

Worklist Algorithms

Simon Wimmer, Peter Lammich

February 6, 2026

Abstract

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

Contents

1	Preliminaries	3
1.1	Search Spaces	3
1.2	Miscellaneous	8
2	Subsumption Graphs	10
2.1	Preliminaries	10
2.2	Definitions	11
2.3	Reachability	15
2.4	Liveness	19
2.5	Appendix	19
2.6	Old Material	22
3	Unified Passed-Waiting-List	24
3.1	Utilities	24
3.2	Generalized Worklist Algorithm	25
3.3	Towards an Implementation of the Unified Passed-Waiting List	30
3.4	Heap Hash Map	38
3.5	Imperative Implementation	43

4	Generic Worklist Algorithm With Subsumption	48
4.1	Utilities	48
4.2	Standard Worklist Algorithm	49
4.3	From Multisets to Lists	55
4.4	Towards an Implementation	57
4.5	Implementation on Lists	62
5	Checking Always Properties	64
5.1	Abstract Implementation	64
5.2	Implementation on Maps	70
5.3	Imperative Implementation	74
6	A Next-Key Operation for Hashmaps	81
6.1	Definition and Key Properties	81
6.2	Computing the Range of a Map	83
7	Checking Leads-To Properties	85
7.1	Abstract Implementation	85
7.2	Implementation on Maps	87
7.3	Imperative Implementation	96

1 Preliminaries

theory *Worklist-Locales*

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

1.1 Search Spaces

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

locale *Search-Space-Defs* =

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

and $a_0 :: 'a$ — Start state

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

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

begin

sublocale *Graph-Start-Defs* E a_0 $\langle \text{proof} \rangle$

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

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

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

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

end

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

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

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

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

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

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

assumes $\text{refl}[\text{intro!}, \text{simp}]: a \preceq a$

and $\text{trans}[\text{trans}]: a \preceq b \implies b \preceq c \implies a \preceq c$

assumes *mono*:

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

and *F-mono*: $a \preceq a' \implies F \ a \implies F \ a'$

begin

sublocale *preorder* (\preceq) (\prec)
 ⟨*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*]: $a \preceq b \implies b \preceq c \implies a \preceq c$

assumes *mono*:

$a \preceq b \implies E a a' \implies V a \implies V b \implies \neg \text{empty } a \implies \exists b'. V b' \wedge E b b' \wedge a' \preceq b'$

and *empty-subsumes*: $\text{empty } a \implies a \preceq a'$

and *empty-mono*: $\neg \text{empty } a \implies a \preceq b \implies \neg \text{empty } b$

and *empty-E*: $V x \implies \text{empty } x \implies E x x' \implies \text{empty } x'$

and *F-mono*: $a \preceq a' \implies F a \implies F a'$

begin

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

sublocale *search-space*:

Search-Space-Nodes $\lambda x y. E x y \wedge \neg \text{empty } y a_0 F (\preceq) \lambda v. V v \wedge \neg \text{empty } v$

⟨*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*]: $a \preceq b \implies b \preceq c \implies a \preceq c$

assumes *mono*:

$a \preceq b \implies E a a' \implies \text{reachable } a \implies \text{reachable } b \implies \neg \text{empty } a \implies \exists b'. E b b' \wedge a' \preceq b'$

and *empty-subsumes*: $\text{empty } a \implies a \preceq a'$

and *empty-mono*: $\neg \text{empty } a \implies a \preceq b \implies \neg \text{empty } b$

and *empty-E*: $\text{reachable } x \implies \text{empty } x \implies E x x' \implies \text{empty } x'$

and *F-mono*: $a \preceq a' \implies F a \implies F a'$

begin

sublocale *preorder* (\preceq) (\prec)
 $\langle proof \rangle$

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

end

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

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

sublocale *Search-Space-finite-strict* \subseteq *Search-Space-finite*
 $\langle proof \rangle$

locale *Search-Space'* = *Search-Space* +
 assumes *final-non-empty*: $F a \implies \neg$ *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* $\implies a \preceq b \longleftrightarrow a \trianglelefteq b$

locale *Search-Space''-start* = *Search-Space''-pre* +
 assumes *start-non-empty* [*simp*]: \neg *empty a*₀

locale *Search-Space''* = *Search-Space''-pre* + *Search-Space'*

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

sublocale *Search-Space''-finite* \subseteq *Search-Space'-finite* $\langle proof \rangle$

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 $subsumes-key[intro, simp]: a \trianglelefteq b \implies key\ a = key\ b$

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

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

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 $Worklist-Map2-finite \subseteq Search-Space''-finite\ \langle proof \rangle$

locale $Worklist4-Impl-Defs = Worklist3-Defs +$
fixes $A :: 'a \Rightarrow 'ai \Rightarrow assn$

fixes *sucsci* :: 'ai ⇒ 'ai list Heap
fixes *a₀i* :: 'ai Heap
fixes *Fi* :: 'ai ⇒ bool Heap
fixes *Lei* :: 'ai ⇒ 'ai ⇒ bool Heap
fixes *emptyi* :: 'ai ⇒ bool Heap

locale *Worklist4-Impl* = *Worklist4-Impl-Defs* + *Worklist4* +
— This is the easy variant: Operations cannot depend on additional heap.
assumes [*sepref-fr-rules*]: (*uncurry0 a₀i*, *uncurry0 (RETURN (PR-CONST a₀))*) ∈ *unit-assn^k →_a A*
assumes [*sepref-fr-rules*]: (*Fi*, *RETURN o PR-CONST F'*) ∈ *A^k →_a bool-assn*
assumes [*sepref-fr-rules*]: (*uncurry Lei*, *uncurry (RETURN oo PR-CONST (≤))*) ∈ *A^k *_a A^k →_a bool-assn*
assumes [*sepref-fr-rules*]: (*sucsci*, *RETURN o PR-CONST succs*) ∈ *A^k →_a list-assn A*
assumes [*sepref-fr-rules*]: (*emptyi*, *RETURN o PR-CONST empty*) ∈ *A^k →_a bool-assn*

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

sublocale *Worklist4-Impl-finite-strict* ⊆ *Search-Space''-finite-strict* ⟨*proof*⟩

locale *Worklist-Map2-Impl-Defs* =
Worklist4-Impl-Defs - - - - - *A* + *Worklist-Map2-Defs a₀* - - - - - *key*
for *A* :: 'a ⇒ 'ai :: {heap} ⇒ - **and** *key* :: 'a ⇒ 'k +
fixes *keyi* :: 'ai ⇒ 'ki :: {hashable, heap} Heap
fixes *copyi* :: 'ai ⇒ 'ai Heap
fixes *tracei* :: string ⇒ 'ai ⇒ unit Heap

end

theory *Worklist-Common*

imports *Worklist-Locales*

begin

lemma *list-ex-foldli*:

list-ex P xs = foldli xs Not (λ x y. P x ∨ y) False
⟨*proof*⟩

lemma (in *Search-Space-finite*) *finitely-branching*:

assumes *reachable a*
shows *finite* ({*a'*. *E a a' ∧ ¬ empty a'*})
⟨*proof*⟩

definition (in *Search-Space-Key-Defs*)

```
map-set-rel =  
  {(m, s).  
    $\bigcup (\text{ran } m) = s \wedge (\forall k. \forall x. m\ k = \text{Some } x \longrightarrow (\forall v \in x. \text{key } v = k))$   
  }  
^  
  {finite (dom m) ^ (\forall k S. m k = Some S \longrightarrow finite S)  
  }
```

end

1.2 Miscellaneous

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

lemma *mset-eq-empty-iff*:

```
 $M = \{\#\} \longleftrightarrow \text{set-mset } M = \{\}$   
<proof>
```

lemma *filter-mset-eq-empty-iff*:

```
 $\{\#x \in \# A. P\ x\#\} = \{\#\} \longleftrightarrow (\forall x \in \text{set-mset } A. \neg P\ x)$   
<proof>
```

lemma *mset-remove-member*:

```
 $x \in \# A - B \text{ if } x \in \# A \wedge x \notin \# B$   
<proof>
```

