Verifying Schorr-Waite Graph Marking in Lightweight Separation

Holger Gast
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen


1 Object Graphs

1.1 Basic Collections of Heap Nodes

theory ObjectGraphs
imports SepAuto ExtSyntax Regions
begin

locale node-set = 
  fixes gctx :: ctx 
  and node :: "addr ⇒ cover"
  assumes node-wf-cover: "wf-cover (node p)"
  and node-contains-base: "node p S ⟷ p ∈ S"
begin

declare node-wf-cover

definition
  "nodes ≡ λN S. ((∀a ∈ N. is-valid (node a)) ∧ 
  (∀a ∈ N. ∀b ∈ N. a ≠ b ⟹ is-valid (node a || node b)) ∧ 
  S = (⋃p ∈ N. (THE S'. node p S')))"

lemma node-in-nodes:
  "p ∈ N ⟹ node p ≤ nodes N"
  apply (rule subcoverI)
  apply (clarsimp simp add: nodes-def)
  apply (rule-tac x="The (node p)" in exI)
  apply (auto intro: wf-cover-accepts-self node-wf-cover)
  done

lemma node-not-empty:
  "is-valid(node p) ⟹ node p ≠ Empty"
  by (auto dest: fun-cong node-contains-base simp add: is-valid-def Empty-def)

declare_cover
  "nodes N"
  apply (simp add: wf-cover-def nodes-def)
  done

lemma nodes-node-unfold:
  "nodes {p} = node p"
  by (auto intro!: ext simp add: is-valid-def nodes-def)
lemma nodes-unfold:
"[ \forall p. \ [ p \in A; p \in B ] \implies \text{False} 
\implies \nodes(A \cup B) = \nodes A \parallel \nodes B"
apply (rule ext iffI)+
apply (clarsimp simp add: nodes-def)
apply (rule-tac SA="\bigcup p \in A. \text{The (node p)}" and 
SB="\bigcup p \in B. \text{The (node p)}" in disjI)
apply (simp add: nodes-def)
apply (simp add: nodes-def)
apply (subst Int-UN-distrib2)
apply (clarsimp simp only: UNION-empty-conv)
apply (rule disjoint-acceptedD)
apply (subgoal-tac "i \neq j")
prefer 2
apply fast
apply fast
apply (rule node-wf-cover)+
apply simp
apply (rename-tac S)
apply (clarsimp elim!: Sep.disjE simp add: nodes-def)
apply (subst(asm) Int-UN-distrib2)
apply (simp only: UNION-empty-conv)
apply (rule conjI)
apply fast
apply fast
done

lemma node-nodes-sub:
"{ \{ p \} \subseteq N } \implies \text{node p} \preceq \nodes N"
by (simp add: node-in-nodes)

lemma nodes-sub:
"A \subseteq B \implies \nodes A \preceq \nodes B"
apply (rule subcoverI)
apply (auto simp add: nodes-def)
done

lemma nodes-unfold-complement:
"A \subseteq B \implies \nodes B = \nodes A \parallel \nodes (B - A)"
apply (subst nodes-unfold[ symmetric])
apply (auto simp add: Un-absorb1)
done
lemma nodes-disjoint-by-layout-disjoint:
"[[ is-valid (nodes P || nodes Q) ]] \implies P \cap Q = {}"
apply (rule ccontr)
apply (erule nonemptyE)
apply clarify
apply (subgoal-tac "is-valid (node x || node x)"")
apply (drule same-layout-not-disjoint)
apply (rule node-not-empty, erule sep-init-valid-valid)
apply (rule sub-match-init)
apply (rule subcover-trans[OF - subcover-left])
apply (rule node-nodes-sub)
apply simp
apply (rule node-wf-cover)
apply assumption
apply (rule sep-init-valid-valid)
apply (rule disj-rec node-nodes-sub | simp)+
done

lemma nodes-disjoint-by-allocated-disjoint:
"[[ M A; nodes P || nodes Q \preceq A ]] \implies P \cap Q = {}"
by (auto intro!: nodes-disjoint-by-layout-disjoint sep-init-allocated-valid sub-match-init)

