x* x' *reachable x empty x*
⟨*proof*⟩

lemma *aux4*:
assumes *pw-inv-frontier passed* {#} *reachable x start-subsumed passed* {#}
passed ⊆ {*a. reachable a* ∧ ¬ *empty a*} ¬ *empty x*
shows ∃ x' ∈ *passed*. $x \preceq x'$
⟨*proof*⟩

lemmas [*intro*] = *reachable-step*

private lemma *aux7*:
assumes
a ∈# *wait*
set-mset wait ⊆ *Collect reachable*
set-mset wait' ⊆ *set-mset* (*wait* − {#*a*#}) ∪ *Collect* (*E a*)
x ∈# *wait'*
shows *reachable x*
⟨*proof*⟩ **lemma** *aux8*:
x ∈ *reachable-subsumed* S' **if** *x* ∈ *reachable-subsumed S* ∀ *s*∈*S*. ∃ *x*∈S'. $s \preceq x$
⟨*proof*⟩ **lemma** *aux9*:
assumes
set-mset wait' ⊆ *set-mset* (*wait* − {#*a*#}) ∪ *Collect* (*E a*)
x ∈# *wait'* ∀ a'. *E a* a' ⟶ ¬ *F* a' *F x*
∀ *a*∈*passed* ∪ *set-mset wait*. ¬ *F a*
shows *False*
⟨*proof*⟩ **lemma** *aux10*:
assumes ∀ *a*∈*passed'* ∪ *set-mset wait*. ¬ *F a F x x* ∈# *wait* − {#*a*#}
shows *False*
⟨*proof*⟩

lemma *aux12*:
size wait' < *size wait* **if** *wait'* ⊆# *wait* − {#*a*#} *a* ∈# *wait*
⟨*proof*⟩

lemma *aux13*:
assumes
passed ⊆ {*a. reachable a* ∧ ¬ *empty a*}
passed' ⊆ *insert a* (*passed* ∪ {a'. *E a* a' ∧ ¬ *empty* a'})
¬ *empty a*
reachable a
∀ *s*∈*passed*. ∃ *x*∈*passed'*. $s \preceq x$

$a'' \in passed'$
$\forall\, x{\in}passed.\ \neg\ a'' \preceq x$
shows
$passed' \subseteq \{a.\ reachable\ a \wedge \neg\ empty\ a\} \wedge reachable\text{-}subsumed\ passed \subset reachable\text{-}subsumed\ passed'$
$\vee\ passed' = passed \wedge size\ wait'' < size\ wait$
⟨*proof*⟩

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

end — Context

end — Search Space

theorem (**in** *Search-Space′-finite*) *pw-algo-correct*:
$pw\text{-}algo \le SPEC\ (\lambda\ (brk,\ passed).$
$(brk \longleftrightarrow F\text{-}reachable)$
$\wedge\ (\neg\ brk \longrightarrow$
$(\forall\ \ a.\ reachable\ a \wedge \neg\ empty\ a \longrightarrow (\exists\ \ b \in passed.\ a \preceq b))$
$\wedge\ passed \subseteq \{a.\ reachable\ a \wedge \neg\ empty\ a\})$
$)$
⟨*proof*⟩

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

end — End of Theory
theory *Unified-PW-Hashing*
imports
Unified-PW
Refine-Imperative-HOL.IICF-List-Mset
Worklist-Algorithms-Misc
Worklist-Algorithms-Tracing
begin

3.3 Towards an Implementation of the Unified Passed-Waiting List

context *Worklist1-Defs*
begin

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

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

end — Worklist1 Defs

context *Worklist1*
begin

lemma *add-pw-unified-spec-ref*:
add-pw-unified-spec passed wait a ≤ *add-pw-spec passed wait a*
if *reachable a a* ∈ *passed*
⟨*proof*⟩

lemma *add-pw-ref*:
add-pw passed wait a ≤ ⇓ *Id* (*add-pw-unified-spec passed wait a*)
⟨*proof*⟩

end — Worklist 1

context *Worklist2-Defs*
begin

definition *add-pw' passed wait a* ≡

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

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

end — Worklist 2 Defs

context *Worklist2*
begin

lemma *empty-subsumes′2*:
empty x ∨ *x* ⊴ *y* ⟷ *x* ⪯ *y*
⟨*proof*⟩

lemma *bex-or*:
$P \vee (\exists\ x \in S.\ Q\ x) \longleftrightarrow (\exists\ x \in S.\ P \vee Q\ x)$ **if** $S \neq \{\}$
⟨*proof*⟩

lemma *add-pw′-ref′*:
add-pw′ passed wait a $\leq \Downarrow$ (*Id* $\cap$ $\{((p,\ w,\ \text{-}),\ \text{-}).\ p \neq \{\} \wedge$ *set-mset* $w \subseteq p\}$) (*add-pw passed wait a*)
if *passed* $\neq \{\}$ *set-mset wait* $\subseteq$ *passed*
⟨*proof*⟩

lemma *add-pw′-ref1* [*refine*]:
add-pw′ passed wait a
$\leq \Downarrow$ (*Id* $\cap$ $\{((p,\ w,\ \text{-}),\ \text{-}).\ p \neq \{\} \wedge$ *set-mset* $w \subseteq p\}$) (*add-pw-spec passed′ wait′ a′*)
if *passed* $\neq \{\}$ *set-mset wait* $\subseteq$ *passed reachable a a* $\in$ *passed*
and [*simp*]: *passed* = *passed′ wait* = *wait′ a* = *a′*
⟨*proof*⟩

lemma *refine-weaken*:
$p \leq \Downarrow R\ p'$ **if** $p \leq \Downarrow S\ p'\ S \subseteq R$
⟨*proof*⟩

lemma *add-pw′-ref*:
add-pw′ passed wait a $\leq$
$\Downarrow (\{((p,\ w,\ b),\ (p',\ w',\ b')).\ p \neq \{\} \wedge p = p' \cup$ *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 a* $\in$ *passed*
and [*simp*]: *passed* = *passed′ wait* = *wait′ a* = *a′*
⟨*proof*⟩

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

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
⟨*proof*⟩

lemma [*refine*]:

take-from-mset wait ≤
  ⇓ {((*x*, *wait*), (*y*, *wait'*)). $x = y \wedge wait = wait' \wedge$ *set-mset wait* ⊆ *passed* ∧ *x* ∈ *passed*}
    (*take-from-mset wait'*)
  if *wait* = *wait'* *set-mset wait* ⊆ *passed* *wait* ≠ {#}
  ⟨*proof*⟩

lemma *pw-algo-unified-ref*:
  pw-algo-unified ≤ ⇓ *Id* *pw-algo*
  ⟨*proof*⟩

end — Worklist 2

Utilities **definition** *take-from-list* **where**
  take-from-list s = *ASSERT* (*s* ≠ []) ≫ *SPEC* (λ (*x*, *s'*). *s* = *x* # *s'*)

lemma *take-from-list-correct*:
  assumes *s* ≠ []
  shows *take-from-list s* ≤ *SPEC* (λ (*x*, *s'*). *s* = *x* # *s'*)
⟨*proof*⟩

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

context *Worklist-Map-Defs*
begin

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

}
)
(*passed*,*wait*,*False*)

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

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

end — Worklist Map Defs

lemma *ran-upd-cases*:
(*x* ∈ *ran m*) ∨ (*x* = *y*) **if** *x* ∈ *ran* (*m*(*a* ↦ *y*))
⟨*proof*⟩

lemma *ran-upd-cases2*:
(∃ *k*. *m k* = *Some x* ∧ *k* ≠ *a*) ∨ (*x* = *y*) **if** *x* ∈ *ran* (*m*(*a* ↦ *y*))
⟨*proof*⟩

context *Worklist-Map*
begin

lemma *add-pw′-map-ref*[*refine*]:
add-pw′-map passed wait a $\leq \Downarrow$ (*map-set-rel* $\times_r$ *list-mset-rel* $\times_r$ *bool-rel*) (*add-pw′ passed′ wait′ a′*)
if (*passed*, *passed′*) $\in$ *map-set-rel* (*wait*, *wait′*) $\in$ *list-mset-rel* (*a*, *a′*) $\in$ *Id*
⟨*proof*⟩

lemma *init-map-ref*[*refine*]:
(([*key* $a_0 \mapsto \{a_0\}$], [a_0]), $\{a_0\}$, {#a_0#}) $\in$ *map-set-rel* $\times_r$ *list-mset-rel*
⟨*proof*⟩

lemma *take-from-list-ref*[*refine*]:
take-from-list xs $\leq \Downarrow$ (*Id* $\times_r$ *list-mset-rel*) (*take-from-mset ms*) **if** (*xs*, *ms*) $\in$ *list-mset-rel*
⟨*proof*⟩

lemma *pw-algo-map-ref*:
pw-algo-map $\leq \Downarrow$ (*Id* $\times_r$ *map-set-rel*) *pw-algo-unified*
⟨*proof*⟩

end — Worklist Map

context *Worklist-Map2-Defs*
begin

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

```
            (passed(k ↦ (insert a passed')), a # wait, False)
        )
      }
    )
    (passed,wait,False)
```