end

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

```
datatype message = ExploredState
```

```
definition write-msg :: message  $\Rightarrow$  unit where  
  write-msg m = ()
```

```
code-printing code-module Tracing  $\rightarrow$  (SML)
```

```
<  
structure Tracing : sig  
  val count-up : unit  $\rightarrow$  unit  
  val get-count : unit  $\rightarrow$  int  
end = struct
```

```

    val counter = Unsynchronized.ref 0;
    fun count-up () = (counter := !counter + 1);
    fun get-count () = !counter;
end
> and (OCaml)
<
module Tracing : sig
  val count-up : unit -> 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-assn*^k \rightarrow_a *unit-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-assn*^k \rightarrow_a *unit-assn*
<proof>

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

end

2 Subsumption Graphs

theory *Worklist-Algorithms-Subsumption-Graphs*

imports

Timed-Automata.Graphs

Timed-Automata.More-List1

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

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

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 $P Q :: 'a \Rightarrow \text{bool}$

assumes *Q-finite*: $\text{finite } \{x. Q x\}$ **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* :: $'a \Rightarrow 'a \Rightarrow \text{bool}$ (**infix** $\langle \preceq \rangle$ 50) **and** *less* (**infix** $\langle \prec \rangle$ 50) +

fixes $E :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ — The full edge set

begin

sublocale *Graph-Defs* E *<proof>*

end

locale *Subsumption-Graph-Pre-Nodes-Defs* = *Subsumption-Graph-Pre-Defs*

+

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

begin

sublocale *Subgraph-Node-Defs-Notation* *<proof>*

end

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

begin

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

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

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

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

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

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

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

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

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

end

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

assumes *mono*:

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

begin

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

end

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

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

end

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

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

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

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

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

begin

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

end

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

ord less-eq less **for** *less-eq* :: 'a ⇒ 'a ⇒ bool (**infix** <pre> 50) **and** *less* (**infix**
<prec> 50) +

fixes *E* :: 'a ⇒ 'a ⇒ bool — The full edge set

and *s₀* :: 'a — Start state

fixes *RE* :: 'a ⇒ 'a ⇒ bool — Subgraph of the graph given by the full
edge set

fixes *SE* :: 'a ⇒ 'a ⇒ bool — Subsumption edges

and *covered* :: 'a ⇒ bool

fixes *closure* :: 'a ⇒ 'b

fixes *P* :: 'a ⇒ bool

fixes *Q* :: 'a ⇒ bool

begin

sublocale *Graph-Start-Defs* *E s₀* $\langle proof \rangle$

sublocale *G*: *Graph-Start-Defs* *RE s₀* $\langle proof \rangle$

end

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

Subsumption-Graph-Closure-View-Defs +

preorder less-eq less +

assumes *mono*:

closure a \prec *closure b* $\implies E a a' \implies P a \implies P b \implies \exists b'. E b b' \wedge$
closure a' \preceq *closure b'*

assumes *closure-eq*:

closure a = *closure b* $\implies E a a' \implies P a \implies P b \implies \exists b'. E b b' \wedge$
closure a' = *closure b'*

assumes *reachability-compatible*:

$\forall s. Q s \implies$ (if *covered s* then $(\exists t. SE s t \wedge G.reachable t)$ else $(\forall s'. E s s' \implies RE s s')$)

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

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

assumes *finite-closure*: *finite* (*closure* 'UNIV)

assumes *P-post*: $a \rightarrow b \implies P b$

assumes *P-pre*: $a \rightarrow b \implies P a$

assumes $P\text{-}s_0$: $P\ s_0$
assumes $Q\text{-}post$: $RE\ a\ b \implies Q\ b$
assumes $Q\text{-}s_0$: $Q\ s_0$
begin

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

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

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

end

locale *Reachability-Compatible-Subsumption-Graph-Final* = *Reachability-Compatible-Subsumption-Graph*
 $+$
fixes $F :: 'a \Rightarrow bool$ — Final states
assumes $F\text{-}mono[intro]$: $F\ a \implies a \preceq b \implies F\ b$

locale *Liveness-Compatible-Subsumption-Graph* = *Reachability-Compatible-Subsumption-Graph-Final*
 $+$
assumes *no-subsumption-cycle*:
 $G'.reachable\ x \implies x \rightarrow_G^{+'}\ x \implies x \rightarrow_G^+\ x$

2.3 Reachability

context *Subsumption-Graph-Defs*
begin

Setup for automation

context
includes *graph-automation*
begin

lemma $G'\text{-}reachable\text{-}G\text{-}reachable[intro]$:
 $G'.reachable\ a$ **if** $G'.reachable\ a$
 $\langle proof \rangle$

lemma $G\text{-}reachable\text{-}G'\text{-}reachable[intro]$:
 $G'.reachable\ a$ **if** $G'.reachable\ a$

$\langle proof \rangle$

lemma *G-G'-reachable-iff*:

$G.reachable\ a \longleftrightarrow G'.reachable\ a$

$\langle proof \rangle$

end

end

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

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

lemma *G'-finite-reachable*: *finite* $\{a.\ G'.reachable\ a\}$

$\langle proof \rangle$

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** $G.reachable\ s$
 $\langle proof \rangle$

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
 $\langle proof \rangle$

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** $\nexists s'.\ G.reachable\ s' \wedge F\ s'$
 $\langle proof \rangle$

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** $G.\text{reachable}\ a$

<proof>

lemma *G-steps-sound*[*intro*]:

steps xs **if** $G.\text{steps}\ xs$

<proof>

lemma *G-run-sound*[*intro*]:

run xs **if** $G.\text{run}\ xs$

<proof>

lemma *G'-reachability-sound*[*intro*]:

reachable a **if** $G'.\text{reachable}\ a$

<proof>

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

<proof>

lemma *G-steps-G'-steps*[*intro*]:

$G'.\text{steps}\ as$ **if** $G.\text{steps}\ as$

<proof>

lemma *reachable-has-surrogate*:

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

<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** $G.\text{reachable}\ s$

<proof>

lemma *subsumption-step*:

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

$\langle proof \rangle$

lemma *subsumption-step'*:

$\exists b'. b \preceq b' \wedge a' \rightarrow_{G^{+'}} b'$ **if** *reachable* $a \rightarrow b$ $G'.reachable\ a'$ $a \preceq a'$
 $\langle proof \rangle$

theorem *reachability-complete'*:

$\exists s'. s \preceq s' \wedge G'.reachable\ s'$ **if** $a \rightarrow^* s$ $G'.reachable\ a$
 $\langle proof \rangle$

corollary *reachability-complete*:

$\exists s'. s \preceq s' \wedge G'.reachable\ s'$ **if** *reachable* s
 $\langle proof \rangle$

corollary *reachability-correct*:

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

lemma *steps-G'-steps*:

$\exists ys\ ns.$ *list-all2* $(\preceq)\ xs\ (nth\ ys\ ns) \wedge G'.steps\ (b \# ys)$ **if**
steps $(a \# xs)$ *reachable* $a \preceq b$ $G'.reachable\ b$
 $\langle proof \rangle$

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

assumes *steps* $(s_0 \# ws @ x \# xs @ [x])$
shows $\exists x' xs' ys'. x \preceq x' \wedge G'.steps\ (s_0 \# xs' @ x' \# ys' @ [x'])$
 $\langle proof \rangle$

lemma *cycle-G'-cycle'*:

assumes *steps* $(s_0 \# ws @ x \# xs @ [x])$
shows $\exists y ys. x \preceq y \wedge G'.steps\ (y \# ys @ [y]) \wedge G'.reachable\ y$
 $\langle proof \rangle$

lemma *cycle-G'-cycle*:

assumes *reachable* $x \rightarrow^+ x$
shows $\exists y ys. x \preceq y \wedge G'.reachable\ y \wedge y \rightarrow_{G^{+'}} y$
 $\langle proof \rangle$

corollary *G'-reachability-complete*:

$\exists s'. s \preceq s' \wedge G'.reachable\ s'$ **if** $G'.reachable\ s$
 $\langle proof \rangle$

end

end

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

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

2.4 Liveness

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

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

2.5 Appendix

context *Subsumption-Graph-Pre-Nodes*

begin

Setup for automation

context

includes *graph-automation*

begin

lemma *steps-mono*:

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

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

<proof> **including** *subgraph-automation*
<proof>

lemma *steps-append-subsumption*:

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

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

<proof>

lemma *steps-replicate-subsumption*:

assumes $x \preceq \text{last } (x \# xs) \ G'.\text{steps } (x \# xs) \ n > 0 \ V x$

notes *[intro] = preorder-intros*

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

<proof>

context

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

begin

lemma *wf-less-on-reachable-set*:

assumes *antisym*: $\bigwedge x y. x \preceq y \implies y \preceq x \implies x = y$

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 $\exists ys. \text{steps } (y \# ys) \wedge \text{list-all2 } (\preceq) xs ys$
 $\langle \text{proof} \rangle$

lemma *steps-append-subsumption:*

assumes $\text{steps } (x \# xs) \text{ steps } (y \# ys) y \preceq \text{last } (x \# xs) \text{ reachable } x$
 $\text{reachable } y$

shows $\exists ys'. \text{steps } (x \# xs @ ys') \wedge \text{list-all2 } (\preceq) ys ys'$
 $\langle \text{proof} \rangle$

lemma *steps-replicate-subsumption:*

assumes $x \preceq \text{last } (x \# xs) \text{ steps } (x \# xs) n > 0 \text{ reachable } x$

notes $[\text{intro}] = \text{preorder-intros}$

shows $\exists ys. \text{steps } (x \# ys) \wedge \text{list-all2 } (\preceq) (\text{concat } (\text{replicate } n xs)) ys$
 $\langle \text{proof} \rangle$

context

assumes $\text{finite-reachable: finite } \{x. \text{reachable } x\}$

begin

lemma *wf-less-on-reachable-set:*

assumes $\text{antisym: } \bigwedge x y. x \preceq y \implies y \preceq x \implies x = y$

shows $\text{wf } \{(x, y). y \prec x \wedge \text{reachable } x \wedge \text{reachable } y\}$ (**is** $\text{wf } ?S$)
 $\langle \text{proof} \rangle$

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

lemma *pre-cycle-cycle':*

assumes $A: x \preceq x' \text{ steps } (x \# xs @ [x']) \text{ reachable } x$

shows $\exists x'' ys. x' \preceq x'' \wedge \text{steps } (x'' \# ys @ [x'']) \wedge \text{reachable } x''$
 $\langle \text{proof} \rangle$

lemma *pre-cycle-cycle:*

$(\exists x x'. \text{reachable } x \wedge \text{reaches } x x' \wedge x \preceq x') \longleftrightarrow (\exists x. \text{reachable } x \wedge \text{reaches } x x)$

including *reaches-steps-iff* $\langle \text{proof} \rangle$

end

end

end

context *Subsumption-Graph-Defs*

begin

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

lemma G'' -reachable- G' [*intro*]:
 $G'.reachable\ x$ **if** $G''.reachable\ x$
 $\langle proof \rangle$

end

locale *Reachability-Compatible-Subsumption-Graph-Total* = *Reachability-Compatible-Subsumption-G*
+

assumes *total*: $reachable\ a \implies reachable\ b \implies a \preceq b \vee b \preceq a$

begin

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

end

2.6 Old Material

locale *Reachability-Compatible-Subsumption-Graph'* = *Subsumption-Graph-Defs*
+ *order* $(\preceq)\ (\prec)$ +

assumes *reachability-compatible*:

$\forall s. G.reachable\ s \implies (\forall s'. E\ s\ s' \implies RE\ s\ s') \vee (\exists t. s \prec t \wedge G.reachable\ t)$

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

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

assumes *mono*:

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

begin

Setup for automation

context

includes *graph-automation*

notes [*intro*] = *order.trans*

begin

lemma *subgraph'*[*intro*]:
 $E\ s\ s'$ **if** $RE\ s\ s'$

$\langle \text{proof} \rangle$

lemma *G-reachability-sound*[intro]:

reachable a **if** *G.reachable a*

$\langle \text{proof} \rangle$

lemma *G-steps-sound*[intro]:

steps xs **if** *G.steps xs*

$\langle \text{proof} \rangle$

lemma *G-run-sound*[intro]:

run xs **if** *G.run xs*

$\langle \text{proof} \rangle$

lemma *reachable-has-surrogate*:

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

$\langle \text{proof} \rangle$

lemma *subsumption-step*:

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

reachable a *E a b* *G.reachable a'* $a \preceq a'$

$\langle \text{proof} \rangle$

theorem *reachability-complete'*:

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

$\langle \text{proof} \rangle$

corollary *reachability-complete*:

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

$\langle \text{proof} \rangle$

corollary *reachability-correct*:

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

$\langle \text{proof} \rangle$

lemma *G'-reachability-sound*[intro]:

reachable a **if** *G'.reachable a*

$\langle \text{proof} \rangle$

corollary *G'-reachability-complete*:

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

$\langle \text{proof} \rangle$

end

end

end

3 Unified Passed-Waiting-List

theory *Unified-PW*

imports *Refine-Imperative-HOL.Sepref Worklist-Common Worklist-Algorithms-Subsumption-Graph*

begin

hide-const *wait*

3.1 Utilities

definition *take-from-set* **where**

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

lemma *take-from-set-correct*:

assumes $s \neq \{\}$

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

<proof>

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

definition *take-from-mset* **where**

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

lemma *take-from-mset-correct*:

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

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

<proof>

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

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

<proof>

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

3.2 Generalized Worklist Algorithm

context *Search-Space-Defs-Empty*

begin

definition *reachable-subsumed* $S = \{x' \mid x \ x'. \text{ reachable } x' \wedge \neg \text{ empty } x' \wedge x' \preceq x \wedge x \in S\}$

definition

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

definition *pw-inv-frontier passed wait* =

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

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

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

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

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

if $\exists a'. E a \ a' \wedge F a'$ *then*
brk
else

$$\begin{aligned}
& \neg brk \wedge set\text{-}mset\ wait' \subseteq set\text{-}mset\ wait \cup \{a' . E\ a\ a'\} \wedge \\
& (\forall s \in set\text{-}mset\ wait. \exists s' \in set\text{-}mset\ wait'. s \preceq s') \wedge \\
& (\forall s \in \{a' . E\ a\ a' \wedge \neg empty\ a'\}. \exists s' \in set\text{-}mset\ wait' \cup passed. s \\
\preceq s') \wedge \\
& (\forall s \in passed \cup \{a\}. \exists s' \in passed'. s \preceq s') \wedge \\
& (passed' \subseteq passed \cup \{a\} \cup \{a' . E\ a\ a' \wedge \neg empty\ a'\} \wedge \\
& ((\exists x \in passed'. \neg (\exists x' \in passed. x \preceq x')) \vee wait' \subseteq\# wait \wedge passed \\
= passed') \\
&) \\
&)
\end{aligned}$$

definition

$$\begin{aligned}
& init\text{-}pw\text{-}spec \equiv \\
& SPEC\ (\lambda (passed, wait). \\
& \quad if\ empty\ a_0\ then\ passed = \{\} \wedge wait \subseteq\# \{\#a_0\# \} \ else\ passed \subseteq \{a_0\} \\
& \wedge wait = \{\#a_0\#\})
\end{aligned}$$

abbreviation *subsumed-elem* :: 'a ⇒ 'a set ⇒ bool

where *subsumed-elem* a M ≡ ∃ a'. a' ∈ M ∧ a ≤ a'

notation

subsumed-elem (⟨(-/ ∈'' -)⟩ [51, 51] 50)

definition *pw-inv-frontier'* passed wait =

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

lemma *pw-inv-frontier-frontier'*:

pw-inv-frontier' passed wait **if**

pw-inv-frontier passed wait passed ⊆ Collect reachable

⟨proof⟩

lemma

pw-inv-frontier passed wait **if** *pw-inv-frontier'* passed wait

⟨proof⟩

definition *pw-algo* where

$$\begin{aligned}
& pw\text{-}algo = do \\
& \{ \\
& \quad if\ F\ a_0\ then\ RETURN\ (True, \{\}) \\
& \quad else\ if\ empty\ a_0\ then\ RETURN\ (False, \{\}) \\
& \quad else\ do\ { \\
& \quad \quad (passed, wait) ← init\text{-}pw\text{-}spec;
\end{aligned}$$