lemma node-notin-set-by-allocated:
"[[ M A; node p || nodes Q \preceq A ]] \implies p \not\in Q"
by (auto simp add: nodes-node-unfold dest: nodes-disjoint-by-allocated-disjoint[where P="\{p\}" and Q=Q])

lemmas node-notin-set-by-covered =
  node-notin-set-by-allocated[OF sep-init-covered-allocated]

lemma nodes-insert:
"p \not\in P \implies nodes (insert p P) = node p || nodes P"
apply (subst insert-is-Un)
apply (subst nodes-unfold)
apply (auto simp add: nodes-node-unfold)
done

lemma nodes-empty[simp]:
"nodes {} = Empty"
by (simp add: nodes-def Empty-def)

lemma nodes-contain-bases:
"is-valid (nodes A) \implies A \subseteq region (nodes A)"
apply (clarsimp elim!: is-validE simp add: region-def)
apply (clarsimp simp add: nodes-def)
apply (drule(1) bspec)
apply (clarsimp simp add: is-valid-def)
apply (rule bexI[rotated], assumption)
apply (auto intro: node-contains-base)
done

lemma nodes-region-nodes-subsetD:
"[[ region (nodes A) \subseteq S; is-valid (nodes A) ]] \implies A \subseteq S"
apply (rule subset-trans[rotated], assumption)
apply (rule nodes-contain-bases, assumption)
done

end
1.2 Split-Heap Unfolding

locale split-heap-node-set = node-set + 
fixes rec :: string 
assumes node-is-record: 
"node r = typed-block gctx r (TStruct rec)"
begin

lemma split-heap-unfold: 
"[\forall r. field-block gctx r (TStruct rec) f \parallel field-block gctx r (TStruct rec) f' \leq typed-block gctx r (TStruct rec); 
p \in \mathbb{N}; q \in \mathbb{N}; \text{same-static ctx gctx} ] \implies field-block ctx p (TStruct rec) f \parallel field-block ctx q (TStruct rec) f' \leq nodes N"
apply (case-tac "p = q") 
apply (rule subcover-trans[rotated]) 
apply (rule-tac p=p in node-nodes-sub) 
apply simp 
apply (simp add: node-is-record) 
apply (rule subcover-trans[rotated]) 
apply localize
apply assumption 
apply (rule subcover-refl) 
apply (rule subcover-trans[rotated]) 
apply (rule-tac A="\{p, q\}" in nodes-sub) 
apply simp 
apply (rule subcover-trans) 
apply (rule subcover-trans[rotated]) 
apply (rule-tac A="\{p\}" and B="\{q\}" in nodes-unfold[THEN sym, THEN subcover-refl-by-eq, standard]) 
apply (rule split-heap-node-set-axioms) 
apply simp 
apply (rule disj-rec) 
apply (rule subcover-trans[rotated]) 
apply (rule node-nodes-sub) 
apply simp 
apply (simp add: node-is-record) 
apply localize 
apply (rule subcover-trans[rotated]) 
apply (drule-tac x=p in meta-spec, assumption) 
apply sub 
apply (rule subcover-trans[rotated]) 
apply (rule node-nodes-sub) 
apply simp 
apply (simp add: node-is-record) 
apply localize 
apply (rule subcover-trans[rotated]) 
apply (drule-tac x=q in meta-spec, assumption) 
apply sub 
apply (simp add: insert-commute) 
apply (rule subcover-refl)
done

lemma split-heap-sub: 
"[\forall r. field-block gctx r (TStruct rec) f \leq typed-block gctx r (TStruct rec); 
p \in \mathbb{N}; \text{same-static ctx gctx} ] \implies field-block ctx p (TStruct rec) f \leq nodes N"
apply (rule subcover-trans[rotated]) 
apply (rule-tac p=p in node-nodes-sub) 
apply simp 
apply (simp add: node-is-record) 
apply localize 
done

\footnote{\texttt{localize} uses a \texttt{same-static} assumption to rewrite functions that ignore variables to access the local context.}
2 Setup of Automatic Disjointness

2.1 Strategy for Proofs about Sets

The unfolding routine for node sets uses a simple strategy for reasoning about pointer sets: it expands the definitions of set relations and -predicates to obtain literals \( p \in S \) or \( p \notin S \). The built-in \texttt{metis} prover can handle these in its usual HOL setup without requiring further special-purpose theorems.