definition *pw-algo-map2* **where**

```
  pw-algo-map2 = do
    {
      if F 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-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 $\leq \Downarrow$ *Id* (*add-pw'-map passed' wait' a'*)
if (*passed*, *passed'*) $\in$ *Id* (*wait*, *wait'*) $\in$ *Id* (*a*, *a'*) $\in$ *Id*
⟨*proof*⟩

lemma *pw-algo-map2-ref*[*refine*]:
pw-algo-map2 $\leq \Downarrow$ *Id pw-algo-map*

⟨*proof*⟩

end — Worklist Map 2

lemma (**in** *Worklist-Map2-finite*) *pw-algo-map2-correct*:
pw-algo-map2 ≤ *SPEC* (λ (*brk*, *passed*).
(*brk* ⟷ *F-reachable*) ∧
(¬ *brk* ⟶
(∃ *p*.
(*passed*, *p*) ∈ *map-set-rel* ∧ (∀ *a*. *reachable a* ∧ ¬ *empty a* ⟶ (∃ *b*∈*p*. *a* ⪯ *b*))
∧ *p* ⊆ {*a*. *reachable a* ∧ ¬ *empty a*})
)
)
⟨*proof*⟩

end — End of Theory

3.4 Heap Hash Map

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

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

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

interpretation *comp-fun-commute-mult*:
comp-fun-commute (∗) :: (*'a* :: *ab-semigroup-mult* ⇒ - ⇒ -)
⟨*proof*⟩

lemma *sep-big-star-insert* [*simp*]: ⋀∗ (*add-mset x S*) = (*x* ∗ ⋀∗ *S*)
⟨*proof*⟩

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

lemma *sep-big-star-empty* [*simp*]: ⋀∗ {#} = *emp*
⟨*proof*⟩

lemma *big-star-entatilst-mono*:
$\bigwedge\!* \ T \Longrightarrow_t \bigwedge\!* \ S$ **if** $S \subseteq\# \ T$
⟨*proof*⟩

definition *map-assn V m mi* ≡
$\uparrow$ (*dom mi* = *dom m* ∧ *finite* (*dom m*)) ∗
($\bigwedge\!*$ {# *V* (*the* (*m k*)) (*the* (*mi k*)) . *k* ∈# *mset-set* (*dom m*)#})

lemma *map-assn-empty-map*[*simp*]:
map-assn A Map.empty Map.empty = *emp*
⟨*proof*⟩

lemma *in-mset-union-split*:
mset-set S = *mset-set* (*S* − {*k*}) + {#*k*#} **if** *k* ∈ *S finite S*
⟨*proof*⟩

lemma *in-mset-dom-union-split*:
mset-set (*dom m*) = *mset-set* (*dom m* − {*k*}) + {#*k*#} **if** *m k* = *Some v finite* (*dom m*)
⟨*proof*⟩

lemma *dom-remove-not-in-dom-simp*[*simp*]:
dom m − {*k*} = *dom m* **if** *m k* = *None*
⟨*proof*⟩

lemma *map-assn-delete*:
map-assn A m mh $\Longrightarrow_A$
map-assn A (*m*(*k* := *None*)) (*mh*(*k* := *None*)) ∗ *option-assn A* (*m k*) (*mh k*)
⟨*proof*⟩

lemma *in-mset-set-iff-in-set*[*simp*]:
z ∈# *mset-set S* ⟷ *z* ∈ *S* **if** *finite S*
⟨*proof*⟩

lemma *ent-refl'*:
a = *b* ⟹ *a* $\Longrightarrow_A$ *b*
⟨*proof*⟩

lemma *map-assn-update-aux*:
map-assn A m mh ∗ *A v vi* $\Longrightarrow_A$ *map-assn A* (*m*(*k* ↦ *v*)) (*mh*(*k* ↦ *vi*))
if *k* ∉ *dom m*

⟨*proof*⟩

lemma *map-assn-update*:
  map-assn A m mh ∗ *A v vi* $\Longrightarrow_A$
   map-assn A (*m*(*k* ↦ *v*)) (*mh*(*k* ↦ *vi*)) ∗ *true*
  ⟨*proof*⟩
definition (**in** *imp-map*) *hms-assn A m mi* ≡ $\exists_A$*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* ≡ *empty*

lemma (**in** *imp-map-empty*) *hms-empty-rule* [*sep-heap-rules*]:
  <*emp*> *hms-empty* <*hms-assn A Map.empty*>$_t$
  ⟨*proof*⟩

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

lemma (**in** *imp-map-update*) *hms-update-rule* [*sep-heap-rules*]:
  <*hms-assn A m mi* ∗ *A v vi*> *hms-update k vi mi* <*hms-assn A* (*m*(*k* ↦ *v*))>$_t$
  ⟨*proof*⟩

lemma *restrict-not-in-dom-simp*[*simp*]:
  m |‘ (− {*k*}) = *m* **if** *m k* = *None*
  ⟨*proof*⟩

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

definition [*code*]:

hms-lookup lookup copy k m =
  do {
    vo ← *lookup k m*;
    case vo of
      None ⇒ *return None* |
      Some v ⇒ *do* {
        v′ ← *copy v*;
        return (*Some v*′)
      }
  }

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

lemma *map-assn-domain-simps*[*simp*]:
  assumes *vassn-tag* (*map-assn A m mh*)
  shows *mh k* = *None* ⟷ *m k* = *None dom mh* = *dom m finite* (*dom m*)
  ⟨*proof*⟩

lemma *hms-extract-rule* [*sep-heap-rules*]:
  <*hms-assn A m mi*>
    hms-extract lookup delete k mi
  <λ (*vi*, *mi*′). *option-assn A* (*m k*) *vi* ∗ *hms-assn A* (*m*(*k* := *None*)) *mi*′>$_t$
  ⟨*proof*⟩

lemma *hms-lookup-rule* [*sep-heap-rules*]:
  assumes
    (*copy*, *RETURN o COPY*) ∈ $A^k \to_a A$
  shows
  <*hms-assn A m mi*>
    hms-lookup lookup copy k mi
  <λ *vi*. *hms-assn A m mi* ∗ *option-assn A* (*m k*) *vi*>$_t$
⟨*proof*⟩

end

context *imp-map-update*
begin

lemma *hms-update-hnr*:
  (*uncurry2 hms-update*, *uncurry2* (*RETURN ooo op-map-update*)) ∈
  id-assn^k $*_a$ A^d $*_a$ (*hms-assn A*)^d $\to_a$ *hms-assn A*
  ⟨*proof*⟩

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

end

context *imp-map-empty*
begin

lemma *hms-empty-hnr*:
($uncurry0$ *hms-empty*, $uncurry0$ (*RETURN op-map-empty*)) $\in$ *unit-assn*^{k} $\rightarrow_a$ *hms-assn A*
⟨*proof*⟩

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

definition *op-hms-empty* $\equiv$ *IICF-Map.op-map-empty*

sublocale *hms*: *map-custom-empty op-hms-empty*
⟨*proof*⟩

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*:
$\lambda k\ m.\ (m\ k,\ m(k := None))$ :: $K \rightarrow \langle K,V\rangle$*map-rel* $\rightarrow \langle V\rangle$*option-rel* $\times_r$ $\langle K,V\rangle$*map-rel*
where *single-valued K single-valued* (K^{-1})
⟨*proof*⟩

context *imp-map-extract-derived*
begin

lemma *hms-extract-hnr*:
(*uncurry* (*hms-extract lookup delete*), *uncurry* (*RETURN oo op-map-extract*)) $\in$
id-assn^k $*_a$ (*hms-assn A*)^d $\rightarrow_a$ *prod-assn* (*option-assn A*) (*hms-assn A*)
⟨*proof*⟩

lemma *hms-lookup-hnr*:
$(uncurry\ (hms\text{-}lookup\ lookup\ copy),\ uncurry\ (RETURN\ oo\ op\text{-}map\text{-}lookup))$
$\in$
$id\text{-}assn^k *_a (hms\text{-}assn\ A)^k \rightarrow_a option\text{-}assn\ A$ **if** $(copy,\ RETURN\ o\ COPY)$
$\in A^k \rightarrow_a A$
⟨*proof*⟩

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

end

interpretation *hms-hm*: *imp-map-extract-derived is-hashmap hm-delete hm-lookup*
⟨*proof*⟩

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*: $\lambda\ (x\ \#\ xs) \Rightarrow (x,\ xs) :: [\lambda l.\ l{\neq}[]]_f\ \langle A\rangle list\text{-}rel \rightarrow A \times_r \langle A\rangle list\text{-}rel$
⟨*proof*⟩
end

context *Worklist-Map2-Defs*
begin

definition *trace* **where**
trace ≡ λ*type a*. *RETURN* ()

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

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

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

definition
  subsumed-string = *''Subsumed''*

definition
  empty-string = *''Empty''*

lemma *add-pw'-map2-alt-def*:
```
add-pw'-map2 passed wait a = do {
 trace explored-string a;
 nfoldli (succs a) (λ(-, -, brk). ¬brk)
  (λa (passed, wait, -).
    do {
    RETURN (
      if empty a then
          (passed, wait, False)
      else if F' a then (passed, wait, True)
      else
        let
          k = key a;
          (v, passed) = op-map-extract k passed
        in
          case v of
            None ⇒ (passed(k ↦ {COPY a}), a # wait, False) |
            Some passed' ⇒
              if ∃ x ∈ passed'. a ⊴ x then
                (passed(k ↦ passed'), wait, False)
              else
                (passed(k ↦ (insert (COPY a) passed')), a # wait, False)
      )
    }
  )
  (passed,wait,False)
}
⟨proof⟩
```

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

if empty a then
do {*trace empty-string a*; *RETURN* (*passed*, *wait*, *False*)}
else if F' *a then do* {*trace final-string a*; *RETURN* (*passed*, *wait*, *True*)}
else
let
k = *key a*;
(*v*, *passed*) = *op-map-extract k passed*
in
case v of
None ⇒ *do* {*trace added-string a*; *RETURN* (*passed*($k \mapsto$ {*COPY a*}), *a # wait*, *False*)} |
Some passed' ⇒
if ∃ *x* ∈ *passed'*. *a* ⊴ *x then*
do {*trace subsumed-string a*; *RETURN* (*passed*($k \mapsto$ *passed'*), *wait*, *False*)}
else do {
trace added-string a;
RETURN (*passed*($k \mapsto$ (*insert* (*COPY a*) *passed'*)), *a # wait*, *False*)
}
}
)
(*passed*,*wait*,*False*)
}
⟨*proof*⟩

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*) ∈ $A^k \to_a K$
assumes [*sepref-fr-rules*]: (*copyi*, *RETURN o COPY*) ∈ $A^k \to_a A$
assumes [*sepref-fr-rules*]: (*uncurry tracei*,*uncurry trace*) ∈ $id\text{-}assn^k *_a A^k \to_a id\text{-}assn$