```

      (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
  ∀ s ∈ passed. ∃ x ∈ passed'. s ≼ x

```

$\forall s \in \#wait - \{\#a\# \}. \text{Multiset.Bex } wait' ((\preceq) s)$
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*
 $\neg b \in' \text{set-mset } wait$
 $E b b'$
 $\neg \text{empty } b \neg \text{empty } b'$
 $b \in' \text{passed}$
 $\text{reachable } b \text{ passed} \subseteq \{a. \text{reachable } a \wedge \neg \text{empty } a\}$
shows $b' \in' \text{passed} \cup \text{set-mset } wait$

⟨proof⟩ **lemma** *pw-inv-frontier-empty-elem*:

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

shows *pw-inv-frontier passed (wait - \{\#a\# \})*

⟨proof⟩ **lemma** *aux3*:

assumes
 $\text{set-mset } wait \subseteq \text{Collect } \text{reachable}$
 $a \in \# \text{ wait}$
 $\forall s \in \text{set-mset } (wait - \{\#a\# \}). \exists s' \in \text{set-mset } wait'. s \preceq s'$
 $\forall s \in \{a'. E a a' \wedge \neg \text{empty } a'\}. \exists s' \in \text{passed} \cup \text{set-mset } wait'. s \preceq s'$
 $\forall s \in \text{passed} \cup \{a\}. \exists s' \in \text{passed}'. s \preceq s'$
 $\text{passed}' \subseteq \text{passed} \cup \{a\} \cup \{a'. E a a' \wedge \neg \text{empty } a\}$
 pw-inv-frontier passed wait
 $\text{passed} \subseteq \{a. \text{reachable } a \wedge \neg \text{empty } a\}$
shows *pw-inv-frontier passed' wait'*

⟨proof⟩ **lemma** *aux6*:

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

shows *start-subsumed (insert a passed) wait'*

⟨proof⟩

lemma *empty-E-star*:

*empty x' if E** x x' reachable x empty x*

⟨proof⟩

lemma aux4:
assumes $pw\text{-inv}\text{-frontier}$ $passed \ \{\#\}$ $reachable \ x$ $start\text{-subsumed}$ $passed \ \{\#\}$
 $passed \subseteq \{a. \text{reachable } a \wedge \neg \text{empty } a\} \neg \text{empty } x$
shows $\exists x' \in passed. x \preceq x'$
 $\langle proof \rangle$

lemmas $[intro] = reachable\text{-step}$

private lemma aux7:

assumes
 $a \in \# \ wait$
 $set\text{-mset } wait \subseteq Collect \ reachable$
 $set\text{-mset } wait' \subseteq set\text{-mset } (wait - \{\#a\}) \cup Collect \ (E \ a)$
 $x \in \# \ wait'$

shows $reachable \ x$

$\langle proof \rangle$ **lemma aux8:**

$x \in reachable\text{-subsumed } S' \ \mathbf{if} \ x \in reachable\text{-subsumed } S \ \forall s \in S. \exists x \in S'.$

$s \preceq x$

$\langle proof \rangle$ **lemma aux9:**

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

shows $False$

$\langle proof \rangle$ **lemma aux10:**

assumes $\forall a \in passed' \cup set\text{-mset } wait. \neg F \ a \ F \ x \ x \in \# \ wait - \{\#a\}$

shows $False$

$\langle proof \rangle$

lemma aux12:

$size \ wait' < size \ wait \ \mathbf{if} \ wait' \subseteq \# \ wait - \{\#a\} \ a \in \# \ wait$

$\langle proof \rangle$

lemma aux13:

assumes
 $passed \subseteq \{a. \text{reachable } a \wedge \neg \text{empty } a\}$
 $passed' \subseteq insert \ a \ (passed \cup \{a'. E \ a \ a' \wedge \neg \text{empty } a'\})$
 $\neg \text{empty } a$
 $reachable \ a$
 $\forall s \in passed. \exists x \in passed'. s \preceq x$
 $a'' \in passed'$
 $\forall x \in passed. \neg a'' \preceq x$

shows
 $passed' \subseteq \{a. \text{reachable } a \wedge \neg \text{empty } a\} \wedge \text{reachable-subsumed } passed \subseteq \text{reachable-subsumed } passed'$
 $\vee passed' = passed \wedge \text{size } wait'' < \text{size } wait$
 $\langle \text{proof} \rangle$

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

end — Context

end — Search Space

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

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* $passed \ wait \ a \equiv SPEC (\lambda (passed', wait', brk).$
 $\text{if } \exists x \in \text{set } (succs \ a). F \ x \ \text{then } brk$

else $passed' \subseteq passed \cup \{x \in set (succs\ a). \neg (\exists y \in passed. x \preceq y)\}$
 $\wedge passed \subseteq passed'$
 $\wedge wait \subseteq\# wait'$
 $\wedge wait' \subseteq\# wait + mset ([x \leftarrow succs\ a. \neg (\exists y \in passed. x \preceq y)])$
 $\wedge (\forall x \in set (succs\ a). \exists y \in passed'. x \preceq y)$
 $\wedge (\forall x \in set (succs\ a). \neg (\exists y \in passed. x \preceq y) \longrightarrow (\exists y \in\# wait'. x \preceq y))$
 $\wedge \neg brk$

definition *add-pw passed wait a* \equiv
 $nfoldli (succs\ a) (\lambda(-, -, brk). \neg brk)$
 $(\lambda a (passed, wait, brk). RETURN ($
 $\textit{if } F\ a\ \textit{then}$
 $(passed, wait, True)$
 $\textit{else if } \exists x \in passed. a \preceq x\ \textit{then}$
 $(passed, wait, False)$
 $\textit{else (insert } a\ passed, add-mset\ a\ wait, False)$
 $)$
 $(passed, wait, False)$

end — Worklist1 Defs

context *Worklist1*
begin

lemma *add-pw-unified-spec-ref:*
 $add-pw-unified-spec\ passed\ wait\ a \leq add-pw-spec\ passed\ wait\ a$
if *reachable a a* $\in passed$
 $\langle proof \rangle$

lemma *add-pw-ref:*
 $add-pw\ passed\ wait\ a \leq \Downarrow Id (add-pw-unified-spec\ passed\ wait\ a)$
 $\langle proof \rangle$

end — Worklist 1

context *Worklist2-Defs*
begin

definition *add-pw' passed wait a* \equiv
 $nfoldli (succs\ a) (\lambda(-, -, brk). \neg brk)$
 $(\lambda a (passed, wait, brk). RETURN ($

```

    if F a then
      (passed, wait, True)
    else if empty a then
      (passed, wait, False)
    else if  $\exists x \in \text{passed}. a \sqsubseteq x$  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 a0 then RETURN (True, {})
    else if empty a0 then RETURN (False, {})
    else do {
      (passed, wait) ← RETURN ({a0}, {#a0#});
      (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*:

$\text{empty } x \vee x \sqsubseteq y \longleftrightarrow x \preceq y$
 ⟨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 $a \leq \Downarrow (Id \cap \{((p, w, -), -). p \neq \{\} \wedge \text{set-mset } w \subseteq p\})$ (*add-pw* passed wait a)

if $\text{passed} \neq \{\}$ *set-mset* wait \subseteq *passed*
 ⟨proof⟩

lemma *add-pw'-ref1* [*refine*]:

add-pw' passed wait a

$\leq \Downarrow (Id \cap \{((p, w, -), -). p \neq \{\} \wedge \text{set-mset } w \subseteq p\})$ (*add-pw-spec* passed' wait' a')

if $\text{passed} \neq \{\}$ *set-mset* wait \subseteq *passed* *reachable* $a \in$ *passed*

and [*simp*]: $\text{passed} = \text{passed}'$ wait = wait' $a = a'$

⟨proof⟩

lemma *refine-weaken*:

$p \leq \Downarrow R$ $p' \text{ if } p \leq \Downarrow S$ $p' S \subseteq R$

⟨proof⟩

lemma *add-pw'-ref*:

add-pw' passed wait $a \leq$

$\Downarrow (\{((p, w, b), (p', w', b')). p \neq \{\} \wedge p = p' \cup \text{set-mset } w \wedge w = w' \wedge b = b'\})$

(*add-pw-spec* passed' wait' a')

if $\text{passed} \neq \{\}$ *set-mset* wait \subseteq *passed* *reachable* $a \in$ *passed*

and [*simp*]: $\text{passed} = \text{passed}'$ wait = wait' $a = a'$

⟨proof⟩

lemma

$((\{a_0\}, \{\#a_0\#}, \text{False}), \{\}, \{\#a_0\#}, \text{False})$

$\in \{((p, w, b), (p', w', b')). p = p' \cup \text{set-mset } w' \wedge w = w' \wedge b = b'\}$

⟨proof⟩

lemma [*refine*]:

RETURN $(\{a_0\}, \{\#a_0\\}) \leq \Downarrow (Id \cap \{((p, w), (p', w')). p \neq \{\} \wedge \text{set-mset } w \subseteq p\})$ *init-pw-spec*

if $\neg \text{empty } a_0$

⟨proof⟩

lemma [*refine*]:

take-from-mset wait \leq

$\Downarrow \{((x, \text{wait}), (y, \text{wait}')). x = y \wedge \text{wait} = \text{wait}' \wedge \text{set-mset } \text{wait} \subseteq \text{passed}$

```

 $\wedge x \in \text{passed}\}$ 
  (take-from-mset wait')
  if wait = wait' set-mset wait  $\subseteq$  passed wait  $\neq \{\#\}$ 
  <proof>

```

```

lemma pw-algo-unified-ref:
  pw-algo-unified  $\leq \Downarrow$  Id pw-algo
  <proof>

```

end — Worklist 2

```

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

```

```

lemma take-from-list-correct:
  assumes s  $\neq$  []
  shows take-from-list s  $\leq$  SPEC ( $\lambda (x, s'). s = x \# s'$ )
  <proof>

```

```

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

```

```

context Worklist-Map-Defs
begin

```

```

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

```

$(passed, wait, False)$

definition

$pw\text{-map}\text{-inv} \equiv \lambda (passed, wait, brk).$

$\exists passed' wait'.$

$(passed, passed') \in map\text{-set}\text{-rel} \wedge (wait, wait') \in list\text{-mset}\text{-rel} \wedge$

$pw\text{-inv} (passed', wait', brk)$

definition $pw\text{-algo}\text{-map}$ **where**

$pw\text{-algo}\text{-map} = do$

{

if $F a_0$ then RETURN ($True, Map.empty$)

else if empty a_0 then RETURN ($False, Map.empty$)

else do {

$(passed, wait) \leftarrow RETURN ([key\ a_0 \mapsto \{a_0\}], [a_0]);$

$(passed, wait, brk) \leftarrow WHILEIT\ pw\text{-map}\text{-inv}\ (\lambda (passed, wait, brk).$

$\neg brk \wedge wait \neq [])$

$(\lambda (passed, wait, brk). do$

{

$(a, wait) \leftarrow take\text{-from}\text{-list}\ wait;$

ASSERT ($reachable\ a$);

if empty a then RETURN ($passed, wait, brk$) else $add\text{-pw}'\text{-map}$

$passed\ wait\ a$

}

)

$(passed, wait, False);$

RETURN ($brk, passed$)

}

}

end — Worklist Map Defs

lemma $ran\text{-upd}\text{-cases}$:

$(x \in ran\ m) \vee (x = y)$ **if** $x \in ran\ (m(a \mapsto y))$

$\langle proof \rangle$

lemma $ran\text{-upd}\text{-cases}2$:

$(\exists k. m\ k = Some\ x \wedge k \neq a) \vee (x = y)$ **if** $x \in ran\ (m(a \mapsto y))$

$\langle proof \rangle$

context $Worklist\text{-Map}$

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*
 \langle *proof* \rangle

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

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*
 \langle *proof* \rangle

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

end — Worklist Map

context *Worklist-Map2-Defs*
begin

definition

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

```

    }
  )
  (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 ≤ ↓ Id (*add-pw'-map* passed' wait' a')

if (passed, passed') ∈ Id (wait, wait') ∈ Id (a, a') ∈ Id

⟨proof⟩

lemma *pw-algo-map2-ref[refine]*:

pw-algo-map2 ≤ ↓ Id *pw-algo-map*

⟨proof⟩

end — Worklist Map 2

lemma (in *Worklist-Map2-finite*) *pw-algo-map2-correct*:

$pw\text{-algo-map2} \leq SPEC (\lambda (brk, passed).$
 $(brk \longleftrightarrow F\text{-reachable}) \wedge$
 $(\neg brk \longrightarrow$
 $(\exists p.$
 $(passed, p) \in map\text{-set-rel} \wedge (\forall a. reachable\ a \wedge \neg empty\ a \longrightarrow (\exists b \in p.$
 $a \preceq b))$
 $\wedge p \subseteq \{a. reachable\ a \wedge \neg empty\ a\})$
 $)$
 $)$
<proof>

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* ($\langle \cdot := \cdot \rangle$ 62)

definition *big-star* :: *assn multiset* \Rightarrow *assn* ($\langle \wedge^* \cdot \rangle$ [60] 90) **where**

big-star $S \equiv fold\text{-mset} (*) emp\ S$

interpretation *comp-fun-commute-mult*:

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

<proof>

lemma *sep-big-star-insert* [*simp*]: $\wedge^* (add\text{-mset}\ x\ S) = (x * \wedge^* S)$

<proof>

lemma *sep-big-star-union* [*simp*]: $\wedge^* (S + T) = (\wedge^* S) * (\wedge^* T)$

<proof>

lemma *sep-big-star-empty* [*simp*]: $\wedge^* \{\#\} = emp$

<proof>

lemma *big-star-entatilst-mono*:

$\bigwedge^* T \Longrightarrow_t \bigwedge^* S$ **if** $S \subseteq_{\#} T$
 ⟨proof⟩

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

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** $k \in S$ *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]:
 $z \in_{\#} mset-set\ S \longleftrightarrow z \in S$ **if** *finite* S
 ⟨proof⟩

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

lemma *map-assn-update-aux*:
 $map-assn\ A\ m\ mh * A\ v\ vi \Longrightarrow_A map-assn\ A\ (m(k \mapsto v))\ (mh(k \mapsto vi))$
if $k \notin dom\ m$
 ⟨proof⟩

lemma *map-assn-update*:

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

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

definition (*in imp-map*) *hms-assn'* $K\ A = hr\text{-}comp\ (hms\text{-}assn\ A)\ (\langle the\text{-}pure\ K, Id \rangle map\text{-}rel)$

declare (*in imp-map*) *hms-assn'-def* [*symmetric, fcomp-norm-unfold*]

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

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

$\langle emp \rangle hms\text{-}empty\ \langle hms\text{-}assn\ A\ Map.empty \rangle_t$
<proof>

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

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

$\langle hms\text{-}assn\ A\ m\ mi\ * \ A\ v\ vi \rangle hms\text{-}update\ k\ vi\ mi\ \langle hms\text{-}assn\ A\ (m(k \mapsto v)) \rangle_t$
<proof>

lemma *restrict-not-in-dom-simp* [*simp*]:

$m \mid' (- \{k\}) = m \ \mathbf{if}\ m\ k = None$
<proof>

definition [*code*]:

hms-extract lookup delete $k\ m =$
do {
 $vo \leftarrow lookup\ k\ m;$
 case vo of
 $None \Rightarrow return\ (None, m) \mid$
 $Some\ v \Rightarrow do \{$
 $m \leftarrow delete\ k\ m;$
 $return\ (Some\ v, m)$
 }
}

definition [*code*]:

hms-lookup lookup copy $k\ m =$
do {