\begin{verbatim}
lemma transform-initial-goal:
  "(\forall n. P n \Rightarrow Q n) \Rightarrow \{n. P n \subseteq \{n. Q n\}\)
  "(\forall n. Q n \Rightarrow \{a \subseteq \{n. Q n\}\)

  by fast+

lemma embed-combine:
  "\[ A \subseteq C; B \subseteq C \] \Rightarrow A \cup B \subseteq C"

and embed-left:
  "(A :: addr set) \subseteq A \cup B"

and embed-right:
  "(B :: addr set) \subseteq A \cup B"

and embed-trivial:
  "(A :: addr set) \subseteq A"

by auto

lemma drop-meta-all:
  "(\forall r. PROP P) \equiv PROP P"

by simp

lemmas atomize-set-assertions =
  subset-iff Diff-iff Int-iff Un-iff all-not-in-conv[ symmetric ] singleton-iff insert-iff
\end{verbatim}

2.2 Setup of the Unfolding Provers

ML sources cannot be loaded via \texttt{use} within the locale environment, and yet the sources need to refer to the locale theorems by usual antiquotations. We take a simple workaround by including the sources and then declaring the localized setup function within the locale context.

\begin{verbatim}
use "nodesunfold.ML"

declaration (in node-set) {* NodesUnfold.declare-unfoldings *}
\end{verbatim}

2.3 Object Graphs Defined by Reachability

locale object-graph = node-set +
  fixes succs :: "addr \Rightarrow memory \Rightarrow addr list"
  assumes succs-accesses:
  "accesses (succs p M) (node p)"

begin

declare succs-accesses[sepacc]

\end{verbatim}

2.3.1 Paths

\begin{verbatim}
inductive
  path :: "addr \Rightarrow addr list \Rightarrow addr \Rightarrow memory \Rightarrow bool"

where
  zeroI: "[ p = q; ps = [] ] \Rightarrow path p ps q M"

| stepI: "[ s \in set (succs p M); path s ps' q M; ps = p \# ps' ] \Rightarrow path p ps q M"
\end{verbatim}
lemma cycle-free-prefix-path:
"path p ps q M =⇒ path p (takeWhile (op ≠ q) ps) q M"
apply (induct rule: path.induct)
apply (auto intro: zeroI stepI)
done

lemma node-in-path-by-cycle-free-prefix:
" path p ps q M; x ∈ set ps =⇒ path p (takeWhile (op ≠ x) ps) x M"
apply (induct rule: path.induct)
apply (auto intro: zeroI stepI)
done

lemma node-in-path-by-cycle-free-postfix:
" path p ps q M; x ∈ set ps =⇒ ∃ qs. path p (ps @ qs) q M ∧ set qs ⊆ set ps ∧ x /∈ set qs"
apply (induct rule: path.induct)
apply (auto intro: zeroI stepI)
done

lemma eqv-inside-stop-cong: "eqv-inside A M M' = eqv-inside A M M'"
by simp

lemma path-empty:" path p [] q M = (p = q)"
by (auto elim: path.cases intro: zeroI)

lemma path-empty-contr:" P ⊆ Q; p ∈ P; set ps ∩ Q = {}; x /∈ Q =⇒ path p ps x M = False"
by (induct ps) (auto elim: path.cases)

lemma path-unfold:" p ≠ q =⇒ path p ps q M = (∃ qs s. path s qs q M ∧ s ∈ set (succs p M) ∧ ps = p # qs)"
by (intro: (auto elim: path.cases))

lemma path-single-start:" path s ps q M =⇒ ∀ s'. path s' ps q M =⇒ s = s'"
apply (erule path.induct)
apply clarsimp
apply clarify
apply (erule path.cases) back
apply hypsubst
apply clarify
apply hypsubst
apply clarify
done