assumes *pure-K*: *is-pure K*
assumes *left-unique-K*: *IS-LEFT-UNIQUE* (*the-pure K*)
assumes *right-unique-K*: *IS-RIGHT-UNIQUE* (*the-pure K*)
begin
sepref-register
PR-CONST a_0 *PR-CONST* F' *PR-CONST* (⊴) *PR-CONST succs PR-CONST empty PR-CONST key*

PR-CONST F trace

lemma [*def-pat-rules*]:
a_0 ≡ *UNPROTECT* a_0 *F′* ≡ *UNPROTECT F′* (⊴) ≡ *UNPROTECT* (⊴) *succs* ≡ *UNPROTECT succs*
empty ≡ *UNPROTECT empty keyi* ≡ *UNPROTECT keyi F* ≡ *UNPROTECT F key* ≡ *UNPROTECT key*
⟨*proof*⟩

lemma *take-from-list-alt-def*:
take-from-list xs = *do* {- ← *ASSERT* (*xs* ≠ []); *RETURN* (*hd-tl xs*)}
⟨*proof*⟩

lemma [*safe-constraint-rules*]: *CN-FALSE is-pure A* ⟹ *is-pure A* ⟨*proof*⟩

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*) ∈ *Id*
(*subsumed-string*, *subsumed-string*) ∈ *Id*
(*added-string*, *added-string*) ∈ *Id*
(*final-string*, *final-string*) ∈ *Id*
(*empty-string*, *empty-string*) ∈ *Id*
⟨*proof*⟩

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*}) :: $unit\text{-}assn^k \rightarrow_a$ *bool-assn*
⟨*proof*⟩

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

end — Worklist Map2 Impl

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

lemma *pw-algo-map2-correct'*:
(*do* {(*r*, *p*) ← *pw-algo-map2*; *RETURN r*}) ≤ *SPEC* (λ*brk*. *brk* = *F-reachable*)
⟨*proof*⟩

lemma *pw-impl-hnr-F-reachable*:
(*uncurry0* (*pw-impl keyi copyi tracei Lei* $a_0 i$ *Fi succsi emptyi*), *uncurry0* (*RETURN F-reachable*))
∈ *unit-assn*^{k} $\rightarrow_a$ *bool-assn*
⟨*proof*⟩

end

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

sepref-decl-op *F-reachable* :: *bool-rel* ⟨*proof*⟩
lemma [*def-pat-rules*]: *F-reachable* ≡ *op-F-reachable* ⟨*proof*⟩

lemma *hnr-op-F-reachable*:
assumes *GEN-ALGO* $a_0 i$ (λ$a_0 i$. (*uncurry0* $a_0 i$, *uncurry0* (*RETURN* a_0)) ∈ *unit-assn*^{k} $\rightarrow_a$ *A*)
assumes *GEN-ALGO Fi* (λ*Fi*. (*Fi*,*RETURN o F'*) ∈ A^k $\rightarrow_a$ *bool-assn*)
assumes *GEN-ALGO Lei* (λ*Lei*. (*uncurry Lei*,*uncurry* (*RETURN oo* (⊴))) ∈ A^k $*_a$ A^k $\rightarrow_a$ *bool-assn*)
assumes *GEN-ALGO succsi* (λ*succsi*. (*succsi*,*RETURN o succs*) ∈ A^k $\rightarrow_a$ *list-assn A*)
assumes *GEN-ALGO emptyi* (λ*Fi*. (*Fi*,*RETURN o empty*) ∈ A^k $\rightarrow_a$ *bool-assn*)
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*
shows
(*uncurry0* (*pw-impl keyi copyi tracei Lei* $a_0 i$ *Fi succsi emptyi*),
uncurry0 (*RETURN* (*PR-CONST op-F-reachable*)))
∈ *unit-assn*^{k} $\rightarrow_a$ *bool-assn*
⟨*proof*⟩

sepref-decl-impl *hnr-op-F-reachable* ⟨*proof*⟩

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

lemma *take-from-set-correct*:
  assumes $s \neq \{\}$
  shows $take\text{-}from\text{-}set\ s \leq SPEC\ (\lambda\ (x,\ s').\ x \in s \wedge s' = s - \{x\})$
⟨*proof*⟩

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

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

lemma *take-from-mset-correct*:
  assumes $s \neq \{\#\}$
  shows $take\text{-}from\text{-}mset\ s \leq SPEC\ (\lambda\ (x,\ s').\ x \in\#\ s \wedge s' = s - \{\#x\#\})$
⟨*proof*⟩

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

lemma *set-mset-mp*: *set-mset m* ⊆ *s* ⟹ *n* < *count m x* ⟹ *x*∈*s*
⟨*proof*⟩

lemma *pred-not-lt-is-zero*: (¬ *n* − *Suc 0* < *n*) ⟷ *n*=*0* ⟨*proof*⟩

lemma (**in** *Search-Space-finite-strict*) *finitely-branching*:
assumes *reachable a*
shows *finite* (*Collect* (*E a*))
⟨*proof*⟩

4.2 Standard Worklist Algorithm

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

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

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

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

definition *worklist-inv* ≡ λ (*passed*, *wait*, *brk*).
passed ⊆ *Collect reachable* ∧
(*brk* ⟶ (∃ *f*. *reachable f* ∧ *F f*)) ∧
(¬ *brk* ⟶
worklist-inv-frontier passed wait
∧ (∀ *a* ∈ *passed* ∪ *set-mset wait*. ¬ *F a*)
∧ *worklist-start-subsumed passed wait*
∧ *set-mset wait* ⊆ *Collect reachable*)

definition *add-succ-spec wait a* ≡ *SPEC* (λ(*wait*′,*brk*).
if ∃ *a*′. *E a a*′ ∧ *F a*′ *then*
brk

else
$\neg$*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'*)
)

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* (λ (*passed*, *wait*, *brk*). $\neg$ *brk* $\wedge$ *wait* $\neq$ {#})
(λ (*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*) *else*
do
{
(*wait*,*brk*) $\leftarrow$ *add-succ-spec wait a*;
let passed = *insert a passed*;
RETURN (*passed*, *wait*, *brk*)
}
}
)
(*passed*, *wait*, *False*);
RETURN brk
}
}

end

Correctness Proof **lemma** (**in** *Search-Space*) *empty-E-star*:
empty x' **if** E^{**} *x x' reachable x empty x*
⟨*proof*⟩

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

lemma *wf-worklist-var*:
wf worklist-var
⟨*proof*⟩

context
begin

private lemma *aux1*:
assumes $\forall x \in passed.\ \neg\ a \preceq x$
and $passed \subseteq Collect\ reachable$
and *reachable a*
shows
$((insert\ a\ passed,\ wait',\ brk'),$
$passed,\ wait,\ brk)$
$\in worklist\text{-}var$
⟨*proof*⟩ **lemma** *aux2*:
assumes
$a' \in passed$
$a \preceq a'$
$a \in\# wait$
worklist-inv-frontier passed wait
shows *worklist-inv-frontier passed* $(wait - \{\#a\#\})$
⟨*proof*⟩ **lemma** *aux5*:
assumes
$a' \in passed$
$a \preceq a'$
$a \in\# wait$
worklist-start-subsumed passed wait
shows *worklist-start-subsumed passed* $(wait - \{\#a\#\})$
⟨*proof*⟩ **lemma** *aux3*:
assumes
$set\text{-}mset\ wait \subseteq Collect\ reachable$
$a \in\# wait$
$\forall\ s \in set\text{-}mset\ (wait - \{\#a\#\}) \cup \{a'.\ E\ a\ a' \wedge \neg\ empty\ a'\}.\ \exists\ s' \in set\text{-}mset\ wait'.\ s \preceq s'$
worklist-inv-frontier passed wait
shows *worklist-inv-frontier* $(insert\ a\ passed)\ wait'$
⟨*proof*⟩ **lemma** *aux6*:
assumes
$a \in\# wait$
worklist-start-subsumed passed wait
$\forall\ s \in set\text{-}mset\ (wait - \{\#a\#\}) \cup \{a'.\ E\ a\ a' \wedge \neg\ empty\ a'\}.\ \exists\ s' \in set\text{-}mset\ wait'.\ s \preceq s'$
shows *worklist-start-subsumed* $(insert\ a\ passed)\ wait'$

⟨*proof*⟩

lemma *aux4*:
assumes *worklist-inv-frontier passed* {#} *reachable x worklist-start-subsumed passed* {#}
passed ⊆ *Collect reachable*
shows ∃ *x′* ∈ *passed*. *x* ≼ *x′*
⟨*proof*⟩

theorem *worklist-algo-correct*:
worklist-algo ≤ *SPEC* (λ *brk*. *brk* ⟷ *F-reachable*)
⟨*proof*⟩

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

end — Context

end — Search Space

context *Search-Space″-Defs*
begin

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

definition *worklist-start-subsumed′ passed wait* = (∃ *a* ∈ *passed* ∪ *set-mset wait*. a_0 ≼ *a*)

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

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

)

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

end — Search Space” Defs

context *Search-Space″-start*
begin

lemma *worklist-algo-list-inv-ref*[*refine*]:
fixes *x x′*
assumes
¬ *F* a_0 ¬ *F* a_0
(*x*, *x′*) ∈ {((*passed*,*wait*,*brk*), (*passed′*,*wait′*,*brk′*)).
passed = *passed′* ∧ *wait* = *wait′* ∧ *brk* = *brk′* ∧ (∀ *a* ∈ *passed*. ¬ *empty a*)

$\wedge$ ($\forall$ $a \in$ *set-mset wait*. $\neg$ *empty a*)}
worklist-inv x'
shows *worklist-inv' x*
⟨*proof*⟩

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* $\neq$ {#}
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *worklist-algo'-ref*[*refine*]: *worklist-algo'* $\leq$ $\Downarrow$*Id worklist-algo*
⟨*proof*⟩

end — Search Space" Start

context *Search-Space''-Defs*
begin

definition *worklist-algo''* **where**
worklist-algo'' $\equiv$
if empty a_0 *then RETURN False else worklist-algo'*

end — Search Space" Defs

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

lemma *worklist-algo''-correct*:
worklist-algo'' $\leq$ *SPEC* (λ *brk*. *brk* $\longleftrightarrow$ *F-reachable*)
⟨*proof*⟩

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

lemma *take-from-list-correct*:
  assumes $s \neq []$
  shows $take\text{-}from\text{-}list\ s \leq SPEC\ (\lambda\ (x,\ s').\ s = x\ \#\ s')$
⟨*proof*⟩

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

context *Search-Space-Defs-Empty*
begin