```

    vo ← lookup k m;
  case vo of
    None ⇒ return None |
    Some v ⇒ do {
      v' ← copy v;
      return (Some v')
    }
}

```

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

lemma *map-assn-domain-simps*[*simp*]:
assumes *vassn-tag* (*map-assn* *A* *m* *mh*)
shows $mh\ k = None \longleftrightarrow m\ k = None \text{ dom } mh = \text{dom } m \text{ finite } (\text{dom } m)$
<proof>

lemma *hms-extract-rule* [*sep-heap-rules*]:
<hms-assn A m mi>
hms-extract lookup delete k mi
 $\langle \lambda (vi, mi'). \text{option-assn } A (m\ k)\ vi * \text{hms-assn } A (m(k := None))\ mi' \rangle_t$
<proof>

lemma *hms-lookup-rule* [*sep-heap-rules*]:
assumes
 $(\text{copy}, \text{RETURN } o\ \text{COPY}) \in A^k \rightarrow_a A$
shows
<hms-assn A m mi>
hms-lookup lookup copy k mi
 $\langle \lambda vi. \text{hms-assn } A\ m\ mi * \text{option-assn } A (m\ k)\ vi \rangle_t$
<proof>

end

context *imp-map-update*
begin

lemma *hms-update-hnr*:
 $(\text{uncurry2 } \text{hms-update}, \text{uncurry2 } (\text{RETURN } ooo\ \text{op-map-update})) \in$
 $\text{id-assn}^k *_a A^d *_a (\text{hms-assn } A)^d \rightarrow_a \text{hms-assn } A$
<proof>

sempref-decl-impl *update*: *hms-update-hnr* **uses** *op-map-update.fref*[**where**

$V = Id]$ $\langle proof \rangle$

end

context *imp-map-empty*

begin

lemma *hms-empty-hnr*:

$(uncurry0\ hms\text{-}empty, uncurry0\ (RETURN\ op\text{-}map\text{-}empty)) \in unit\text{-}assn^k$
 $\rightarrow_a\ hms\text{-}assn\ A$
 $\langle proof \rangle$

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

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

sublocale *hms: map-custom-empty op-hms-empty*

$\langle proof \rangle$

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*:

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

context *imp-map-extract-derived*

begin

lemma *hms-extract-hnr*:

$(uncurry\ (hms\text{-}extract\ lookup\ delete), uncurry\ (RETURN\ oo\ op\text{-}map\text{-}extract))$
 \in
 $id\text{-}assn^k *_{a}\ (hms\text{-}assn\ A)^d \rightarrow_a\ prod\text{-}assn\ (option\text{-}assn\ A)\ (hms\text{-}assn\ A)$
 $\langle proof \rangle$

lemma *hms-lookup-hnr*:

$(uncurry\ (hms\text{-}lookup\ lookup\ copy), uncurry\ (RETURN\ oo\ op\text{-}map\text{-}lookup))$

```

∈
  id-assnk *a (hms-assn A)k →a option-assn A if (copy, RETURN o COPY)
∈ Ak →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: λ (x # xs) ⇒ (x, xs) :: [λl. l≠[]]f ⟨A⟩list-rel → A
  ×r ⟨A⟩list-rel
  ⟨proof⟩
end

```

```

context Worklist-Map2-Defs
begin

```

```

definition trace where
  trace ≡ λtype a. RETURN ()

```

```

definition
  explored-string = "Explored"

```

```

definition
  final-string = "Final"

```

```

definition
  added-string = "Add"

```

definition

subsumed-string = "Subsumed"

definition

empty-string = "Empty"

lemma *add-pw'-map2-alt-def:*

```

add-pw'-map2 passed wait a = do {
  trace explored-string a;
  nfoldli (succs a) ( $\lambda(-, -, brk). \neg brk$ )
  ( $\lambda a$  (passed, wait, -).
    do {
      RETURN (
        if empty a then
          (passed, wait, False)
        else if F' a then (passed, wait, True)
        else
          let
            k = key a;
            (v, passed) = op-map-extract k passed
          in
            case v of
              None  $\Rightarrow$  (passed(k  $\mapsto$  {COPY a}), a # wait, False) |
              Some passed'  $\Rightarrow$ 
                if  $\exists x \in$  passed'. a  $\leq$  x then
                  (passed(k  $\mapsto$  passed'), wait, False)
                else
                  (passed(k  $\mapsto$  (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) ( $\lambda(-, -, brk). \neg brk$ )
  ( $\lambda a$  (passed, wait, -).
    do {
      if empty a then
        do {trace empty-string a; RETURN (passed, wait, False)}
    }
  )
}

```

```

    else if F' a then do {trace final-string a; RETURN (passed, wait,
True)}
    else
      let
        k = key a;
        (v, passed) = op-map-extract k passed
      in
        case v of
          None => do {trace added-string a; RETURN (passed(k ↦ {COPY
a}), a # wait, False)} |
          Some passed' =>
            if ∃ x ∈ passed'. a ⊆ x then
              do {trace subsumed-string a; RETURN (passed(k ↦ passed'),
wait, False)}
            else do {
              trace added-string a;
              RETURN (passed(k ↦ (insert (COPY a) passed')), a #
wait, False)
            }
        }
    )
  (passed, wait, False)
}
⟨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*) ∈ $A^k \rightarrow_a K$

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

assumes [*sepref-fr-rules*]: (*uncurry tracei*, *uncurry trace*) ∈ $id-assign^k *_a A^k \rightarrow_a id-assign$

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*]:
 $a_0 \equiv \text{UNPROTECT } a_0 \quad F' \equiv \text{UNPROTECT } F' \quad (\trianglelefteq) \equiv \text{UNPROTECT}$
 $(\trianglelefteq) \text{ succs} \equiv \text{UNPROTECT succs}$
 $\text{empty} \equiv \text{UNPROTECT empty} \quad \text{keyi} \equiv \text{UNPROTECT keyi} \quad F \equiv \text{UNPROTECT } F \quad \text{key} \equiv \text{UNPROTECT key}$
 $\langle \text{proof} \rangle$

lemma *take-from-list-alt-def*:
 $\text{take-from-list } xs = \text{do } \{- \leftarrow \text{ASSERT } (xs \neq []); \text{RETURN } (\text{hd-tl } xs)\}$
 $\langle \text{proof} \rangle$

lemma [*safe-constraint-rules*]: $\text{CN-FALSE is-pure } A \implies \text{is-pure } A \langle \text{proof} \rangle$

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

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

lemma [*sepref-import-param*]:
 $(\text{explored-string}, \text{explored-string}) \in \text{Id}$
 $(\text{subsumed-string}, \text{subsumed-string}) \in \text{Id}$
 $(\text{added-string}, \text{added-string}) \in \text{Id}$
 $(\text{final-string}, \text{final-string}) \in \text{Id}$
 $(\text{empty-string}, \text{empty-string}) \in \text{Id}$
 $\langle \text{proof} \rangle$

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*
 $\text{uncurry0 } (\text{do } \{(r, p) \leftarrow \text{pw-algo-map2}; \text{RETURN } r\}) :: \text{unit-assn}^k \rightarrow_a$
 bool-assn
 $\langle \text{proof} \rangle$

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

end — *Worklist Map2 Impl*

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

begin

lemma *pw-algo-map2-correct'*:

$(do \{(r, p) \leftarrow pw\text{-algo-map2}; RETURN r\}) \leq SPEC (\lambda brk. brk = F\text{-reachable})$
 $\langle proof \rangle$

lemma *pw-impl-hnr-F-reachable*:

$(uncurry0 (pw\text{-impl keyi copyi tracei Lei } a_0i Fi succsi emptyi), uncurry0$
 $(RETURN F\text{-reachable}))$
 $\in unit\text{-assn}^k \rightarrow_a bool\text{-assn}$
 $\langle proof \rangle$

end

locale *Worklist-Map2-Hashable* =

Worklist-Map2-Impl-finite

begin

sepref-decl-op *F-reachable* :: *bool-rel* $\langle proof \rangle$

lemma [*def-pat-rules*]: *F-reachable* $\equiv op\text{-F-reachable}$ $\langle proof \rangle$

lemma *hnr-op-F-reachable*:

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

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

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

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

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

assumes [*sepref-fr-rules*]: $(keyi, RETURN o PR\text{-CONST } key) \in A^k \rightarrow_a$
 K

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

shows

$(uncurry0 (pw\text{-impl keyi copyi tracei Lei } a_0i Fi succsi emptyi),$
 $uncurry0 (RETURN (PR\text{-CONST } op\text{-F-reachable})))$

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

$\langle proof \rangle$

sepref-decl-impl *hnr-op-F-reachable* $\langle proof \rangle$

end — Worklist Map 2

end — End of Theory

4 Generic Worklist Algorithm With Subsumption

theory *Worklist-Subsumption-Multiset*

imports

Refine-Imperative-HOL.Sepref

Worklist-Algorithms-Misc

Worklist-Locales

Unified-PW — only for shared definitions

begin

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

4.1 Utilities

definition *take-from-set where*

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

lemma *take-from-set-correct:*

assumes $s \neq \{\}$

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

<proof>

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

definition *take-from-mset where*

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

lemma *take-from-mset-correct:*

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

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

<proof>

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

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

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

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

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

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

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

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

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*)
 }
)
 (*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

\langle proof \rangle

context

begin

private lemma *aux1*:

assumes $\forall x \in \text{passed}. \neg a \preceq x$
and $\text{passed} \subseteq \text{Collect reachable}$
and $\text{reachable } a$

shows

$((\text{insert } a \text{ passed}, \text{wait}', \text{brk}'),$
 $\text{passed}, \text{wait}, \text{brk})$
 $\in \text{worklist-var}$

\langle proof \rangle **lemma** *aux2*:

assumes

$a' \in \text{passed}$
 $a \preceq a'$
 $a \in \# \text{wait}$
 $\text{worklist-inv-frontier passed wait}$

shows $\text{worklist-inv-frontier passed } (\text{wait} - \{\#a\# \})$

\langle proof \rangle **lemma** *aux5*:

assumes

$a' \in \text{passed}$
 $a \preceq a'$
 $a \in \# \text{wait}$
 $\text{worklist-start-subsumed passed wait}$

shows $\text{worklist-start-subsumed passed } (\text{wait} - \{\#a\# \})$

\langle proof \rangle **lemma** *aux3*:

assumes

$\text{set-mset wait} \subseteq \text{Collect reachable}$
 $a \in \# \text{wait}$
 $\forall s \in \text{set-mset } (\text{wait} - \{\#a\# \}) \cup \{a'. E a a' \wedge \neg \text{empty } a'\}. \exists s' \in$
 $\text{set-mset wait}'. s \preceq s'$
 $\text{worklist-inv-frontier passed wait}$

shows $\text{worklist-inv-frontier } (\text{insert } a \text{ passed}) \text{wait}'$

\langle proof \rangle **lemma** *aux6*:

assumes

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

shows $\text{worklist-start-subsumed } (\text{insert } a \text{ passed}) \text{wait}'$

\langle proof \rangle

lemma *aux4*:

assumes *worklist-inv-frontier passed* $\{\#\}$ *reachable* *x* *worklist-start-subsumed*
passed $\{\#\}$

$passed \subseteq Collect\ reachable$

shows $\exists x' \in passed. x \preceq x'$

$\langle proof \rangle$

theorem *worklist-algo-correct*:

$worklist-algo \leq SPEC (\lambda brk. brk \longleftrightarrow F\text{-reachable})$

$\langle proof \rangle$

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

end — Context

end — Search Space

context *Search-Space''-Defs*

begin

definition *worklist-inv-frontier'* *passed* *wait* =

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

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

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

$worklist\ inv\ (passed, wait, brk) \wedge (\forall a \in passed. \neg empty\ a) \wedge (\forall a \in set\ mset\ wait. \neg empty\ a)$

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

$($
if $\exists a'. E\ a\ a' \wedge F\ a'$ *then*

brk

else

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

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

wait'. $s \preceq s'$)

$) \wedge (\forall s \in set\ mset\ wait'. \neg empty\ s)$

$)$

definition *worklist-algo'* **where**
worklist-algo' = do
{
 if $F a_0$ then RETURN True
 else do {
 let passed = {};
 let wait = {# a_0 #};
 (*passed*, *wait*, *brk*) \leftarrow WHILEIT *worklist-inv'* (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*. $a \leq a'$) 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 — Search Space” Defs

context *Search-Space''-start*

begin

lemma *worklist-algo-list-inv-ref*[*refine*]:

fixes $x x'$

assumes

$\neg F a_0 \neg F a_0$

$(x, x') \in \{((\textit{passed}, \textit{wait}, \textit{brk}), (\textit{passed}', \textit{wait}', \textit{brk}'))\}$.