lemma path-concat:" path p ps q M =⇒ path p (ps @ qs) r M"
apply (induct rule: path.induct)
apply simp
apply clarsimp
apply (erule path.cases)
apply simp
apply clarsimp
apply (drule meta-spec, drule(1) meta-mp)
apply (rule stepI)
apply simp-all
done
lemma path-take-front:
" \[
\text{path } p \hspace{1em} \text{ps} \hspace{1em} q \hspace{1em} M;
q' \in \text{set ps}
\Rightarrow \hspace{1em} \text{path } p \hspace{1em} \text{(takeWhile } (\text{op} \neq q') \hspace{1em} \text{ps}) \hspace{1em} q' \hspace{1em} M
\]
apply (induct rule: path.induct)
apply (auto intro: path.intros)
done

lemma path-cons-unfold:
" path \hspace{1em} d \hspace{1em} (d \neq rs) \hspace{1em} q \hspace{1em} M = (\exists s \in \text{set } \text{(succs } d \hspace{1em} M). \hspace{1em} \text{path } s \hspace{1em} rs \hspace{1em} q \hspace{1em} M)"
by (auto elim: path.cases intro: stepl)

declare accessor
" path \hspace{1em} p \hspace{1em} ps \hspace{1em} q " \" nodes (set ps)"
apply (rule accessesI)
apply (rule iffI)
apply (erule mp[rotated])
apply (erule path.induct)
simp
clarsimp
apply (drule mp)
apply (erule eqv-inside-sub, rule nodes-sub)
simp
clarsimp
apply (rule stepI)
apply (fastsep use-frames)
simp
apply simp
apply simp
apply simp
apply simp
by (rule_tac x="[]" in exI)
apply (auto intro: zeroI)
done

2.3.2 Reachability

Our definition of reachability includes a boundary set which enables graphs to be split at an arbitrary set of nodes, such that these nodes are effectively extracted (see [1]).

definition
" reachable \hspace{1em} P \hspace{1em} Q \hspace{1em} R \hspace{1em} M \equiv (R = \{q. \exists p \in P. \exists \text{ps. set ps} \cap Q = \{} \land q \notin Q \land \text{path } p \hspace{1em} \text{ps} \hspace{1em} q \hspace{1em} M \})"

lemma reachable-empty:
" P \subseteq Q \Rightarrow \text{reachable } P \hspace{1em} Q \hspace{1em} R \hspace{1em} M = (R = \{\})"
by (auto simp add: reachable-def path-empty-contr)

lemma reachable-zeroD:
" reachable \hspace{1em} P \hspace{1em} Q \hspace{1em} R \hspace{1em} M \Rightarrow (P - Q) \subseteq R"
by (force intro: zerol simp add: reachable-def)

lemma reachable-includes-direct:
" [ \text{reachable } P \hspace{1em} Q \hspace{1em} R \hspace{1em} M ] \Rightarrow P - Q \subseteq R"
apply (auto simp add: reachable-def intro: zerol)
apply (rule bexI[rotated], assumption)
apply (rule-tac x="[]" in exI)
apply (auto intro: zerol)
done
lemma reachable-includes-direct-spec:
"[[ reachable P Q R M; D ⊆ P; D ∩ Q = {} ]] ⟹ D ⊆ R"
by (auto dest: reachable-includes-direct)

lemma reachable-discard-Q:
"reachable P Q R M = reachable (P - Q) Q R M"
apply (simp add: reachable-def)
apply auto
apply (rule-tac x=p in bexI[rotated])
apply (force elim: path.cases)
apply fast
apply (rule-tac x=p in bexI[rotated])
apply (force elim: path.cases)
apply fast
done

lemma reachable-cong:
"[[ P = P'; Q = Q'; R = R' ]] ⟹ reachable P Q R M = reachable P' Q' R' M"
by simp

lemma reachable-closed:
"[[ reachable P Q R M; r ∈ R ]] ⟹ set (succs r M) ⊆ R ∪ Q"
apply (clarsimp simp add: reachable-def)
apply (rule bexI[rotated], assumption)
apply (rule-tac x="ps @ [r]" in exI)
apply simp
apply (blast intro: path-concat stepI zeroI)
done

lemma reachable-joinD:
"[[ reachable P Q R M; reachable P' Q R' M ]] ⟹ reachable (P ∪ P') Q (R ∪ R') M"
apply (clarsimp simp add: reachable-def)
apply auto
done