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

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

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

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

end — Search Space Empty Defs

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

end — Search Space” Defs

context *Search-Space''-pre*
begin

lemma *worklist-algo-list-inv-ref*:
fixes $x\ x'$
assumes
$\neg\ F\ a_0\ \neg\ F\ a_0$
$(x,\ x') \in \{((passed,wait,brk),\ (passed',wait',brk')).\ passed = passed' \land mset\ wait = wait' \land brk = brk'\}$
worklist-inv' x'
shows *worklist-inv-list* x
⟨*proof*⟩

lemma *take-from-list-take-from-mset-ref*[*refine*]:
take-from-list $xs \leq \Downarrow \{((x,\ xs),(y,\ m)).\ x = y \land mset\ xs = m\}$ (*take-from-mset* m)
if $mset\ xs = m$
⟨*proof*⟩

lemma *add-succ-spec-list-add-succ-spec-ref*[*refine*]:
add-succ-spec-list $xs\ b \leq \Downarrow \{((xs,\ b),\ (m,\ b')).\ mset\ xs = m \land b = b'\}$ (*add-succ-spec'* $m\ b'$)
if $mset\ xs = m\ b = b'$
⟨*proof*⟩

lemma *worklist-algo-list-ref*[*refine*]: *worklist-algo-list* $\leq \Downarrow Id$ *worklist-algo'*
⟨*proof*⟩

end — Search Space"

4.4 Towards an Implementation

context *Worklist1-Defs*
begin

definition
add-succ1 wait a ≡
nfoldli (*succs a*) (λ(-,*brk*). ¬*brk*)
(λ*a* (*wait*,*brk*).
do {
ASSERT ($\forall\ x \in set\ wait.\ \neg\ empty\ x$);
if F a then RETURN (*wait*,*True*) *else RETURN* (
if ($\exists\ x \in set\ wait.\ a \preceq x \land \neg\ x \preceq a$) $\lor$ *empty a*

then [$x \leftarrow$ *wait*. $\neg$ $x \preceq a$]
else a # [$x \leftarrow$ *wait*. $\neg$ $x \preceq a$],*False*)
}
)
(*wait*,*False*)

end

context *Worklist2-Defs*
begin

definition
add-succ1' *wait* $a \equiv$
nfoldli (*succs* a) (λ(-,*brk*). $\neg$*brk*)
(λa (*wait*,*brk*).
if F a *then* *RETURN* (*wait*,*True*) *else* *RETURN* (
if *empty* a
then *wait*
else *if* $\exists$ $x \in$ *set* *wait*. $a \trianglelefteq x \wedge \neg$ $x \trianglelefteq a$
then [$x \leftarrow$ *wait*. $\neg$ $x \trianglelefteq a$]
else a # [$x \leftarrow$ *wait*. $\neg$ $x \trianglelefteq a$],*False*)
)
(*wait*,*False*)

end

context *Worklist1*
begin

lemma *add-succ1-ref*[*refine*]:
add-succ1 *wait* $a \le \Downarrow$(*Id* $\times_r$ *bool-rel*) (*add-succ-spec-list* *wait'* *a'*)
if (*wait*,*wait'*)$\in$*Id* (*a*,*a'*)$\in$*b-rel* *Id* *reachable* $\forall$ $x \in$ *set* *wait'*. $\neg$ *empty* x
⟨*proof*⟩

end

context *Worklist2*
begin

lemma *add-succ1'-ref*[*refine*]:
add-succ1' *wait* $a \le \Downarrow$(*Id* $\times_r$ *bool-rel*) (*add-succ1* *wait'* *a'*)
if (*wait*,*wait'*)$\in$*Id* (*a*,*a'*)$\in$*b-rel* *Id* *reachable* $\forall$ $x \in$ *set* *wait'*. $\neg$ *empty* x
⟨*proof*⟩

lemma *add-succ1′-ref′*[*refine*]:
add-succ1′ wait a $\leq \Downarrow$(*Id* $\times_r$ *bool-rel*) (*add-succ-spec-list wait′ a′*)
if (*wait*,*wait′*)∈*Id* (*a*,*a′*)∈*b-rel Id reachable* ∀ *x* ∈ *set wait′*. ¬ *empty x*
⟨*proof*⟩

definition *worklist-algo1′* **where**
worklist-algo1′ = *do*
{
if F a_0 *then RETURN True*
else do {
let passed = {};
let wait = [a_0];
(*passed*, *wait*, *brk*) ← *WHILEIT worklist-inv-list* (λ (*passed*, *wait*, *brk*). ¬ *brk* ∧ *wait* ≠ [])
(λ (*passed*, *wait*, *brk*). *do*
{
(*a*, *wait*) ← *take-from-list wait*;
if (∃ *a′* ∈ *passed*. *a* ⊴ *a′*) *then RETURN* (*passed*, *wait*, *brk*) *else*
do
{
(*wait*,*brk*) ← *add-succ1′ wait a*;
let passed = *insert a passed*;
RETURN (*passed*, *wait*, *brk*)
}
}
)
(*passed*, *wait*, *False*);
RETURN brk
}
}

lemma *take-from-list-ref*[*refine*]:
take-from-list wait ≤
⇓ {((*x*, *wait*), (*y*, *wait′*)). *x* = *y* ∧ *wait* = *wait′* ∧ ¬ *empty x* ∧ (∀ *a* ∈ *set wait*. ¬ *empty a*)}
(*take-from-list wait′*)
if *wait* = *wait′* ∀ *a* ∈ *set wait*. ¬ *empty a wait* ≠ []
⟨*proof*⟩

lemma *worklist-algo1-list-ref*[*refine*]: *worklist-algo1′* ≤ ⇓*Id worklist-algo-list*
⟨*proof*⟩

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

lemma *worklist-algo1-ref*[*refine*]: *worklist-algo1* ≤ ⇓*Id worklist-algo''*
⟨*proof*⟩

end — Worklist2

context *Worklist3-Defs*
begin

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

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

end

context *Worklist3*
begin

lemma *filter-insert-wait-alt-def*:
filter-insert-wait wait a = (
let
(*f*, *xs*) =
fold (λ *x* (*f*, *xs*). *if x* ⊴ *a then* (*f*, *xs*) *else* (*f* ∨ *a* ⊴ *x*, *x* # *xs*))
wait (*False*, [])
in

if f then rev xs else a # rev xs
)

⟨*proof*⟩

lemma *add-succ2-alt-def*:
add-succ2 wait a ≡
nfoldli (*succs a*) (λ(-,*brk*). ¬*brk*)
(λ*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*)
⟨*proof*⟩

lemma *add-succ2-ref*[*refine*]:
add-succ2 wait a ≤ ⇓(*Id* $\times_r$ *bool-rel*) (*add-succ1′ wait′ a′*)
if (*wait*,*wait′*)∈*Id* (*a*,*a′*)∈*Id*
⟨*proof*⟩

definition *worklist-algo2′* **where**
worklist-algo2′ = *do*
{
if F′ a_0 *then RETURN True*
else do {
let passed = {};
let wait = [a_0];
(*passed*, *wait*, *brk*) ← *WHILEIT worklist-inv-list* (λ (*passed*, *wait*, *brk*). ¬ *brk* ∧ *wait* ≠ [])
(λ (*passed*, *wait*, *brk*). *do*
{
(*a*, *wait*) ← *take-from-list wait*;
if (∃ *a′* ∈ *passed*. *a* ⊴ *a′*) *then RETURN* (*passed*, *wait*, *brk*) *else*
do
{
(*wait*,*brk*) ← *add-succ2 wait a*;
let passed = *insert a passed*;
RETURN (*passed*, *wait*, *brk*)
}
}
)
(*passed*, *wait*, *False*);

RETURN brk
}
}

lemma *worklist-algo2′-ref* [*refine*]: *worklist-algo2′* ≤ ⇓*Id worklist-algo1′* **if** ¬ *empty* a_0
⟨*proof*⟩

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

lemma *worklist-algo2-ref* [*refine*]: *worklist-algo2* ≤ ⇓*Id worklist-algo″*
⟨*proof*⟩
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*⟩

context notes [*split*!] = *list.split* **begin**
sepref-decl-op *list-hdtl*: λ (*x # xs*) ⇒ (*x*, *xs*) :: $[\lambda l.\ l \neq []]_f$ ⟨*A*⟩*list-rel* → *A* $\times_r$ ⟨*A*⟩*list-rel*
⟨*proof*⟩
end

context *Worklist4-Impl*
begin
sepref-register *PR-CONST* a_0 *PR-CONST* *F′* *PR-CONST* (⊴) *PR-CONST* *succs* *PR-CONST* *empty*

lemma [*def-pat-rules*]:
a_0 ≡ *UNPROTECT* a_0 *F′* ≡ *UNPROTECT* *F′* (⊴) ≡ *UNPROTECT* (⊴) *succs* ≡ *UNPROTECT succs*
empty ≡ *UNPROTECT empty*

⟨*proof*⟩

lemma *take-from-list-alt-def*:
take-from-list xs = *do* {- ← *ASSERT* (*xs* ≠ []); *RETURN* (*hd-tl xs*)}
⟨*proof*⟩

lemma [*safe-constraint-rules*]: *CN-FALSE is-pure A* ⟹ *is-pure A* ⟨*proof*⟩

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*
⟨*proof*⟩

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

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*
⟨*proof*⟩

concrete-definition (**in** −) *worklist-algo2-impl*
for *Lei* a_0i *Fi succsi emptyi*
uses *Worklist4-Impl.worklist-algo2-impl.refine-raw* **is** (*uncurry0 ?f*,-)∈-

end — Worklist4 Impl

context *Worklist4-Impl-finite-strict*
begin

lemma *worklist-algo2-impl-hnr-F-reachable*:
(*uncurry0* (*worklist-algo2-impl Lei* a_0i *Fi succsi emptyi*), *uncurry0* (*RETURN F-reachable*))
∈ *unit-assn*^{k} $\rightarrow_a$ *bool-assn*
⟨*proof*⟩

sepref-decl-op *F-reachable* :: *bool-rel* ⟨*proof*⟩
lemma [*def-pat-rules*]: *F-reachable* ≡ *op-F-reachable* ⟨*proof*⟩

lemma *hnr-op-F-reachable*:
assumes *GEN-ALGO* a_0i ($\lambda a_0 i$. (*uncurry0* a_0i, *uncurry0* (*RETURN* a_0)) ∈ *unit-assn*^{k} $\rightarrow_a$ *A*)
assumes *GEN-ALGO Fi* (λ*Fi*. (*Fi*,*RETURN o F′*) ∈ A^k $\rightarrow_a$ *bool-assn*)
assumes *GEN-ALGO Lei* (λ*Lei*. (*uncurry Lei*,*uncurry* (*RETURN oo* (⊴))) ∈ A^k $*_a$ A^k $\rightarrow_a$ *bool-assn*)
assumes *GEN-ALGO succsi* (λ*succsi*. (*succsi*,*RETURN o succs*) ∈ A^k $\rightarrow_a$ *list-assn A*)
assumes *GEN-ALGO emptyi* (λ*Fi*. (*Fi*,*RETURN o empty*) ∈ A^k $\rightarrow_a$ *bool-assn*)
shows
(*uncurry0* (*worklist-algo2-impl Lei* a_0i *Fi succsi emptyi*), *uncurry0* (*RETURN* (*PR-CONST op-F-reachable*)))
∈ *unit-assn*^{k} $\rightarrow_a$ *bool-assn*
⟨*proof*⟩

sepref-decl-impl *hnr-op-F-reachable* ⟨*proof*⟩

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* ⟨*proof*⟩

no-notation *E* (- → - [*100*, *100*] *40*)
notation *G.E′* (- → - [*100*, *100*] *40*)
no-notation *reaches* (- →* - [*100*, *100*] *40*)