$\textit{passed} = \textit{passed}' \wedge \textit{wait} = \textit{wait}' \wedge \textit{brk} = \textit{brk}' \wedge (\forall a \in \textit{passed}. \neg$
empty a)

$\wedge (\forall a \in \textit{set-mset wait}. \neg \textit{empty a})\}$

worklist-inv x'

shows *worklist-inv'* x
 ⟨*proof*⟩

lemma [*refine*]:
 $take\text{-}from\text{-}mset\ wait \leq$
 $\Downarrow \{(x, wait), (y, wait')\}. x = y \wedge wait = wait' \wedge \neg empty\ x \wedge (\forall a \in$
 $set\text{-}mset\ wait. \neg empty\ a)\}$
 ($take\text{-}from\text{-}mset\ wait'$)
if $wait = wait' \forall a \in set\text{-}mset\ wait. \neg empty\ a\ wait \neq \{\#\}$
 ⟨*proof*⟩

lemma [*refine*]:
 $add\text{-}succ\text{-}spec' wait\ x \leq$
 $\Downarrow (\{(wait, wait'). wait = wait' \wedge (\forall a \in set\text{-}mset\ wait. \neg empty\ a)\} \times_r$
 $bool\text{-}rel)$
 ($add\text{-}succ\text{-}spec\ wait'\ x'$)
if $wait = wait'\ x = x' \forall a \in set\text{-}mset\ wait. \neg empty\ a$
 ⟨*proof*⟩

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

end — Search Space” Start

context *Search-Space''-Defs*
begin

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

end — Search Space” Defs

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

lemma *worklist-algo''-correct*:
 $worklist\text{-}algo'' \leq SPEC (\lambda brk. brk \longleftrightarrow F\text{-reachable})$
 ⟨*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-from-list $s = \text{ASSERT } (s \neq []) \gg \text{SPEC } (\lambda (x, s'). s = x \# s')$

lemma *take-from-list-correct*:

assumes $s \neq []$

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

<proof>

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

context *Search-Space-Defs-Empty*

begin

definition *worklist-inv-frontier-list passed wait* =

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

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

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

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

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

$(\neg \text{brk} \longrightarrow$

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

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

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

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

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

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

$($

```

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

```

end — Search Space Empty Defs

context *Search-Space''-Defs*

begin

definition *worklist-algo-list* **where**

worklist-algo-list = do

{

if $F a_0$ then RETURN True

else do {

let *passed* = {};

let *wait* = [a_0];

$(passed, wait, brk) \leftarrow WHILEIT\ worklist-inv-list\ (\lambda (passed, wait,$
 $brk). \neg brk \wedge wait \neq [])$

$(\lambda (passed, wait, brk). do$

{

$(a, wait) \leftarrow take-from-list\ wait;$

ASSERT (*reachable* a);

if $(\exists a' \in passed. a \preceq a')$ then RETURN $(passed, wait, brk)$

else

do

{

$(wait, brk) \leftarrow add-succ-spec-list\ wait\ a;$

let *passed* = insert a *passed*;

RETURN $(passed, wait, brk)$

}

}

)

$(passed, wait, False);$

RETURN *brk*

}

}

end — Search Space'' Defs

context *Search-Space''-pre*

begin

lemma *worklist-algo-list-inv-ref*:

fixes $x x'$

assumes

$\neg F a_0 \neg F a_0$

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

worklist-inv' x'

shows *worklist-inv-list* x

$\langle proof \rangle$

lemma *take-from-list-take-from-mset-ref*[*refine*]:

take-from-list $xs \leq \Downarrow \{((x, xs), (y, m)). x = y \wedge mset\ xs = m\}$ (*take-from-mset*
 m)

if $mset\ xs = m$

$\langle proof \rangle$

lemma *add-succ-spec-list-add-succ-spec-ref*[*refine*]:

add-succ-spec-list $xs\ b \leq \Downarrow \{((xs, b), (m, b')). mset\ xs = m \wedge b = b'\}$
(*add-succ-spec'* $m\ b'$)

if $mset\ xs = m\ b = b'$

$\langle proof \rangle$

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

$\langle proof \rangle$

end — Search Space”

4.4 Towards an Implementation

context *Worklist1-Defs*

begin

definition

add-succ1 $wait\ a \equiv$

nfoldli (*succs* a) ($\lambda(-, brk). \neg brk$)

($\lambda a (wait, brk).$

do {

ASSERT ($\forall x \in set\ wait. \neg empty\ x$);

if $F\ a$ *then RETURN* ($wait, True$) *else RETURN* (

if ($\exists x \in set\ wait. a \preceq x \wedge \neg x \preceq a$) $\vee empty\ a$

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) ( $\lambda(-, brk). \neg brk$ )
  ( $\lambda a (wait, brk).$ 
    if F a then RETURN (wait, True) else RETURN (
      if empty a
      then wait
      else if  $\exists x \in set\ wait. a \sqsubseteq x \wedge \neg x \sqsubseteq a$ 
      then  $[x \leftarrow wait. \neg x \sqsubseteq a]$ 
      else  $a \# [x \leftarrow wait. \neg x \sqsubseteq a], False$ 
    )
  )
  (wait, False)

```

end

context *Worklist1*

begin

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')  $\in Id$  (a, a')  $\in b\ rel$  Id reachable  $\forall x \in set\ wait'. \neg empty\ x$ 
   $\langle proof \rangle$ 

```

end

context *Worklist2*

begin

lemma *add-succ1'-ref[refine]*:

```

add-succ1' wait a  $\leq \Downarrow (Id \times_r\ bool\ rel)$  (add-succ1 wait' a')
if (wait, wait')  $\in Id$  (a, a')  $\in b\ rel$  Id reachable  $\forall x \in set\ wait'. \neg empty\ x$ 
   $\langle proof \rangle$ 

```

lemma *add-succ1'-ref'[refine]*:

$add_succ1' \text{ wait } a \leq \Downarrow (Id \times_r \text{ bool-rel}) (add_succ_spec_list \text{ wait}' a')$
if $(\text{wait}, \text{wait}') \in Id \ (a, a') \in b\text{-rel } Id \text{ reachable } \forall x \in \text{set } \text{wait}' . \neg \text{empty } x$
 $\langle \text{proof} \rangle$

definition *worklist-algo1'* **where**

$worklist_algo1' = do$
 $\{$
 $\text{if } F \ a_0 \text{ then RETURN True}$
 $\text{else do } \{$
 $\text{let } passed = \{\};$
 $\text{let } wait = [a_0];$
 $(passed, wait, brk) \leftarrow WHILEIT \ worklist_inv_list \ (\lambda (passed, wait,$
 $brk). \neg brk \wedge wait \neq [])$
 $(\lambda (passed, wait, brk). do$
 $\{$
 $(a, wait) \leftarrow \text{take-from-list } wait;$
 $\text{if } (\exists a' \in passed. a \sqsubseteq a') \text{ then RETURN } (passed, wait, brk)$
 else
 do
 $\{$
 $(wait, brk) \leftarrow add_succ1' \text{ wait } a;$
 $\text{let } passed = \text{insert } a \text{ passed};$
 $\text{RETURN } (passed, wait, brk)$
 $\}$
 $\}$
 $\}$
 $\}$
 $(passed, wait, False);$
 $\text{RETURN } brk$
 $\}$
 $\}$

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

$take_from_list \text{ wait} \leq$
 $\Downarrow \{((x, wait), (y, wait')). x = y \wedge wait = wait' \wedge \neg \text{empty } x \wedge (\forall a \in$
 $\text{set } wait. \neg \text{empty } a)\}$
 $(take_from_list \text{ wait}')$
if $wait = wait' \ \forall a \in \text{set } wait. \neg \text{empty } a \ \text{wait} \neq []$
 $\langle \text{proof} \rangle$

lemma *worklist-algo1-list-ref[refine]*: $worklist_algo1' \leq \Downarrow Id \ worklist_algo_list$

$\langle \text{proof} \rangle$

definition *worklist-algo1* **where**

worklist-algo1 \equiv if empty a_0 then RETURN False else *worklist-algo1'*

lemma *worklist-algo1-ref[refine]*: *worklist-algo1* \leq \Downarrow Id *worklist-algo''*
 ⟨proof⟩

end — Worklist2

context *Worklist3-Defs*

begin

definition

add-succ2 wait a \equiv
nfoldli (*succs a*) ($\lambda(-,brk). \neg brk$)
 ($\lambda a (wait,brk).$
 if empty a then RETURN (*wait*, False)
 else if $F' a$ then RETURN (*wait*, True)
 else RETURN (
 if $\exists x \in set\ wait. a \sqsubseteq x \wedge \neg x \sqsubseteq a$
 then [$x \leftarrow wait. \neg x \sqsubseteq a$]
 else $a \# [x \leftarrow wait. \neg x \sqsubseteq a], False$)
)
 (*wait*, False)

definition

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

end

context *Worklist3*

begin

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

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

$\langle \text{proof} \rangle$

lemma *add-succ2-alt-def*:

```
add-succ2 wait a  $\equiv$ 
  nfoldli (succs a) ( $\lambda(-,brk). \neg brk$ )
  ( $\lambda a (wait,brk).$ 
    if empty a then RETURN (wait, False)
    else if F' a then RETURN (wait, True)
    else RETURN (filter-insert-wait wait a, False)
  )
  (wait, False)
 $\langle \text{proof} \rangle$ 
```

lemma *add-succ2-ref[refine]*:

```
add-succ2 wait a  $\leq \Downarrow (Id \times_r \text{bool-rel}) (add-succ1' wait' a')$ 
if (wait, wait')  $\in Id$  (a, a')  $\in Id$ 
 $\langle \text{proof} \rangle$ 
```

definition *worklist-algo2' where*

```
worklist-algo2' = do
  {
    if F' a0 then RETURN True
    else do {
      let passed = {};
      let wait = [a0];
      (passed, wait, brk)  $\leftarrow$  WHILEIT worklist-inv-list ( $\lambda (passed, wait,$ 
brk).  $\neg brk \wedge wait \neq []$ )
      ( $\lambda (passed, wait, brk).$  do
        {
          (a, wait)  $\leftarrow$  take-from-list wait;
          if ( $\exists a' \in passed. a \sqsubseteq a'$ ) then RETURN (passed, wait, brk)
        }
      )
    }
  )
  (passed, wait, False);
  RETURN brk
}
```

}

lemma *worklist-algo2'-ref[refine]*: *worklist-algo2' ≤ ↓Id worklist-algo1' if*
 \neg *empty a₀*
 ⟨*proof*⟩

definition *worklist-algo2* **where**
worklist-algo2 \equiv *if empty a₀ 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*:

$[x \leftarrow xs. P x] = \text{rev} (\text{foldli } xs (\lambda x. \text{True}) (\lambda x xs. \text{if } P x \text{ then } x \# xs \text{ else } xs) [])$

(**is** - = $\text{rev} (\text{foldli } xs ?c ?f [])$)

⟨*proof*⟩

context notes [*split!*] = *list.split* **begin**

sempref-decl-op *list-hdtl*: $\lambda (x \# xs) \Rightarrow (x, xs) :: [\lambda l. l \neq []]_f \langle A \rangle \text{list-rel} \rightarrow A$
 $\times_r \langle A \rangle \text{list-rel}$

⟨*proof*⟩

end

context *Worklist4-Impl*

begin

sempref-register *PR-CONST a₀ PR-CONST F' PR-CONST (≤) PR-CONST*
succs PR-CONST empty

lemma [*def-pat-rules*]:

$a_0 \equiv \text{UNPROTECT } a_0 \text{ } F' \equiv \text{UNPROTECT } F' (\leq) \equiv \text{UNPROTECT}$

$(\leq) \text{ succs} \equiv \text{UNPROTECT } \text{succs}$

$\text{empty} \equiv \text{UNPROTECT } \text{empty}$

⟨*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 →_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*
→_a bool-assn
⟨proof⟩

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

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

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

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

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

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

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

shows

($\text{uncurry0 } (\text{worklist-algo2-impl } Lei \ a_0i \ Fi \ sucsci \ emptyi), \text{uncurry0 } (\text{RETURN } (\text{PR-CONST } \text{op-F-reachable})))$

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

<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 ($\langle \cdot \rightarrow \cdot \rangle [100, 100] 40$)

notation $G.E'$ ($\langle \cdot \rightarrow \cdot \rangle [100, 100] 40$)

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

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

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

notation $G.G'.reaches1$ ($\langle \cdot \rightarrow^+ \cdot \rangle [100, 100] 40$)

Plain set membership is also an option.