lemma set-sub-diff:
"[[ A ∩ B = {}; A ⊆ C ]] ⟹ A ⊆ C - B"
by auto

lemma simp-minus-Q:
"{q. ∃ p ∈ P. ∃ ps. set ps ∩ Q = {} ∧ q ∉ Q ∧ path p ps q M} - Q = {q. ∃ p ∈ P. ∃ ps. set ps ∩ Q = {} ∧ q ∉ Q ∧ path p ps q M}"
by auto

lemma list-at-end[rule-format]:
"ps ≠ [] ⟹ (∃ ps' r. ps = ps' @ [r])"
apply (induct ps)
apply simp
apply clarsimp
apply (case-tac "ps=[]")
apply auto
done

lemma list-at-end':
"ps = a ≠ b ⟹ (∃ ps' r. ps = ps' @ [r])"
by (auto intro: list-at-end)
lemma path-eqv-backwards: 
\"\( \gamma p P. [eqv-inside (nodes \{ q. \exists p \in P. \exists ps ps \cap Q = \{\} \land q \notin Q \land path p ps q M\})\] \\
M M'\):
\( p \in P; set ps \cap Q = \{\}; x \notin Q; path p ps x M\) \\
\( \implies path p ps x M\) \\
apply (induct ps)
apply (fast intro: path.intros elim: path.cases)
apply simp
apply clarsimp
apply (rename-tac p' ps')
apply (erule path.cases)
apply simp
apply clarsimp
apply (drule eqv-inside-commI)
back
apply (fastsep use-frames)
apply (drule-tac x="s" in meta-spec)
apply (drule-tac x="set (succs p' M)" in meta-spec)
apply simp
apply (erule meta-mp)
apply (erule eqv-inside-sub, rule nodes-sub)
apply clarify
apply (rule-tac x=p' in bexI)
apply (rule-tac x="p' \# ps" in exI)
apply simp
apply (erule eqv-inside-sub, rule nodes-sub)
apply clarify
apply (rule-tac x=p' in bexI)
apply (rule-tac x="p' \# ps" in exI)
apply simp
apply (rule stepI)
apply (assumption, assumption, simp, simp)
apply (rule stepI)
apply (assumption, assumption, simp)
done

This is the main result: the reachability predicate never touches the boundary set \( Q \), such that any objects contained in that set can be manipulated.

declare accessor
\"reachable P Q R\" \"nodes (R - Q)\"
apply (rule accessesI)
apply (unfold reachable-def)
apply (rule iffI)
apply clarify
apply (rule set-ext, rule iffI)
apply (clar simp add: simp-minus-Q)
apply (fastsep use-frames)
apply (rule-tac x=p in bexI, simp-all)
apply (rule-tac x=ps in exI)
apply simp
apply (clar simp add: simp-minus-Q)
apply (rule-tac x=p in bexI, simp-all)
apply (rule-tac x=ps in exI)
apply (fast intro: path-eqv-backwards)
apply (rule set-ext, rule iffI)
apply (clar simp add: simp-minus-Q)
apply (rule-tac x=p in bexI, simp-all)
apply (rule-tac x=ps in exI)
apply (fastsep use-frames)
apply simp
apply (clarsimp simp add: simp-minus-Q)
apply (rule-tac x=p in bexI, simp-all)
apply simp
apply (drule eqv-inside-commI)
apply (rule path-eqv-backwards)
apply assumption+
done

lemma reachable-disjoint-border:
"reachable P Q R M \implies R \cap Q = \{\}"
by (auto simp add: reachable-def)

lemma path-without-crossing-setD:
"\[ [ \exists d \in D. \exists qs rs. \text{path p qs d M} \land \text{path d (d # rs) q M} \land \text{set rs \cap D = \{\} \land} \\
\text{d \notin Q \land set qs \subseteq set ps \land set rs \subseteq set ps} ] \] \implies
apply (induct rule: path.induct)
apply simp
apply clarsimp
apply (case-tac "set ps' \cap D = \{\}"")
apply simp
apply (rule bexI[rotated], assumption)
apply (rule-tac x="ps" in exI)
apply fast
apply (simp del: disj-not1)
apply (case-tac "x \in D")
apply simp
apply (rule-tac x="x" in bexI)
apply (rule-tac x="[]" in exI)
apply simp
apply simp
apply fast
apply simp
apply simp
apply fast
apply simp