notation *G.G′.reaches* (- →* - [*100*, *100*] *40*)

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

Plain set membership is also an option.

definition *check-loop v ST* = (∃ *v′* ∈ *set ST*. *v′* ⪯ *v*)

definition *dfs* :: *′a set* ⇒ (*bool* × *′a set*) *nres* **where**
  dfs P ≡ *do* {
    (*P*,*ST*,*r*) ← *RECT* (λ*dfs* (*P*,*ST*,*v*).
      do {
        if check-loop v ST then RETURN (*P*, *ST*, *True*)
        else do {
          if ∃ *v′* ∈ *P*. *v* ⪯ *v′ then*
            RETURN (*P*, *ST*, *False*)
          else do {
              let ST = *v* # *ST*;
              (*P*, *ST′*, *r*) ←
                  FOREACHcd {*v′*. *v* → *v′*} (λ(-,-,*b*). ¬*b*) (λ*v′* (*P*,*ST*,-). *dfs*
(*P*,*ST*,*v′*)) (*P*,*ST*,*False*);
              ASSERT (*ST′* = *ST*);
              let ST = *tl ST′*;
              let P = *insert v P*;
              RETURN (*P*, *ST*, *r*)
            }
        }
      }
) (*P*,[],a_0);
    RETURN (*r*, *P*)
  }

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

definition *dfs-spec* ≡
  SPEC (λ (*r*, *P*).
    (*r* ⟶ (∃ *x*. a_0 →∗ *x* ∧ *x* $\rightarrow^+$ *x*))
  ∧ (¬ *r* ⟶ ¬ (∃ *x*. a_0 →∗ *x* ∧ *x* $\rightarrow^+$ *x*)
      ∧ *liveness-compatible P* ∧ *P* ⊆ {*x*. *V x*}
)
)

end

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

lemma *check-loop-loop*: $\exists\ v' \in set\ ST.\ v' \preceq v$ **if** *check-loop v ST*
⟨*proof*⟩

lemma *check-loop-no-loop*: $v \notin set\ ST$ **if** $\neg$ *check-loop v ST*
⟨*proof*⟩

lemma *mono*:
$a \preceq b \Longrightarrow a \rightarrow a' \Longrightarrow V\ b \Longrightarrow \exists\ b'.\ b \rightarrow b' \wedge a' \preceq b'$
⟨*proof*⟩

context
fixes P :: *'a set* **and** *E1 E2* :: *'a* ⇒ *'a* ⇒ *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* ⟨*proof*⟩
interpretation *G'*: *Graph-Defs E2* ⟨*proof*⟩
interpretation *SG*: *Subgraph E2 E1* ⟨*proof*⟩
interpretation *SG'*: *Subgraph-Start E* a_0 *E1* ⟨*proof*⟩
interpretation *SG''*: *Subgraph-Start E* a_0 *E2* ⟨*proof*⟩

lemma *G-subgraph-reaches*[*intro*]:
G.G'.reaches a b **if** *G.reaches a b*
⟨*proof*⟩

lemma *G'-subgraph-reaches*[*intro*]:
G.G'.reaches a b **if** *G'.reaches a b*
⟨*proof*⟩

lemma *liveness-compatible-extend*:
assumes
$V\ s\ V\ v\ s \preceq v$
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*
⟨*proof*⟩
including *subgraph-automation* ⟨*proof*⟩
including *subgraph-automation* ⟨*proof*⟩

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*⟩
including *graph-automation-aggressive*
⟨*proof*⟩

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*
⟨*proof*⟩

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* $v\ P$)
⟨*proof*⟩

interpretation *subsumption*: *Subsumption-Graph-Pre-Nodes* $(\preceq)\ (\prec)\ E\ V$
⟨*proof*⟩

lemma *pre-cycle-cycle*:
$(\exists\ x\ x'.\ a_0 \rightarrow{*}\ x \land x \rightarrow^{+}\ x' \land x \preceq x') \longleftrightarrow (\exists\ x.\ a_0 \rightarrow{*}\ x \land x \rightarrow^{+}\ x)$
⟨*proof*⟩

lemma *reachable-alt*:
$V\ v$ **if** $V\ a_0\ a_0 \rightarrow{*}\ v$
⟨*proof*⟩

lemma *dfs-correct*:
  dfs $P \leq$ *dfs-spec* **if** $V\ a_0$ *liveness-compatible* P $P \subseteq \{x.\ V\ x\}$
⟨*proof*⟩
    including *graph-automation*
    ⟨*proof*⟩

end

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

definition *dfs1* :: *'a set* ⇒ (*bool* × *'a set*) *nres* **where**
  dfs1 P ≡ *do* {
    (*P*,*ST*,*r*) ← *RECT* (λ*dfs* (*P*,*ST*,*v*).
      do {
        ASSERT (*V v* ∧ *set ST* ⊆ {*x*. *V x*});
        if check-loop v ST then RETURN (*P*, *ST*, *True*)
        else do {
          if ∃ *v'* ∈ *P*. *v* ⪯ *v'* *then*
            RETURN (*P*, *ST*, *False*)
          else do {
              let ST = *v* # *ST*;
              (*P*, *ST'*, *r*) ←
                nfoldli (*succs v*) (λ(-,-,*b*). ¬*b*) (λ*v'* (*P*,*ST*,-). *dfs* (*P*,*ST*,*v'*))
(*P*,*ST*,*False*);
              ASSERT (*ST'* = *ST*);
              let ST = *tl ST'*;
              let P = *insert v P*;
              RETURN (*P*, *ST*, *r*)
            }
        }
      }
) (*P*,[],a_0);
    RETURN (*r*, *P*)
  }

end

locale *Liveness-Search-Space* =
  Liveness-Search-Space-Defs +
  Liveness-Search-Space-pre +
  assumes *succs-correct*: *V a* ⟹ *set* (*succs a*) = {*x*. *a* → *x*}

assumes *finite-V*: *finite* $\{a.\ V\ a\}$
begin

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

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

lemma *start-ref*[*refine*]:
$((P, [], a_0), P, [], a_0) \in Id \times_r br\ id\ (\lambda\ xs.\ set\ xs \subseteq \{x.\ V\ x\}) \times_r br\ id\ V$ **if** $V\ a_0$
$\langle proof\rangle$

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

lemma *refine-loop*:
$(\bigwedge x\ x'.\ (x, x') \in Id \times_r br\ id\ (\lambda xs.\ set\ xs \subseteq \{x.\ V\ x\}) \times_r br\ id\ V \Longrightarrow$
$dfs'\ x \leq \Downarrow (Id \times_r br\ id\ (\lambda xs.\ set\ xs \subseteq Collect\ V) \times_r bool\text{-}rel)$ $(dfsa\ x')) \Longrightarrow$
$(x, x') \in Id \times_r br\ id\ (\lambda xs.\ set\ xs \subseteq \{x.\ V\ x\}) \times_r br\ id\ V \Longrightarrow$
$x2 = (x1a, x2a) \Longrightarrow$
$x' = (x1, x2) \Longrightarrow$
$x2b = (x1c, x2c) \Longrightarrow$
$x = (x1b, x2b) \Longrightarrow$
$nfoldli\ (succs\ x2c)\ (\lambda(\text{-}, \text{-}, b).\ \neg\ b)\ (\lambda v'\ (P, ST, \text{-}).\ dfs'\ (P, ST, v'))\ (x1b, x2c\ \#\ x1c, False)$
$\leq \Downarrow (Id \times_r br\ id\ (\lambda xs.\ set\ xs \subseteq \{x.\ V\ x\}) \times_r bool\text{-}rel)$
$(FOREACHcd\ \{v'.\ x2a \rightarrow v'\}\ (\lambda(\text{-}, \text{-}, b).\ \neg\ b)$
$(\lambda v'\ (P, ST, \text{-}).\ dfsa\ (P, ST, v'))\ (x1, x2a\ \#\ x1a, False))$
$\langle proof\rangle$

lemma *dfs1-dfs-ref*[*refine*]:
$dfs1\ P \leq \Downarrow Id\ (dfs\ P)$ **if** $V\ a_0$
$\langle proof\rangle$

end

end

5.2 Implementation on Maps

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

locale *Liveness-Search-Space-Key-Defs* =
  Liveness-Search-Space-Defs E **for** *E* :: $'v \Rightarrow 'v \Rightarrow$ *bool* +
  fixes *key* :: $'v \Rightarrow 'k$
  fixes *subsumes'* :: $'v \Rightarrow 'v \Rightarrow$ *bool* (**infix** $\trianglelefteq$ *50*)
begin

sublocale *Search-Space-Key-Defs* ⟨*proof*⟩

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

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

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

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

definition *dfs-map* :: $('k \rightharpoonup 'v\ set) \Rightarrow (bool \times ('k \rightharpoonup 'v\ set))$ *nres* **where**
  dfs-map P ≡ *do* {
    $(P,ST,r) \leftarrow$ *RECT* $(\lambda dfs\ (P,ST,v).$
      if check-subsumption-map-list v ST then RETURN (P, ST, True)

else do {
if check-subsumption-map-set v P then
RETURN $(P, ST, False)$
else do {
let ST = *push-map-list v ST*;
$(P, ST, r) \leftarrow$
nfoldli (*succs v*) $(\lambda(\text{-},\text{-},b).\ \neg b)$ $(\lambda v'\ (P,ST,\text{-}).\ \mathit{dfs}\ (P,ST,v'))$ (P,ST,False);
let ST = *pop-map-list v ST*;
let P = *insert-map-set v P*;
RETURN (P, ST, r)
}
}
) $(P,(\mathit{Map.empty}::('k \rightharpoonup 'v\ \mathit{list})),a_0)$;
RETURN (r, P)
}

end

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

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

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

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

lemma *empty-map-list-rel*:
$(\mathit{Map.empty}, []) \in \mathit{map\text{-}list\text{-}rel}$

$\langle proof \rangle$

lemma *rel-start*[*refine*]:
$((P,\ Map.empty,\ a_0),\ P',\ [],\ a_0) \in \mathit{map\text{-}set\text{-}rel} \times_r \mathit{map\text{-}list\text{-}rel} \times_r \mathit{Id}$ **if** $(P,\ P') \in \mathit{map\text{-}set\text{-}rel}$
$\langle proof \rangle$

lemma *refine-True*:
$(x1b,\ x1) \in \mathit{map\text{-}set\text{-}rel} \Longrightarrow (x1c,\ x1a) \in \mathit{map\text{-}list\text{-}rel}$
$\Longrightarrow ((x1b,\ x1c,\ True),\ x1,\ x1a,\ True) \in \mathit{map\text{-}set\text{-}rel} \times_r \mathit{map\text{-}list\text{-}rel} \times_r \mathit{Id}$
$\langle proof \rangle$

lemma *check-subsumption-ref*[*refine*]:
$V\ x2a \Longrightarrow (x1b,\ x1) \in \mathit{map\text{-}set\text{-}rel} \Longrightarrow \mathit{check\text{-}subsumption\text{-}map\text{-}set}\ x2a\ x1b = (\exists x \in x1.\ x2a \preceq x)$
$\langle proof \rangle$

lemma *check-subsumption'-ref*[*refine*]:
$set\ xs \subseteq \{x.\ V\ x\} \Longrightarrow (m,\ xs) \in \mathit{map\text{-}list\text{-}rel}$
$\Longrightarrow \mathit{check\text{-}subsumption\text{-}map\text{-}list}\ x\ m = \mathit{check\text{-}loop}\ x\ xs$
$\langle proof \rangle$

lemma *not-check-loop-non-elem*:
$x \notin set\ xs$ **if** $\neg\ \mathit{check\text{-}loop\text{-}list}\ x\ xs$
$\langle proof \rangle$

lemma *insert-ref*[*refine*]:
$(x1b,\ x1) \in \mathit{map\text{-}set\text{-}rel} \Longrightarrow$
$(x1c,\ x1a) \in \langle Id \rangle \mathit{list\text{-}set\text{-}rel} \Longrightarrow$
$\neg\ \mathit{check\text{-}loop\text{-}list}\ x2a\ x1c \Longrightarrow$
$((x1b,\ x2a\ \#\ x1c,\ False),\ x1,\ insert\ x2a\ x1a,\ False) \in \mathit{map\text{-}set\text{-}rel} \times_r \mathit{list\text{-}set\text{-}hd\text{-}rel}\ x2a \times_r \mathit{Id}$
$\langle proof \rangle$

lemma *insert-map-set-ref*:
$(m,\ S) \in \mathit{map\text{-}set\text{-}rel} \Longrightarrow (\mathit{insert\text{-}map\text{-}set}\ x\ m,\ insert\ x\ S) \in \mathit{map\text{-}set\text{-}rel}$
$\langle proof \rangle$

lemma *map-list-rel-memD*:
assumes $(m,\ xs) \in \mathit{map\text{-}list\text{-}rel}\ x \in set\ xs$
obtains xs' **where** $x \in set\ xs'\ m\ (key\ x) = Some\ xs'$
$\langle proof \rangle$

lemma *map-list-rel-memI*:

$(m, xs) \in map\text{-}list\text{-}rel \Longrightarrow m\ k = Some\ xs' \Longrightarrow x' \in set\ xs' \Longrightarrow x' \in set\ xs$
⟨proof⟩

lemma *map-list-rel-grouped-by-key*:
$x' \in set\ xs' \Longrightarrow (m, xs) \in map\text{-}list\text{-}rel \Longrightarrow m\ k = Some\ xs' \Longrightarrow key\ x' = k$
⟨proof⟩

lemma *map-list-rel-not-memI*:
$k \neq key\ x \Longrightarrow m\ k = Some\ xs' \Longrightarrow (m, xs) \in map\text{-}list\text{-}rel \Longrightarrow x \notin set\ xs'$
⟨proof⟩

lemma *map-list-rel-not-memI2*:
$x \notin set\ xs'$ **if** $m\ a = Some\ xs'$ $(m, xs) \in map\text{-}list\text{-}rel$ $x \notin set\ xs$
⟨proof⟩

lemma *push-map-list-ref*:
$x \notin set\ xs \Longrightarrow (m, xs) \in map\text{-}list\text{-}rel \Longrightarrow (push\text{-}map\text{-}list\ x\ m, x\ \#\ xs) \in map\text{-}list\text{-}rel$
⟨proof⟩

lemma *insert-map-set-ref'*[*refine*]:
$(x1b, x1) \in map\text{-}set\text{-}rel \Longrightarrow$
$(x1c, x1a) \in map\text{-}set\text{-}rel \Longrightarrow$
$\neg\ check\text{-}subsumption'\ x2a\ x1c \Longrightarrow$
$((x1b, insert\text{-}map\text{-}set\ x2a\ x1c, False), x1, insert\ x2a\ x1a, False) \in map\text{-}set\text{-}rel \times_r map\text{-}set\text{-}rel \times_r Id$
⟨proof⟩

lemma *map-list-rel-check-subsumption-map-list*:
$set\ xs \subseteq \{x.\ V\ x\} \Longrightarrow (m, xs) \in map\text{-}list\text{-}rel \Longrightarrow \neg\ check\text{-}subsumption\text{-}map\text{-}list\ x\ m \Longrightarrow x \notin set\ xs$
⟨proof⟩

lemma *push-map-list-ref'*[*refine*]:
$set\ x1a \subseteq \{x.\ V\ x\} \Longrightarrow$
$(x1b, x1) \in map\text{-}set\text{-}rel \Longrightarrow$
$(x1c, x1a) \in map\text{-}list\text{-}rel \Longrightarrow$
$\neg\ check\text{-}subsumption\text{-}map\text{-}list\ x2a\ x1c \Longrightarrow$
$((x1b, push\text{-}map\text{-}list\ x2a\ x1c, False), x1, x2a\ \#\ x1a, False) \in map\text{-}set\text{-}rel \times_r map\text{-}list\text{-}rel \times_r Id$
⟨proof⟩

lemma *sorted-wrt-tl*:
sorted-wrt R (*tl xs*) **if** *sorted-wrt R xs*
⟨*proof*⟩

lemma *irrefl-trans-on-mono*:
irrefl-trans-on R S **if** *irrefl-trans-on R S′ S* ⊆ *S′*
⟨*proof*⟩

lemma *pop-map-list-ref*[*refine*]:
(*pop-map-list v m*, *S*) ∈ *map-list-rel* **if** (*m*, *v* # *S*) ∈ *map-list-rel*
⟨*proof*⟩

lemma *tl-list-set-ref*:
(*m*, *S*) ∈ *map-set-rel* ⟹
(*st*, *ST*) ∈ *list-set-hd-rel x* ⟹
(*tl st*, *ST* − {*x*}) ∈ ⟨*Id*⟩*list-set-rel*
⟨*proof*⟩

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

lemma *dfs-map-dfs-refine′*:
dfs-map P ≤ ⇓ (*Id* $\times_r$ *map-set-rel*) (*dfs1 P′*) **if** (*P*, *P′*) ∈ *map-set-rel*
⟨*proof*⟩

lemma *dfs-map-dfs-refine*:
dfs-map P ≤ ⇓ (*Id* $\times_r$ *map-set-rel*) (*dfs P′*) **if** (*P*, *P′*) ∈ *map-set-rel V* a_0
⟨*proof*⟩

end

end

5.3 Imperative Implementation

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

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

locale *Liveness-Search-Space-Key-Impl-Defs* =
Liveness-Search-Space-Key-Defs - - - - - - *key* **for** *key* :: ′*a* ⇒ ′*k* +

fixes *A* :: *'a* ⇒ (*'ai* :: *heap*) ⇒ *assn*
fixes *succsi* :: *'ai* ⇒ *'ai list Heap*
fixes $a_0 i$ :: *'ai Heap*
fixes *Lei* :: *'ai* ⇒ *'ai* ⇒ *bool Heap*
fixes *keyi* :: *'ai* ⇒ *'ki* :: {*hashable*, *heap*} *Heap*
fixes *copyi* :: *'ai* ⇒ *'ai Heap*

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

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*}))
)

⟨*proof*⟩

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

$k = key\ v$; $(S', S) = op\text{-}map\text{-}extract\ k\ S$;
$S1' = (case\ S'\ of\ Some\ S1 \Rightarrow S1 \mid None \Rightarrow \{\})$
$in\ (\exists\ x \in S1'.\ v \trianglelefteq x)$
)

⟨*proof*⟩

lemma *check-subsumption-map-set-extract*: $(S, check\text{-}subsumption\text{-}map\text{-}set\ v\ S) = ($
let
$k = key\ v$; $(S', S) = op\text{-}map\text{-}extract\ k\ S$
$in\ ($
$case\ S'\ of$
$Some\ S1 \Rightarrow (let\ r = (\exists\ x \in S1.\ v \trianglelefteq x)\ in\ (op\text{-}map\text{-}update\ k\ S1\ S, r))$
$\mid None \Rightarrow (S, False)$
)
)

⟨*proof*⟩

lemma *check-subsumption-map-list-extract*: $(S, check\text{-}subsumption\text{-}map\text{-}list\ v\ S) = ($
let
$k = key\ v$; $(S', S) = op\text{-}map\text{-}extract\ k\ S$
$in\ ($
$case\ S'\ of$
$Some\ S1 \Rightarrow (let\ r = (\exists\ x \in set\ S1.\ x \trianglelefteq v)\ in\ (op\text{-}map\text{-}update\ k\ S1\ S, r))$
$\mid None \Rightarrow (S, False)$
)
)

⟨*proof*⟩

lemma *push-map-list-alt-def*: $((), push\text{-}map\text{-}list\ v\ S) = ($
let
$k = key\ v$; $(S', S) = op\text{-}map\text{-}extract\ k\ S$
in
$case\ S'\ of$
$Some\ S1 \Rightarrow ((), S(k \mapsto v\ \#\ S1))$
$\mid None \Rightarrow ((), S(k \mapsto [v]))$
)

⟨*proof*⟩

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* ↦ []))
)

  ⟨*proof*⟩

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* (λ*dfs* (*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*) (λ*v'* (*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_0);
    RETURN (*r*, *P*)
  })
  ⟨*proof*⟩

definition *dfs-map'* **where**
  dfs-map' a P ≡ *do* {
    (*P*,*ST*,*r*) ← *RECT* (λ*dfs* (*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′ a_0 = *dfs-map*
⟨*proof*⟩

end

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

lemma (**in** *imp-map-delete*) *hms-delete-rule* [*sep-heap-rules*]:
<hms-assn A m mi> hms-delete k mi <hms-assn A (m(k := None))>$_t$
⟨*proof*⟩

context *imp-map-delete*
begin

lemma *hms-delete-hnr*:
(*uncurry hms-delete, uncurry (RETURN oo op-map-delete)*) ∈
id-assn^k $*_a$ (*hms-assn A*)^d $\rightarrow_a$ *hms-assn A*
⟨*proof*⟩

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

end

lemma *fold-insert-set*:
fold insert xs S = *set xs* ∪ *S*
⟨*proof*⟩

lemma *set-alt-def*:
set xs = *fold insert xs* {}
⟨*proof*⟩

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

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

lemma [*def-pat-rules*]:
a_0 ≡ *UNPROTECT* a_0 (⊴) ≡ *UNPROTECT* (⊴) *succs* ≡ *UNPROTECT succs key* ≡ *UNPROTECT key*
⟨*proof*⟩

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

lemma [*intf-of-assn*]:
intf-of-assn (*hm.hms-assn′ a b*) *TYPE*((*′aa*, *′bb*) *i-map*)
⟨*proof*⟩

sepref-definition *dfs-map-impl* **is**
PR-CONST dfs-map :: $passed\text{-}assn^d \rightarrow_a$ *prod-assn bool-assn passed-assn*
⟨*proof*⟩

sepref-register *dfs-map*

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

lemma *passed-empty-refine*[*sepref-fr-rules*]:
(*uncurry0 hm.hms-empty*, *uncurry0* (*RETURN* (*PR-CONST hm.op-hms-empty*)))
∈ $unit\text{-}assn^k \rightarrow_a$ *passed-assn*