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

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

```

dfs  $P \equiv \text{do}$  {
   $(P, ST, r) \leftarrow \text{RECT } (\lambda \text{dfs } (P, ST, v).$ 
     $\text{do}$  {
      if check-loop  $v$   $ST$  then  $\text{RETURN } (P, ST, \text{True})$ 
      else do {
        if  $\exists v' \in P. v \preceq v'$  then
           $\text{RETURN } (P, ST, \text{False})$ 
        else do {
           $\text{let } ST = v \# ST;$ 
           $(P, ST', r) \leftarrow$ 
             $\text{FOREACHcd } \{v'. v \rightarrow v'\} (\lambda(-, -, b). \neg b) (\lambda v' (P, ST, -). \text{dfs}$ 
 $(P, ST, v')) (P, ST, \text{False});$ 
           $\text{ASSERT } (ST' = ST);$ 
           $\text{let } ST = \text{tl } ST';$ 
           $\text{let } P = \text{insert } v P;$ 
           $\text{RETURN } (P, ST, r)$ 
        }
      }
    }
  }
)  $(P, [], a_0);$ 
 $\text{RETURN } (r, P)$ 
}

```

definition *liveness-compatible* **where** *liveness-compatible* $P \equiv$

$$\begin{aligned}
& (\forall x x' y. x \rightarrow y \wedge x' \in P \wedge x \preceq x' \longrightarrow (\exists y' \in P. y \preceq y')) \wedge \\
& (\forall s' \in P. \forall s. s \preceq s' \wedge V s \longrightarrow \\
& \quad \neg (\lambda x y. x \rightarrow y \wedge (\exists x' \in P. \exists y' \in P. x \preceq x' \wedge y \preceq y'))^{++} s s)
\end{aligned}$$

definition *dfs-spec* \equiv

```

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

```

end

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

lemma *check-loop-loop*: $\exists v' \in \text{set } ST. v' \preceq v$ **if** *check-loop* *v ST*
 $\langle \text{proof} \rangle$

lemma *check-loop-no-loop*: $v \notin \text{set } ST$ **if** $\neg \text{check-loop } v ST$
 $\langle \text{proof} \rangle$

lemma *mono*:

$a \preceq b \implies a \rightarrow a' \implies \forall b. b \implies \exists b'. b \rightarrow b' \wedge a' \preceq b'$
 $\langle \text{proof} \rangle$

context

fixes *P* :: '*a set* **and** *E1 E2* :: '*a* \Rightarrow '*a* \Rightarrow *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* $\langle \text{proof} \rangle$

interpretation *G'*: *Graph-Defs E2* $\langle \text{proof} \rangle$

interpretation *SG*: *Subgraph E2 E1* $\langle \text{proof} \rangle$

interpretation *SG'*: *Subgraph-Start E a₀ E1* $\langle \text{proof} \rangle$

interpretation *SG''*: *Subgraph-Start E a₀ E2* $\langle \text{proof} \rangle$

lemma *G-subgraph-reaches*[*intro*]:
G.G'.reaches a b **if** *G.reaches a b*
 $\langle \text{proof} \rangle$

lemma *G'-subgraph-reaches*[*intro*]:
G.G'.reaches a b **if** *G'.reaches a b*
 $\langle \text{proof} \rangle$

lemma *liveness-compatible-extend*:

assumes

$\forall s V v s \preceq v$

liveness-compatible P

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

E2⁺⁺ s s

shows *False*

$\langle \text{proof} \rangle$

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

lemma *liveness-compatible-extend'*:

assumes

$\forall s \ V \ v \ s \preceq \ s' \ s' \in P$
 $\forall va. \ v \rightarrow va \longrightarrow (\exists x \in P. \ va \preceq x)$
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 $V \ v \ \textit{liveness-compatible} \ P \ \forall va. \ v \rightarrow va \longrightarrow (\exists x \in P. \ va \preceq x)$

shows *liveness-compatible* (*insert* $v \ P$)

⟨*proof*⟩

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

⟨*proof*⟩

lemma *pre-cycle-cycle*:

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

⟨*proof*⟩

lemma *reachable-alt*:

$V \ v \ \textit{if} \ V \ a_0 \ a_0 \rightarrow^* \ v$

⟨*proof*⟩

lemma *dfs-correct*:

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

```

⟨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 (λdfs (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) (λv' (P,ST,-). dfs (P,ST,v'))
            (P,ST,False);
            ASSERT (ST' = ST);
            let ST = tl ST';
            let P = insert v P;
            RETURN (P, ST, r)
          }
        }
      }
    ) (P,[],a0);
    RETURN (r, P)
  }

```

end

```

locale Liveness-Search-Space =
  Liveness-Search-Space-Defs +
  Liveness-Search-Space-pre +
  assumes succs-correct: V a ⇒ 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]*:

$(\text{succs } a, \text{succs } b) \in \langle \text{Id} \rangle \text{list-rel}$ **if** $(a, b) \in \text{Id}$
 $\langle \text{proof} \rangle$

lemma *start-ref[refine]*:

$((P, [], a_0), P, [], a_0) \in \text{Id} \times_r \text{br id } (\lambda xs. \text{set } xs \subseteq \{x. V x\}) \times_r \text{br id } V$
if $V a_0$
 $\langle \text{proof} \rangle$

lemma *refine-aux[refine]*:

$((x, x1c, \text{True}), x', x1a, \text{True}) \in \text{Id} \times_r \text{br id } (\lambda xs. \text{set } xs \subseteq \text{Collect } V) \times_r \text{Id}$
if $(x1c, x1a) \in \text{br id } (\lambda xs. \text{set } xs \subseteq \{x. V x\}) (x, x') \in \text{Id}$
 $\langle \text{proof} \rangle$

lemma *refine-loop*:

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

lemma *dfs1-dfs-ref[refine]*:

$\text{dfs1 } P \leq \Downarrow \text{Id } (\text{dfs } P)$ **if** $V a_0$
 $\langle \text{proof} \rangle$

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 **for** $E :: 'v \Rightarrow 'v \Rightarrow \text{bool}$ +

fixes $\text{key} :: 'v \Rightarrow 'k$

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

begin

sublocale *Search-Space-Key-Defs* **where** $\text{empty} = \text{undefined}$ **and** $\text{subsumes}' = \text{subsumes}' \langle \text{proof} \rangle$

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

definition *insert-map-set* $v \ S \equiv$

let $k = \text{key } v$; $S' = (\text{case } S \ k \ \text{of } \text{Some } S \Rightarrow S \mid \text{None} \Rightarrow \{\})$ *in* $S(k \mapsto (\text{insert } v \ S'))$

definition *push-map-list* $v \ S \equiv$

let $k = \text{key } v$; $S' = (\text{case } S \ k \ \text{of } \text{Some } S \Rightarrow S \mid \text{None} \Rightarrow [])$ *in* $S(k \mapsto v \ \# \ S')$

definition *check-subsumption-map-set* $v \ S \equiv$

let $k = \text{key } v$; $S' = (\text{case } S \ k \ \text{of } \text{Some } S \Rightarrow S \mid \text{None} \Rightarrow \{\})$ *in* $(\exists x \in S'. v \trianglelefteq x)$

definition *check-subsumption-map-list* $v \ S \equiv$

let $k = \text{key } v$; $S' = (\text{case } S \ k \ \text{of } \text{Some } S \Rightarrow S \mid \text{None} \Rightarrow [])$ *in* $(\exists x \in \text{set } S'. x \trianglelefteq v)$

definition *pop-map-list* $v \ S \equiv$

let $k = \text{key } v$; $S' = (\text{case } S \ k \ \text{of } \text{Some } S \Rightarrow \text{tl } S \mid \text{None} \Rightarrow [])$ *in* $S(k \mapsto S')$

definition *dfs-map* $:: ('k \rightarrow 'v \ \text{set}) \Rightarrow (\text{bool} \times ('k \rightarrow 'v \ \text{set})) \ \text{nres}$ **where**

dfs-map $P \equiv \text{do } \{$

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

```

    if check-subsumption-map-list v ST then RETURN (P, ST, True)
  else do {
    if check-subsumption-map-set v P then
      RETURN (P, ST, False)
    else do {
      let ST = push-map-list v ST;
      (P, ST, r) ←
        nfoldli (succs v) (λ(-,-,b). ¬b) (λv' (P,ST,-). dfs (P,ST,v'))
      (P,ST,False);
      let ST = pop-map-list v ST;
      let P = insert-map-set v P;
      RETURN (P, ST, r)
    }
  }
) (P,(Map.empty::('k → 'v list)),a0);
RETURN (r, P)
}

```

end

locale *Liveness-Search-Space-Key* =

Liveness-Search-Space + *Liveness-Search-Space-Key-Defs* +
assumes *subsumes-key*[*intro*, *simp*]: $a \trianglelefteq b \implies \text{key } a = \text{key } b$
assumes *V-subsumes'*: $\forall a \implies a \preceq b \iff a \trianglelefteq b$

begin

definition

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

definition

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

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

lemma *empty-map-list-rel*:

$(Map.empty, []) \in map-list-rel$
 $\langle proof \rangle$

lemma *rel-start[refine]*:

$((P, Map.empty, a_0), P', [], a_0) \in map-set-rel \times_r map-list-rel \times_r Id$ **if**
 $(P, P') \in map-set-rel$
 $\langle proof \rangle$

lemma *refine-True*:

$(x1b, x1) \in map-set-rel \implies (x1c, x1a) \in map-list-rel$
 $\implies ((x1b, x1c, True), x1, x1a, True) \in map-set-rel \times_r map-list-rel \times_r Id$
 $\langle proof \rangle$

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

$V x2a \implies (x1b, x1) \in map-set-rel \implies check-subsumption-map-set x2a$
 $x1b = (\exists x \in x1. x2a \preceq x)$
 $\langle proof \rangle$

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

$set\ xs \subseteq \{x. V x\} \implies (m, xs) \in map-list-rel$
 $\implies check-subsumption-map-list\ x\ m = check-loop\ x\ xs$
 $\langle proof \rangle$

lemma *not-check-loop-non-elem*:

$x \notin set\ xs$ **if** $\neg check-loop-list\ x\ xs$
 $\langle proof \rangle$

lemma *insert-ref[refine]*:

$(x1b, x1) \in map-set-rel \implies$
 $(x1c, x1a) \in \langle Id \rangle list-set-rel \implies$
 $\neg check-loop-list\ x2a\ x1c \implies$
 $((x1b, x2a \# x1c, False), x1, insert\ x2a\ x1a, False) \in map-set-rel \times_r$
 $list-set-hd-rel\ x2a \times_r Id$
 $\langle proof \rangle$

lemma *insert-map-set-ref*:

$(m, S) \in map-set-rel \implies (insert-map-set\ x\ m, insert\ x\ S) \in map-set-rel$
 $\langle proof \rangle$

lemma *map-list-rel-memD*:

assumes $(m, xs) \in map-list-rel$ $x \in set\ xs$
obtains xs' **where** $x \in set\ xs'$ $m\ (key\ x) = Some\ xs'$
 $\langle proof \rangle$

lemma *map-list-rel-memI*:

$(m, xs) \in \text{map-list-rel} \implies m\ k = \text{Some } xs' \implies x' \in \text{set } xs' \implies x' \in \text{set } xs$
<proof>

lemma *map-list-rel-grouped-by-key*:

$x' \in \text{set } xs' \implies (m, xs) \in \text{map-list-rel} \implies m\ k = \text{Some } xs' \implies \text{key } x' = k$
<proof>

lemma *map-list-rel-not-memI*:

$k \neq \text{key } x \implies m\ k = \text{Some } xs' \implies (m, xs) \in \text{map-list-rel} \implies x \notin \text{set } xs'$
<proof>

lemma *map-list-rel-not-memI2*:

$x \notin \text{set } xs' \text{ if } m\ a = \text{Some } xs' \implies (m, xs) \in \text{map-list-rel} \implies x \notin \text{set } xs$
<proof>

lemma *push-map-list-ref*:

$x \notin \text{set } xs \implies (m, xs) \in \text{map-list-rel} \implies (\text{push-map-list } x\ m, x \# xs) \in \text{map-list-rel}$
<proof>

lemma *insert-map-set-ref'[refine]*:

$(x1b, x1) \in \text{map-set-rel} \implies$
 $(x1c, x1a) \in \text{map-set-rel} \implies$
 $\neg \text{check-subsumption}'\ x2a\ x1c \implies$
 $((x1b, \text{insert-map-set } x2a\ x1c, \text{False}), x1, \text{insert } x2a\ x1a, \text{False}) \in \text{map-set-rel}$
 $\times_r \text{map-set-rel} \times_r \text{Id}$
<proof>

lemma *map-list-rel-check-subsumption-map-list*:

$\text{set } xs \subseteq \{x. \forall x\} \implies (m, xs) \in \text{map-list-rel} \implies \neg \text{check-subsumption-map-list } x\ m \implies x \notin \text{set } xs$
<proof>

lemma *push-map-list-ref'[refine]*:

$\text{set } x1a \subseteq \{x. \forall x\} \implies$
 $(x1b, x1) \in \text{map-set-rel} \implies$
 $(x1c, x1a) \in \text{map-list-rel} \implies$
 $\neg \text{check-subsumption-map-list } x2a\ x1c \implies$
 $((x1b, \text{push-map-list } x2a\ x1c, \text{False}), x1, x2a \# x1a, \text{False}) \in \text{map-set-rel}$
 $\times_r \text{map-list-rel} \times_r \text{Id}$
<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* $S \subseteq S'$
<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 \implies
(st, ST) ∈ list-set-hd-rel x \implies
(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 $\leq \Downarrow (Id \times_r \text{map-set-rel}) (dfs1 P')$ **if** $(P, P') \in \text{map-set-rel}$
<proof>

lemma *dfs-map-dfs-refine*:

dfs-map P $\leq \Downarrow (Id \times_r \text{map-set-rel}) (dfs P')$ **if** $(P, P') \in \text{map-set-rel}$ $V a_0$
<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* ($\langle \cdot \rangle := \rightarrow$ 62)

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

Liveness-Search-Space-Key-Defs - - - - - key **for** key :: 'a ⇒ 'k +
fixes A :: 'a ⇒ ('ai :: heap) ⇒ assn
fixes succsi :: 'ai ⇒ 'ai list Heap
fixes a₀i :: 'ai Heap
fixes Lei :: 'ai ⇒ 'ai ⇒ bool Heap
fixes keyi :: 'ai ⇒ 'ki :: {hashable, heap} Heap
fixes copyi :: 'ai ⇒ 'ai Heap

locale *Liveness-Search-Space-Key-Impl* =
Liveness-Search-Space-Key-Impl-Defs +
Liveness-Search-Space-Key +
fixes K
assumes pure-K[*safe-constraint-rules*]: is-pure K
assumes left-unique-K[*safe-constraint-rules*]: IS-LEFT-UNIQUE (the-pure K)
assumes right-unique-K[*safe-constraint-rules*]: IS-RIGHT-UNIQUE (the-pure K)
assumes refinements[*sepref-fr-rules*]:
 (uncurry0 a₀i, uncurry0 (RETURN (PR-CONST a₀))) ∈ unit-assn^k
 →_a A
 (uncurry Lei, uncurry (RETURN oo PR-CONST (≤))) ∈ A^k *_a A^k →_a
 bool-assn
 (succsi, RETURN o PR-CONST succs) ∈ A^k →_a list-assn A
 (keyi, RETURN o PR-CONST key) ∈ A^k →_a K
 (copyi, RETURN o COPY) ∈ A^k →_a A

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

— The lemma has this form to avoid unwanted eta-expansions. The eta-expansions arise from the type of *insert-map-set v S*.

lemma *insert-map-set-alt-def*: ((), *insert-map-set v S*) = (
 let
 k = key v; (S', S) = op-map-extract k S
 in
 case S' of
 Some S1 ⇒ ((), S(k ↦ (insert v S1)))
 | None ⇒ ((), S(k ↦ {v}))
)

⟨proof⟩

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

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

$\langle \text{proof} \rangle$

lemma *check-subsumption-map-set-extract*: $(S, \text{check-subsumption-map-set } v \ S) = ($

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

$\langle \text{proof} \rangle$

lemma *check-subsumption-map-list-extract*: $(S, \text{check-subsumption-map-list } v \ S) = ($

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

$\langle \text{proof} \rangle$

lemma *push-map-list-alt-def*: $((), \text{push-map-list } v \ S) = ($

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

<proof>
lemma *pop-map-list-alt-def*: $(((), \text{pop-map-list } v \ S) = ($
 let
 $k = \text{key } v; (S', S) = \text{op-map-extract } k \ S$
 in
 case S' *of*
 $\text{Some } S1 \Rightarrow (((), S(k \mapsto \text{if op-list-is-empty } S1 \text{ then } [] \text{ else } \text{tl } S1))$
 $| \text{None} \Rightarrow (((), S(k \mapsto []))$
)

<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*:
 $\text{dfs-map} = (\lambda \ P. \ \text{do } \{$
 $(P, ST, r) \leftarrow \text{RECT } (\lambda \text{dfs } (P, ST, v).$
 let $(ST, b) = (ST, \text{check-subsumption-map-list } v \ ST)$ *in*
 if b *then* $\text{RETURN } (P, ST, \text{True})$
 else *do* $\{$
 let $(P, b1) = (P, \text{check-subsumption-map-set } v \ P)$ *in*
 if $b1$ *then*
 $\text{RETURN } (P, ST, \text{False})$
 else *do* $\{$
 let $(-, ST1) = (((), \text{push-map-list } (\text{COPY } v) \ ST);$
 $(P1, ST2, r) \leftarrow$
 $\text{nfoldli } (\text{succs } v) (\lambda(-, -, b). \neg b) (\lambda v' (P, ST, -). \text{dfs } (P, ST, v'))$
 $(P, ST1, \text{False});$
 let $(-, ST')$ $= (((), \text{pop-map-list } (\text{COPY } v) \ ST2);$
 let $(-, P')$ $= (((), \text{insert-map-set } (\text{COPY } v) \ P1);$
 $\text{TRACE } (\text{ExploredState});$
 $\text{RETURN } (P', ST', r)$
 $\}$
 $\}$
) $(P, \text{Map.empty}, a_0);$
 $\text{RETURN } (r, P)$
)

<proof>

definition *dfs-map'* **where**
 $\text{dfs-map}' \ a \ P \equiv \ \text{do } \{$
 $(P, ST, r) \leftarrow \text{RECT } (\lambda \text{dfs } (P, ST, v).$