⟨*proof*⟩

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

sepref-thm *dfs-map-impl′* **is**
uncurry0 (*do* {(*r*, *p*) ← *dfs-map Map.empty*; *RETURN r*}) :: $unit\text{-}assn^k$

$\rightarrow_a$ *bool-assn*
⟨*proof*⟩

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*)
⟨*proof*⟩

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

lemma (**in** *Liveness-Search-Space-Key*) *dfs-map-empty*:
dfs-map Map.empty ≤ ⇓ (*bool-rel* $\times_r$ *map-set-rel*) *dfs-spec* **if** $V\ a_0$
⟨*proof*⟩

lemma (**in** *Liveness-Search-Space-Key*) *dfs-map-empty-correct*:
do {(*r*, *p*) ← *dfs-map Map.empty*; *RETURN r*} ≤ *SPEC* (λ *r*. *r* ⟷ (∃ *x*. a_0 →∗ *x* ∧ *x* →$^+$ *x*))
if $V\ a_0$
⟨*proof*⟩

lemma *dfs-map-impl'-hnr*:
(*uncurry0* (*dfs-map-impl' succsi* $a_0 i$ *Lei keyi copyi*),
uncurry0 (*SPEC* (λ*r*. *r* = (∃ *x*. a_0 →∗ *x* ∧ *x* →$^+$ *x*)))
) ∈ *unit-assn*$^k \rightarrow_a$ *bool-assn* **if** $V\ a_0$
⟨*proof*⟩

lemma *dfs-map-impl'-hoare-triple*:
<↑(*V* a_0)>
dfs-map-impl' succsi $a_0 i$ *Lei keyi copyi*
<λ*r*. ↑(*r* ⟷ (∃ *x*. a_0 →∗ *x* ∧ *x* →$^+$ *x*))>$_t$
⟨*proof*⟩

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*
⟨*proof*⟩

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* <λ*r*. *is-hashmap m ht* ∗ ↑ (*r* ∈ *dom m*)>
if *m* ≠ *Map.empty*
⟨*proof*⟩

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
<λ*r*. $\exists_A$*xa*. *hm.assn K V m mi* ∗ *K xa r* ∗ *true* ∗ ↑ (*RETURN xa* ≤ *next-key m*)>

⟨*proof*⟩

lemma *hm-it-next-key-next-key*:
assumes *CONSTRAINT is-pure K*
shows (*hm-it-next-key*, *next-key*) ∈ (*hm.assn K V*)^k $\rightarrow_a$ *K*
⟨*proof*⟩

lemma *hm-it-next-key-next-key′*:
(*hm-it-next-key*, *next-key*) ∈ (*hm.hms-assn V*)^k $\rightarrow_a$ *id-assn*
⟨*proof*⟩

lemma *no-fail-next-key-iff*:
nofail (*next-key m*) ⟷ *m* ≠ *Map.empty*
⟨*proof*⟩

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

private lemma *k-aux*:
assumes *k* ∈ *dom mi* (*mi*, *m*) ∈ ⟨*K*, *Id*⟩*map-rel*
shows ∃ *k′*. (*k*, *k′*) ∈ *K*
⟨*proof*⟩ **lemma** *k-aux2*:
assumes *k* ∈ *dom mi* (*k*, *k′*) ∈ *K*
shows *k′* ∈ *dom m*
⟨*proof*⟩ **lemma** *map-empty-iff*: *mi* ≠ *Map.empty* ⟷ *m* ≠ *Map.empty*
⟨*proof*⟩ **lemma** *aux*:
assumes *RETURN k* ≤ *next-key mi*
shows *RETURN* (*SOME k′*. (*k*, *k′*) ∈ *K*) ≤ *next-key m*
⟨*proof*⟩ **lemma** *aux1*:
assumes *RETURN k* ≤ *next-key mi nofail* (*next-key m*)
shows (*k*, *SOME k′*. (*k*, *k′*) ∈ *K*) ∈ *K*
⟨*proof*⟩

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

end

lemma *hm-it-next-key-next-key″*:
assumes *is-pure K*
shows (*hm-it-next-key*, *next-key*) ∈ (*hm.hms-assn′ K V*)^k $\rightarrow_a$ *K*
⟨*proof*⟩

6.2 Computing the Range of a Map

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

context
begin

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

private lemma *wf-ran-of-map-var*:
wf ran-of-map-var
⟨*proof*⟩

lemma *ran-of-map-correct*[*refine*]:
ran-of-map m ≤ *SPEC* (λ *r*. *set r* = *ran m*) **if** *finite* (*dom m*)
⟨*proof*⟩

end — End of private context for auxiliary facts and definitions

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

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

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

⟨*proof*⟩

context *imp-map-is-empty*
begin

lemma *hms-is-empty-hnr*[*sepref-fr-rules*]:
(*hms-is-empty*, *RETURN o op-map-is-empty*) $\in$ (*hms-assn A*)^k $\rightarrow_a$ *bool-assn*
⟨*proof*⟩

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

end

lemma (**in** *imp-map*) *hms-assn′-id-hms-assn*:
hms-assn′ id-assn A = *hms-assn A*
⟨*proof*⟩

lemma [*intf-of-assn*]:
intf-of-assn (*hm.hms-assn′ a b*) *TYPE*((*′aa*, *′bb*) *i-map*)
⟨*proof*⟩

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 $\rightarrow_a$ *list-assn A*
⟨*proof*⟩

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 :: $(hm.hms\text{-}assn\ A)^d \rightarrow_a list\text{-}assn\ A$
⟨*proof*⟩

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*⟩

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*⟩
including *reaches-steps-iff*
⟨*proof*⟩
including *reaches-steps-iff* ⟨*proof*⟩

end

end

locale *Leadsto-Search-Space* =
A: *Search-Space′-finite* $E\ a_0$ - $(\preceq)$ *empty*
for $E\ a_0$ *empty* **and** *subsumes* :: $'a \Rightarrow\ 'a \Rightarrow bool$ (**infix** $\preceq$ *50*)

+
fixes $P\ Q :: {'a} \Rightarrow bool$
assumes *P-mono*: $a \preceq a' \Longrightarrow \neg\ empty\ a \Longrightarrow P\ a \Longrightarrow P\ a'$
assumes *Q-mono*: $a \preceq a' \Longrightarrow \neg\ empty\ a \Longrightarrow Q\ a \Longrightarrow Q\ a'$
fixes *succs-Q* :: $'a \Rightarrow {'a}\ list$
assumes *succs-Q-correct*: $A.reachable\ a \Longrightarrow set\ (succs\text{-}Q\ a) = \{y.\ E\ a\ y \wedge Q\ y \wedge \neg\ empty\ y\}$
begin

sublocale A': *Search-Space′-finite* $E\ a_0\ \lambda$ -. *False* $(\preceq)$ *empty*
⟨*proof*⟩

sublocale B:
Liveness-Search-Space
$\lambda\ x\ y.\ E\ x\ y \wedge Q\ y \wedge \neg\ empty\ y\ a_0\ \lambda$ -. *False* $(\preceq)$ $\lambda\ x.\ A.reachable\ x \wedge \neg\ empty\ x$
succs-Q
⟨*proof*⟩

context
fixes $a_1 :: {'a}$
begin

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

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

end

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

definition

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

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

lemma
leadsto $\leq$ *leadsto-spec*
⟨*proof*⟩

definition *leadsto-spec-alt* **where**
leadsto-spec-alt =
  SPEC (λ r.
    r $\longleftrightarrow$
    ($\exists$ a. (λ x y. E x y $\wedge$ $\neg$ *empty* y)** a_0 a $\wedge$ $\neg$ *empty* a $\wedge$ P a $\wedge$ Q a $\wedge$ *reaches-cycle* a)
)

lemma *E-reaches-non-empty*:
(λ x y. E x y $\wedge$ $\neg$ *empty* y)** a b **if** a $\rightarrow*$ b *A.reachable* a $\neg$ *empty* b **for** a b
⟨*proof*⟩

lemma *leadsto-spec-leadsto-spec-alt*:
leadsto-spec $\leq$ *leadsto-spec-alt*
⟨*proof*⟩

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* :: ($'b \rightharpoonup 'a$ *set*) $\Rightarrow$ $'a$ *set* **where**
map-to-set m = ($\bigcup$ (*ran* m))

hide-const *wait*

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

context *Worklist-Map2-Defs*
begin

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

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

definition *pw-algo-map3* **where**
pw-algo-map3 = *do*
{
if F a_0 *then RETURN* (*True*, *Map.empty*)

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

end — Worklist Map 2 Defs

lemma *map-list-set-rel-empty*[*refine*, *simp*, *intro*]:
(*Map.empty*, *Map.empty*) ∈ *map-list-set-rel*
⟨*proof*⟩

lemma *map-list-set-rel-single*:
(*ml*(*key* a_0 ↦ [a_0]), *ms*(*key* a_0 ↦ {a_0})) ∈ *map-list-set-rel* **if** (*ml*, *ms*) ∈ *map-list-set-rel*
⟨*proof*⟩

context *Worklist-Map2*
begin

lemma *refine-start*[*refine*]:
(([*key* a_0 ↦ [a_0]], [a_0]), [*key* a_0 ↦ {a_0}], [a_0]) ∈ *map-list-set-rel* $\times_r$ *Id*
⟨*proof*⟩

lemma *pw-map-inv-ref*:
pw-map-inv (*x1*, *x2*, *x3*) ⟹ (*x1a*, *x1*) ∈ *map-list-set-rel* ⟹ *pw-map-inv3* (*x1a*, *x2*, *x3*)
⟨*proof*⟩

lemma *refine-aux*[*refine*]:
(*x1*, *x*) ∈ *map-list-set-rel* ⟹ ((*x1*, *x2*, *False*), *x*, *x2*, *False*) ∈ *map-list-set-rel*

$\times_r$ *Id* $\times_r$ *Id*
⟨*proof*⟩

lemma *map-list-set-relD*:
ms k = *Some* (*set xs*) **if** (*ml*, *ms*) ∈ *map-list-set-rel ml k* = *Some xs*
⟨*proof*⟩

lemma *map-list-set-rel-distinct*:
distinct xs **if** (*ml*, *ms*) ∈ *map-list-set-rel ml k* = *Some xs*
⟨*proof*⟩

lemma *map-list-set-rel-NoneD1*[*dest*, *intro*]:
ms k = *None* **if** (*ml*, *ms*) ∈ *map-list-set-rel ml k* = *None*
⟨*proof*⟩

lemma *map-list-set-rel-NoneD2*[*dest*, *intro*]:
ml k = *None* **if** (*ml*, *ms*) ∈ *map-list-set-rel ms k* = *None*
⟨*proof*⟩

lemma *map-list-set-rel-insert*:
(*ml*, *ms*) ∈ *map-list-set-rel* ⟹
ml (*key a*) = *Some xs* ⟹
ms (*key a*) = *Some* (*set xs*) ⟹
a ∉ *set xs* ⟹
(*ml*(*key a* ↦ *a* # *xs*), *ms*(*key a* ↦ *insert a* (*set xs*))) ∈ *map-list-set-rel*
⟨*proof*⟩

lemma *add-pw′-map3-ref*:
add-pw′-map3 ml xs a ≤ ⇓ (*map-list-set-rel* $\times_r$ *Id*) (*add-pw′-map2 ms xs a*)