```

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*:
 $dfs\text{-map}' a_0 = dfs\text{-map}$
 $\langle proof \rangle$

end

definition (in *imp-map-delete*) [code-unfold]: $hms\text{-delete} = delete$

lemma (in *imp-map-delete*) *hms-delete-rule* [sep-heap-rules]:
 $\langle hms\text{-assn } A \ m \ mi \rangle \ hms\text{-delete } k \ mi \ \langle hms\text{-assn } A \ (m(k := None)) \rangle_t$
 $\langle proof \rangle$

context *imp-map-delete*
begin

lemma *hms-delete-hnr*:
 $(uncurry \ hms\text{-delete}, \ uncurry \ (RETURN \ oo \ op\text{-map-delete})) \in$
 $id\text{-assn}^k *_a (hms\text{-assn } A)^d \rightarrow_a hms\text{-assn } A$
 $\langle proof \rangle$

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

end

lemma *fold-insert-set*:

fold insert xs S = set xs \cup S
<proof>

lemma *set-alt-def*:

set xs = fold insert xs {}
<proof>

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

sempref-register

PR-CONST a₀ PR-CONST (\trianglelefteq) PR-CONST succs PR-CONST key

lemma [*def-pat-rules*]:

a₀ \equiv UNPROTECT a₀ (\trianglelefteq) \equiv UNPROTECT (\trianglelefteq) succs \equiv UNPROTECT succs key \equiv UNPROTECT key
<proof>

abbreviation *passed-assn \equiv 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>

sempref-definition *dfs-map-impl is*

PR-CONST dfs-map :: passed-assn^d \rightarrow_a prod-assn bool-assn passed-assn
<proof>

sempref-register *dfs-map*

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

lemma *passed-empty-refine*[*sempref-fr-rules*]:

(uncurry0 hm.hms-empty, uncurry0 (RETURN (PR-CONST hm.op-hms-empty)))
 \in unit-assn^k \rightarrow_a passed-assn
<proof>

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

sempref-thm *dfs-map-impl' is*

$uncurry0 (do \{(r, p) \leftarrow dfs\text{-}map \text{Map.empty}; RETURN r\}) :: unit\text{-}assn^k$
 $\rightarrow_a bool\text{-}assn$
 $\langle proof \rangle$

sempref-definition *dfs-map'-impl* **is**

$uncurry \text{dfs-map}'$
 $:: A^d *_a (hm.hms\text{-}assn' K (lso\text{-}assn A))^d \rightarrow_a bool\text{-}assn \times_a hm.hms\text{-}assn'$
 $K (lso\text{-}assn A)$
 $\langle proof \rangle$

concrete-definition (**in** $-$) *dfs-map-impl'*

uses *Liveness-Search-Space-Key-Impl.dfs-map-impl'.refine-raw* **is** $(uncurry0$
 $?f, -) \in -$

lemma (**in** *Liveness-Search-Space-Key*) *dfs-map-empty*:

$dfs\text{-}map \text{Map.empty} \leq \Downarrow (bool\text{-}rel \times_r map\text{-}set\text{-}rel) \text{dfs-spec}$ **if** $V a_0$
 $\langle proof \rangle$

lemma (**in** *Liveness-Search-Space-Key*) *dfs-map-empty-correct*:

$do \{(r, p) \leftarrow dfs\text{-}map \text{Map.empty}; RETURN r\} \leq SPEC (\lambda r. r \longleftrightarrow (\exists$
 $x. a_0 \rightarrow^* x \wedge x \rightarrow^+ x))$
if $V a_0$
 $\langle proof \rangle$

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

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

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

$\langle \uparrow (V a_0) \rangle$
 $dfs\text{-}map\text{-}impl' \text{succsi } a_0 i \text{Lei keyi copyi}$
 $\langle \lambda r. \uparrow (r \longleftrightarrow (\exists x. a_0 \rightarrow^* x \wedge x \rightarrow^+ x)) \rangle_t$
 $\langle proof \rangle$

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** $m\ k = Some\ v$
 $\langle proof \rangle$

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*:

$\langle is-hashmap\ m\ ht \rangle\ hm-it-next-key\ ht\ \langle \lambda r. is-hashmap\ m\ ht\ * \uparrow (r \in dom\ m) \rangle$
if $m \neq Map.empty$
 $\langle proof \rangle$

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
 $\langle hm.assn\ K\ V\ m\ mi \rangle$
 $hm-it-next-key\ mi$
 $\langle \lambda r. \exists_A xa. hm.assn\ K\ V\ m\ mi\ * K\ xa\ r\ * true\ * \uparrow (RETURN\ xa \leq next-key\ m) \rangle$

$\langle \text{proof} \rangle$

lemma *hm-it-next-key-next-key*:

assumes *CONSTRAINT is-pure K*

shows $(\text{hm-it-next-key}, \text{next-key}) \in (\text{hm.assn } K \ V)^k \rightarrow_a K$

$\langle \text{proof} \rangle$

lemma *hm-it-next-key-next-key'*:

$(\text{hm-it-next-key}, \text{next-key}) \in (\text{hm.hms-assn } V)^k \rightarrow_a \text{id-assn}$

$\langle \text{proof} \rangle$

lemma *no-fail-next-key-iff*:

$\text{nofail } (\text{next-key } m) \longleftrightarrow m \neq \text{Map.empty}$

$\langle \text{proof} \rangle$

context

fixes *mi m K*

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

begin

private lemma *k-aux*:

assumes $k \in \text{dom } \text{mi} \ (\text{mi}, m) \in \langle K, \text{Id} \rangle \text{map-rel}$

shows $\exists k'. (k, k') \in K$

$\langle \text{proof} \rangle$ **lemma** *k-aux2*:

assumes $k \in \text{dom } \text{mi} \ (k, k') \in K$

shows $k' \in \text{dom } m$

$\langle \text{proof} \rangle$ **lemma** *map-empty-iff*: $\text{mi} \neq \text{Map.empty} \longleftrightarrow m \neq \text{Map.empty}$

$\langle \text{proof} \rangle$ **lemma** *aux*:

assumes *RETURN k ≤ next-key mi*

shows *RETURN (SOME k'. (k, k') ∈ K) ≤ next-key m*

$\langle \text{proof} \rangle$ **lemma** *aux1*:

assumes *RETURN k ≤ next-key mi nofail (next-key m)*

shows $(k, \text{SOME } k'. (k, k') \in K) \in K$

$\langle \text{proof} \rangle$

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

end

lemma *hm-it-next-key-next-key''*:

assumes *is-pure K*

shows $(\text{hm-it-next-key}, \text{next-key}) \in (\text{hm.hms-assn}' K \ V)^k \rightarrow_a K$

$\langle \text{proof} \rangle$

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 <λr. 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) ∈ (hms-assn A)k →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 →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

```

sempref-definition *ran-of-map-impl'* **is**
ran-of-map :: (*hm.hms-assn* *A*)^d →_a *list-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* ≼ *x'* *V* *x* *V* *x'*

shows ∃ *y'* *xs'* *ys'*. *y* ≼ *y'* ∧ *G'.steps* (*x'* # *xs'* @ *y'* # *ys'* @ [*y'*])

⟨*proof*⟩

lemma *reaches-cycle-mono*:

assumes *G'.reaches* *x* *y* *y* →⁺ *y* *x* ≼ *x'* *V* *x* *V* *x'*

shows ∃ *y'*. *y* ≼ *y'* ∧ *G'.reaches* *x'* *y'* ∧ *y'* →⁺ *y'*

⟨*proof*⟩

including *reaches-steps-iff*

⟨*proof*⟩

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

end

end

locale *Leadsto-Search-Space* =

A: *Search-Space'-finite* *E* *a*₀ - (≼) *empty*

for *E* *a*₀ *empty* **and** *subsumes* :: '*a* ⇒ '*a* ⇒ *bool* (**infix** <≼> 50)

```

+
fixes P Q :: 'a ⇒ bool
assumes P-mono: a ≼ a' ⇒ ¬ empty a ⇒ P a ⇒ P a'
assumes Q-mono: a ≼ a' ⇒ ¬ empty a ⇒ Q a ⇒ Q a'
fixes succs-Q :: 'a ⇒ 'a list
assumes succs-Q-correct: A.reachable a ⇒ set (succs-Q a) = {y. E a y
∧ Q y ∧ ¬ empty y}
begin

sublocale A': Search-Space'-finite E a0 λ -. False (≼) empty
  ⟨proof⟩

sublocale B:
  Liveness-Search-Space
  λ x y. E x y ∧ Q y ∧ ¬ empty y a0 λ -. False (≼) λ x. A.reachable x ∧ ¬
empty x
  succs-Q
  ⟨proof⟩

context
  fixes a1 :: 'a
begin

interpretation B':
  Liveness-Search-Space
  λ x y. E x y ∧ Q y ∧ ¬ empty y a1 λ -. False (≼) λ x. A.reachable x ∧ ¬
empty x succs-Q
  ⟨proof⟩