if (*ml*, *ms*) ∈ *map-list-set-rel* ¬ *empty a*
⟨*proof*⟩

lemma *pw-algo-map3-ref*[*refine*]:
pw-algo-map3 ≤ ⇓ (*Id* $\times_r$ *map-list-set-rel*) *pw-algo-map2*
⟨*proof*⟩

lemma *pw-algo-map2-ref′*:
pw-algo-map2 ≤ ⇓ (*bool-rel* $\times_r$ *map-set-rel*) *pw-algo*
⟨*proof*⟩

lemma *pw-algo-map3-ref′*[*refine*]:
pw-algo-map3 ≤ ⇓ (*bool-rel* $\times_r$ (*map-list-set-rel O map-set-rel*)) *pw-algo*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma (**in** *Worklist-Map2-finite*) *map-set-rel-finiteI*:
finite S **if** (*m*, *S*) ∈ *map-set-rel*
⟨*proof*⟩

lemma (**in** *Worklist-Map2-finite*) *map-set-rel-finite-ranI*[*intro*]:
finite S′ **if** (*m*, *S*) ∈ *map-set-rel S′* ∈ *ran m*
⟨*proof*⟩

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

sublocale *A′*: *Worklist-Map2-finite* a_0 λ -. *False* (≼) *empty* (⊴) *E key succs1* λ -. *False*
⟨*proof*⟩

interpretation *B*:
Liveness-Search-Space-Key
λ *x y*. *E x y* ∧ *Q y* ∧ ¬ *empty y* a_0 λ -. *False* (≼) λ *x*. *A.reachable x* ∧ ¬ *empty x*
succs-Q key
⟨*proof*⟩

context
fixes a_1 :: *′a*
begin

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

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

context

assumes *A.reachable* a_1
begin

interpretation B':
Liveness-Search-Space-Key
λ *x y. E x y* $\wedge$ *Q y* $\wedge$ $\neg$ *empty y* a_1 λ -. *False* ($\preceq$) λ *x. A.reachable x* $\wedge$ $\neg$ *empty x*
succs-Q key
⟨*proof*⟩

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

end

end

definition *outer-inv* **where**
outer-inv passed done todo $\equiv$ λ (*r*, *passed'*).
(*r* $\longrightarrow$ ($\exists$ *a. A.reachable a* $\wedge$ $\neg$ *empty a* $\wedge$ *P a* $\wedge$ *Q a* $\wedge$ *reaches-cycle a*))
$\wedge$ ($\neg$ *r* $\longrightarrow$
($\forall$ *a* $\in$ $\bigcup$ *done. P a* $\wedge$ *Q a* $\longrightarrow$ $\neg$ *reaches-cycle a*)
$\wedge$ *B.liveness-compatible passed'*
$\wedge$ *passed'* $\subseteq$ {*x. A.reachable x* $\wedge$ $\neg$ *empty x*}
)

definition *inner-inv* **where**
inner-inv passed done todo $\equiv$ λ (*r*, *passed'*).
(*r* $\longrightarrow$ ($\exists$ *a. A.reachable a* $\wedge$ $\neg$ *empty a* $\wedge$ *P a* $\wedge$ *Q a* $\wedge$ *reaches-cycle a*))
$\wedge$ ($\neg$ *r* $\longrightarrow$
($\forall$ *a* $\in$ *done. P a* $\wedge$ *Q a* $\longrightarrow$ $\neg$ *reaches-cycle a*)
$\wedge$ *B.liveness-compatible passed'*
$\wedge$ *passed'* $\subseteq$ {*x. A.reachable x* $\wedge$ $\neg$ *empty x*}
)

definition *leadsto'* :: *bool nres* **where**
leadsto' = *do* {
(*r*, *passed*) $\leftarrow$ A'.*pw-algo-map2*;
let passed = *ran passed*;
(*r*, -) $\leftarrow$ *FOREACHcdi* (*outer-inv passed*) *passed* (λ(*b*,-). $\neg$*b*)

$(\lambda$ *passed'* (-,*acc*).
FOREACHcdi (*inner-inv acc*) *passed'* (λ(*b*,-). $\neg$*b*)
(λ*v'* (-,*passed*).
do {
ASSERT(*A.reachable v'* $\wedge$ $\neg$ *empty v'*);
if P v' $\wedge$ *Q v' then has-cycle v' passed else RETURN* (*False*, *passed*)
}
)
(*False*, *acc*)
)
(*False*, {});
RETURN r
}

lemma *leadsto'-correct*:
leadsto' $\leq$ *leadsto-spec*
⟨*proof*⟩

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

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'*)
⟨*proof*⟩

definition *leadsto-map3'* :: *bool nres* **where**
leadsto-map3' = *do* {
(*r*, *passed*) $\leftarrow$ *A'.pw-algo-map2*;
let passed = *ran passed*;
(*r*, -) $\leftarrow$ *FOREACHcd passed* (λ(*b*,-). $\neg$*b*)
(λ *passed'* (-,*acc*).
do {
passed' $\leftarrow$ *SPEC* (λ*l*. *set l* = *passed'*);
nfoldli passed' (λ(*b*,-). $\neg$*b*)
(λ*v'* (-,*passed*).
if P v' $\wedge$ *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*⟩

lemma *leadsto-map3′-ref*[*refine*]:
leadsto-map3′ ≤ ⇓ *Id leadsto′*
⟨*proof*⟩

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*)
(λ*v′* (-,*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* $\times_r$ *map-list-set-rel*
⟨*proof*⟩

lemma *map-list-set-rel-ran-set-rel*:
(*ran ml*, *ran ms*) ∈ ⟨*br set* (λ-. *True*)⟩*set-rel* **if** (*ml*, *ms*) ∈ *map-list-set-rel*
⟨*proof*⟩

lemma *Id-list-rel-ref*:

$(x'a, x'a) \in \langle Id \rangle list\text{-}rel$
⟨*proof*⟩

lemma *map-list-set-rel-finite-ran*:
finite (*ran ml*) **if** (*ml*, *ms*) ∈ *map-list-set-rel*
⟨*proof*⟩

lemma *leadsto-map3-ref*[*refine*]:
leadsto-map3 ≤ ⇓ *Id leadsto′*
⟨*proof*⟩

definition *leadsto-map4* :: *bool nres* **where**
```
leadsto-map4 = do {
  (r, passed) ← A′.pw-algo-map3;
  ASSERT (finite (dom passed));
  passed ← ran-of-map passed;
  (r, -) ← nfoldli passed (λ(b,-). ¬b)
    (λ passed′ (-,acc).
        nfoldli passed′ (λ(b,-). ¬b)
          (λv′ (-,passed).
          if P v′ ∧ Q v′ then has-cycle-map v′ passed else RETURN (False,
passed)
          )
          (False, acc)
    )
    (False, Map.empty);
  RETURN r
}
```

lemma *ran-of-map-ref*:
ran-of-map m ≤ *SPEC* (λ*c*. (*c*, *ran m′*) ∈ *br set* (λ -. *True*)) **if** *finite* (*dom m*) (*m*, *m′*) ∈ *Id*
⟨*proof*⟩

lemma *aux-ref*:
$(xa, x'a) \in Id \Longrightarrow$
$x'a = (x1b, x2b) \Longrightarrow xa = (x1c, x2c) \Longrightarrow (x1c, x1b) \in bool\text{-}rel$
⟨*proof*⟩

definition *foo* = *A′.pw-algo-map3*

lemma [*refine*]:
A′.pw-algo-map3 ≤ ⇓ (*br id* (λ (-, *m*). *finite* (*dom m*))) *foo*
⟨*proof*⟩

lemma *leadsto-map4-ref* [*refine*]:
  leadsto-map4 $\leq \Downarrow$ *Id leadsto-map3*
  ⟨*proof*⟩

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* (λ*l*. *set l* = *passed′*);
          nfoldli passed′ (λ(*b*,-). ¬*b*)
            (λ*v′* (-,*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′ $\leq \Downarrow$ *Id leadsto-map3′*
  ⟨*proof*⟩

lemma *leadsto-map4′-correct*:
  leadsto-map4′ ≤ *leadsto-spec-alt*
⟨*proof*⟩

end

end

7.3 Imperative Implementation

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

definition
list-of-set $S = SPEC\ (\lambda l.\ set\ l = S)$

lemma *lso-id-hnr*:
$(return\ o\ id,\ list\text{-}of\text{-}set) \in (lso\text{-}assn\ A)^d \rightarrow_a list\text{-}assn\ A$
⟨*proof*⟩

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

context *Worklist-Map2-Impl*
begin

sepref-thm *pw-algo-map2-impl* **is**
uncurry0 (*pw-algo-map2*) ::
$unit\text{-}assn^k \rightarrow_a bool\text{-}assn \times_a (hm.hms\text{-}assn'\ K\ (lso\text{-}assn\ A))$
⟨*proof*⟩

end

locale *Leadsto-Search-Space-Key-Impl* =
Leadsto-Search-Space-Key a_0 *F - empty - E key F′ P Q succs-Q succs1* +
liveness: *Liveness-Search-Space-Key-Impl* a_0 *F - V succs-Q* $\lambda\ x\ y.\ E\ x\ y \wedge \neg\ empty\ y \wedge Q\ y$
- *key A succsi* $a_0 i$ *Lei keyi copyi*
for *key* :: $'v \Rightarrow\ 'k$
and a_0 *F F′ copyi P Q V empty succs-Q succs1 E A succsi* $a_0 i$ *Lei keyi* +
fixes *succs1i* **and** *emptyi* **and** *Pi Qi* **and** *tracei*
assumes *succs1-impl*: $(succs1i,\ (RETURN \circ\circ PR\text{-}CONST)\ succs1) \in A^k \rightarrow_a list\text{-}assn\ A$
and *empty-impl*:
$(emptyi, RETURN\ o\ PR\text{-}CONST\ empty) \in A^k \rightarrow_a bool\text{-}assn$
assumes [*sepref-fr-rules*]:
$(Pi, RETURN\ o\ PR\text{-}CONST\ P) \in A^k \rightarrow_a bool\text{-}assn\ (Qi, RETURN\ o\ PR\text{-}CONST\ Q) \in A^k \rightarrow_a bool\text{-}assn$
assumes *trace-impl*:
$(uncurry\ tracei, uncurry\ (\lambda(- :: string)\ \text{-}.\ RETURN\ ())) \in id\text{-}assn^k *_a A^k \rightarrow_a id\text{-}assn$
begin

sublocale *Worklist-Map2-Impl* - - λ -. *False* - *succs1* - - λ-. *False* - *succs1i* -
λ-. *return False Lei*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

sepref-register *has-cycle-map-copy*

lemma *has-cycle-map-fold*:
has-cycle-map = *liveness.dfs-map′*
⟨*proof*⟩

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′ :: $unit\text{-}assn^k \rightarrow_a bool\text{-}assn$
⟨*proof*⟩

concrete-definition (**in** −) *leadsto-impl*
uses *Leadsto-Search-Space-Key-Impl.leadsto-impl.refine-raw* **is** (*uncurry0*
?f,-)∈-

lemma *leadsto-impl-hnr*:
(*uncurry0* (
leadsto-impl copyi succsi a_0i *Lei keyi succs1i emptyi Pi Qi tracei*
),
uncurry0 leadsto-spec-alt
) ∈ $unit\text{-}assn^k \rightarrow_a bool\text{-}assn$ **if** $V\ a_0$
⟨*proof*⟩

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.