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

end

definition leadsto :: bool nres where
  leadsto = do {
    (r, passed) ← A'.pw-algo;
    let P = {x. x ∈ passed ∧ P x ∧ Q x};
    (r, -) ←
      FOREACHC P (λ(b,-). ¬b) (λv' (-,P). has-cycle v' P) (False,{});
    RETURN r
  }

definition

```

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

definition *leadsto-spec* **where**

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

lemma

$leadsto \leq leadsto-spec$
 $\langle proof \rangle$

definition *leadsto-spec-alt* **where**

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

lemma *E-reaches-non-empty*:

$(\lambda x\ y. E\ x\ y \wedge \neg\ empty\ y)**\ a\ b$ **if** $a \rightarrow^* b$ $A.reachable\ a \wedge \neg\ empty\ b$ **for**
 $a\ b$
 $\langle proof \rangle$

lemma *leadsto-spec-leadsto-spec-alt*:

$leadsto-spec \leq leadsto-spec-alt$
 $\langle proof \rangle$

end

end

7.2 Implementation on Maps

theory *Leadsto-Map*

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

begin

definition *map-to-set* $:: ('b \rightarrow 'a\ set) \Rightarrow 'a\ set$ **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)
  (λa (passed, wait, -).
    do {
      RETURN (
        if empty a then
          (passed, wait, False)
        else if F' a then (passed, wait, True)
        else
          let k = key a; passed' = (case passed k of Some passed' ⇒ passed' |
None ⇒ [])
          in
            if ∃ x ∈ set passed'. a ≤ x then
              (passed, wait, False)
            else
              (passed(k ↦ (a # passed')), a # wait, False)
          )
        }
    )
  (passed, wait, False)

```

definition

```

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

```

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

```

pw-algo-map3 = do
  {
    if F a0 then RETURN (True, Map.empty)
  }

```

```

    else if empty a0 then RETURN (False, Map.empty)
    else do {
      (passed, wait) ← RETURN ([key a0 ↦ [a0]], [a0]);
      (passed, wait, brk) ← WHILEIT pw-map-inv3 (λ (passed, wait, brk).
    ¬ brk ∧ wait ≠ [])
      (λ (passed, wait, brk). do
        {
          (a, wait) ← take-from-list wait;
          ASSERT (reachable a);
          if empty a then RETURN (passed, wait, brk) else add-pw'-map3
passed wait a
        }
      )
      (passed, wait, False);
      RETURN (brk, passed)
    }
  }
}

```

end — Worklist Map 2 Defs

lemma *map-list-set-rel-empty*[*refine, simp, intro*]:

(*Map.empty*, *Map.empty*) ∈ *map-list-set-rel*
 ⟨*proof*⟩

lemma *map-list-set-rel-single*:

(*ml*(*key* a₀ ↦ [a₀]), *ms*(*key* a₀ ↦ {a₀})) ∈ *map-list-set-rel* **if** (*ml*, *ms*) ∈
map-list-set-rel
 ⟨*proof*⟩

context *Worklist-Map2*

begin

lemma *refine-start*[*refine*]:

(([*key* a₀ ↦ [a₀]], [a₀]), [*key* a₀ ↦ {a₀}], [a₀]) ∈ *map-list-set-rel* ×_{*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*

$\times_r Id \times_r Id$
 $\langle proof \rangle$

lemma *map-list-set-relD*:

$ms\ k = Some\ (set\ xs)$ **if** $(ml, ms) \in map\text{-}list\text{-}set\text{-}rel$ $ml\ k = Some\ xs$
 $\langle proof \rangle$

lemma *map-list-set-rel-distinct*:

distinct xs **if** $(ml, ms) \in map\text{-}list\text{-}set\text{-}rel$ $ml\ k = Some\ xs$
 $\langle proof \rangle$

lemma *map-list-set-rel-NoneD1*[*dest, intro*]:

$ms\ k = None$ **if** $(ml, ms) \in map\text{-}list\text{-}set\text{-}rel$ $ml\ k = None$
 $\langle proof \rangle$

lemma *map-list-set-rel-NoneD2*[*dest, intro*]:

$ml\ k = None$ **if** $(ml, ms) \in map\text{-}list\text{-}set\text{-}rel$ $ms\ k = None$
 $\langle proof \rangle$

lemma *map-list-set-rel-insert*:

$(ml, ms) \in map\text{-}list\text{-}set\text{-}rel \implies$
 $ml\ (key\ a) = Some\ xs \implies$
 $ms\ (key\ a) = Some\ (set\ xs) \implies$
 $a \notin set\ xs \implies$
 $(ml(key\ a \mapsto a \# xs), ms(key\ a \mapsto insert\ a\ (set\ xs))) \in map\text{-}list\text{-}set\text{-}rel$
 $\langle proof \rangle$

lemma *add-pw'-map3-ref*:

$add\text{-}pw'\text{-}map3\ ml\ xs\ a \leq \Downarrow (map\text{-}list\text{-}set\text{-}rel \times_r Id) (add\text{-}pw'\text{-}map2\ ms\ xs$
 $a)$
if $(ml, ms) \in map\text{-}list\text{-}set\text{-}rel \neg empty\ a$
 $\langle proof \rangle$

lemma *pw-algo-map3-ref*[*refine*]:

$pw\text{-}algo\text{-}map3 \leq \Downarrow (Id \times_r map\text{-}list\text{-}set\text{-}rel) pw\text{-}algo\text{-}map2$
 $\langle proof \rangle$

lemma *pw-algo-map2-ref'*:

$pw\text{-}algo\text{-}map2 \leq \Downarrow (bool\text{-}rel \times_r map\text{-}set\text{-}rel) pw\text{-}algo$
 $\langle proof \rangle$

lemma *pw-algo-map3-ref'*[*refine*]:

$pw\text{-}algo\text{-}map3 \leq \Downarrow (bool\text{-}rel \times_r (map\text{-}list\text{-}set\text{-}rel\ O\ map\text{-}set\text{-}rel)) pw\text{-}algo$
 $\langle proof \rangle$

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 a₀ 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 a₀ λ -. False (≼) λ x. A.reachable x ∧ ¬
empty x
 succs-Q key
 ⟨*proof*⟩

context
 fixes *a₁ :: 'a*
begin

interpretation *B'*:
 Liveness-Search-Space-Key-Defs
 a₁ 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
 $\lambda\ x\ y.\ E\ x\ y\ \wedge\ Q\ y\ \wedge\ \neg\ empty\ y\ a_1\ \lambda\ -. \ False\ (\preceq)\ \lambda\ x.\ A.reachable\ x\ \wedge\ \neg\ empty\ x$
succs-Q key
 $\langle proof \rangle$

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 $\equiv\ \lambda\ (r,\ passed')$
 $(r \longrightarrow (\exists\ a.\ A.reachable\ a\ \wedge\ \neg\ empty\ a\ \wedge\ P\ a\ \wedge\ Q\ a\ \wedge\ reaches-cycle\ a))$
 $\wedge\ (\neg\ r \longrightarrow$
 $\quad (\forall\ a \in \bigcup\ done.\ P\ a\ \wedge\ Q\ a \longrightarrow \neg\ reaches-cycle\ a)$
 $\quad \wedge\ B.liveness-compatible\ passed'$
 $\quad \wedge\ passed' \subseteq \{x.\ A.reachable\ x\ \wedge\ \neg\ empty\ x\}$
 $\quad)$

definition *inner-inv* **where**

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

definition *leadsto'* **:: bool nres where**

leadsto' $=\ do\ \{$
 $\quad (r,\ passed) \leftarrow A'.pw-algo-map2;$
 $\quad let\ passed = ran\ passed;$
 $\quad (r,\ -) \leftarrow FOREACHcdi\ (outer-inv\ passed)\ passed\ (\lambda(b,-).\ \neg b)$
 $\}$


```

    }
  )
  (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)
    (λv' (-,passed).
      if P v' ∧ Q v' then has-cycle-map v' passed else RETURN (False,
passed)
    )
  )
  (False, acc)
)
  (False, Map.empty);
  RETURN r
}

```

lemma *start-ref*:

((False, Map.empty), False, Map.empty) ∈ *Id* ×_{*r*} *map-list-set-rel*
 ⟨*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*:

$(x'a, x'a) \in \langle Id \rangle list-rel$
 $\langle proof \rangle$

lemma *map-list-set-rel-finite-ran*:
finite (ran ml) if (ml, ms) ∈ map-list-set-rel
 $\langle proof \rangle$

lemma *leadsto-map3-ref[refine]*:
leadsto-map3 ≤ ↓ Id leadsto'
 $\langle proof \rangle$

definition *leadsto-map4* :: *bool nres where*

```

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

```

lemma *ran-of-map-ref*:
ran-of-map m ≤ SPEC (λc. (c, ran m') ∈ br set (λ -. True)) if finite (dom m) (m, m') ∈ Id
 $\langle proof \rangle$

lemma *aux-ref*:
 $(xa, x'a) \in Id \implies$
 $x'a = (x1b, x2b) \implies xa = (x1c, x2c) \implies (x1c, x1b) \in bool-rel$
 $\langle proof \rangle$

definition *foo* = *A'.pw-algo-map3*

lemma [*refine*]:
A'.pw-algo-map3 ≤ ↓ (br id (λ (-, m). finite (dom m))) foo
 $\langle proof \rangle$

lemma *leadsto-map4-ref[refine]*:
leadsto-map4 \leq \Downarrow *Id* *leadsto-map3*
 \langle *proof* \rangle

definition *leadsto-map4'* :: *bool nres* **where**

```

leadsto-map4' = do {
  (r, passed)  $\leftarrow$  A'.pw-algo-map2;
  ASSERT (finite (dom passed));
  passed  $\leftarrow$  ran-of-map passed;
  (r, -)  $\leftarrow$  nfoldli passed ( $\lambda(b,-). \neg b$ )
  ( $\lambda$  passed' (-, acc).
    do {
      passed'  $\leftarrow$  SPEC ( $\lambda l. \text{set } l = \text{passed}'$ );
      nfoldli passed' ( $\lambda(b,-). \neg b$ )
      ( $\lambda v' (-, \text{passed})$ .
        if P v'  $\wedge$  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'*
 \langle *proof* \rangle

lemma *leadsto-map4'-correct*:
leadsto-map4' \leq *leadsto-spec-alt*
 \langle *proof* \rangle

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*:

$(return\ o\ id, list-of-set) \in (lso-assn\ A)^d \rightarrow_a list-assn\ A$
 $\langle proof \rangle$

sepref-register *hm.op-hms-empty***context** *Worklist-Map2-Impl***begin****sepref-thm** *pw-algo-map2-impl* **is**

$uncurry0\ (pw-algo-map2) ::$
 $unit-assn^k \rightarrow_a bool-assn \times_a (hm.hms-assn'\ K\ (lso-assn\ A))$
 $\langle proof \rangle$

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* $A\ succsi\ a_0i\ Lei\ keyi\ copyi$
for *key* $:: 'v \Rightarrow 'k$
and $a_0\ F\ F'\ copyi\ P\ Q\ V\ empty\ succs-Q\ succs1\ E\ A\ succsi\ a_0i\ Lei\ keyi +$
fixes *succs1i* **and** *emptyi* **and** $Pi\ Qi$ **and** *tracei*
assumes *succs1-impl*: $(succs1i, (RETURN \circ\circ\ PR-CONST)\ succs1) \in$
 $A^k \rightarrow_a list-assn\ A$
and *empty-impl*:
 $(emptyi, RETURN\ o\ PR-CONST\ empty) \in A^k \rightarrow_a bool-assn$
assumes [*sepref-fr-rules*]:
 $(Pi, RETURN\ o\ PR-CONST\ P) \in A^k \rightarrow_a bool-assn$ $(Qi, RETURN\ o$
 $PR-CONST\ Q) \in A^k \rightarrow_a bool-assn$
assumes *trace-impl*:
 $(uncurry\ tracei, uncurry\ (\lambda(- :: string) -. RETURN\ ())) \in id-assn^k *_a$
 $A^k \rightarrow_a id-assn$

begin**sublocale** *Worklist-Map2-Impl* - - $\lambda -.$ *False* - *succs1* - - $\lambda -.$ *False* - *succs1i*

-

$\lambda -.$ *return False Lei*

$\langle proof \rangle$

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* \equiv *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-assn^k \rightarrow_a bool-assn
<proof>

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

lemma *leadsto-impl-hnr*:
(uncurry0 (
leadsto-impl copyi succsi a₀i Lei keyi succs1i emptyi Pi Qi tracei
),
uncurry0 leadsto-spec-alt
*) \in unit-assn^k \rightarrow_a bool-assn **if** $\forall a_0$
*<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.