The following lemma now enables a set of nodes to be extracted from the graph defined reachability, and thus to be manipulated by the program.

lemma reachable-split:
"reachable P Q R M \implies \exists R1 R2. \text{reachable P (Q \cup D) R1 M} \land \
\text{reachable (D \cap R) Q R2 M} \land R = R1 \cup R2"
apply (simp add: reachable-def)
apply clarify
apply (rule iffI set-ext)+
apply (erule CollectE bexE exE conjE)+
apply (case-tac "set ps \cap D = \{\} \land x \notin D ")
apply clarify
apply (rule bexI[rotated], assumption)
apply (rule-tac x="ps" in exI)
apply fast
apply (simp del: disj-not1)
apply (case-tac "x \in D")
apply simp
apply (rule-tac x="x" in bexI)
apply (rule-tac x="[]" in exI)
apply simp
apply simp
apply simp
apply fast
apply simp
apply simp
apply fast
apply simp

12
It only remains to extract nodes from the graph such that the remainder of the graph is described in terms of the nodes’ successors. Such a formulation then mirrors the working of garbage collectors directly: at a very abstract level, a garbage collector repeatedly takes some reachable node that is not yet processed, removes it from the graph by processing it, and then continues with its successors. As shown in §3.8, the proofs can then follow the working of the algorithm.

definition
“Succs P M = (⋃p ∈ P. set (succs p M))”
declare accessor
“Succs P” ”nodes P”
by accesses

lemma Succs-singleton[simp]:
 “Succs {p} M = set (succs p M)”
by (simp add: Succs-def)

lemma Succs-empty[simp]:
 “Succs {} M = {}”
by (simp add: Succs-def)

lemma Succs-insert:
 “Succs (insert p Q) M = set (succs p M) ∪ Succs Q M”
by (simp add: Succs-def)

lemma Succs-Un:
 “Succs (A ∪ B) M = Succs A M ∪ Succs B M”
by (simp add: Succs-def)
lemma reachable-unfold-step:
"reachable P Q R M = (∃ R'. reachable (Succs (P - Q) M) (Q ∪ P) R' M ∧ R = R' ∪ (P - Q))"
apply (rule iffI)
apply simp add: reachable-def
apply (rule set-ext iffI)+
apply simp
apply (erule bexE exE conjE)+
apply (erule path.cases)
apply simp
apply clarsimp
apply (rename-tac p ps' q)
apply (case-tac "set ps' ∩ P = {""")
apply (rule-tac x=s in bex[rotated])
apply (simp add: Succs-def)
apply blast
apply blast
apply (erule_tac Q=Q and D=P in path-without-crossing-setD, simp, simp)
apply clarify
apply (clarsimp simp add: path-cons-unfold)
apply (rename-tac s')
apply (rule-tac x=s' in bex[rotated])
apply (simp add: Succs-def)
apply blast
apply blast
apply clarify
apply (erule UnE)
apply (clarsimp simp add: Succs-def)
apply (erule bex[rotated])
apply (rule-tac x="a ≠ ps" in exl)
apply simp
apply (blast intro: stepl)
apply clarify
apply (erule bex[rotated])
apply (rule-tac x="[]" in exl)
apply simp
apply clarsimp
apply simp
apply (erule UnE)
apply (clarsimp simp add: Succs-def)
apply (erule bex[rotated])
apply (rule-tac x="a ≠ ps" in exl)
apply simp
apply (blast intro: stepl)
apply clarify
apply (erule bex[rotated])
apply (rule-tac x="[]" in exl)
apply simp
apply simp
apply (erule bexE exE conjE)+
apply simp
apply (case-tac "x ∈ P")
apply simp
apply simp
apply simp
apply (simp add: Succs-def)
apply (erule-tac Q=Q and D=P in path-without-crossing-setD)
apply simp
apply (force elim: path.cases)
apply clarify
apply (erule-tac x=d in bex[rotated])
apply simp
apply (clarsimp simp add: path-cons-unfold)
apply blast
done
The central theorem follows: a graph can be split at any subset within the set of reachable nodes.

```
lemma reachable-exhibit-nodes:
"[ [ reachable P Q R M; D ⊆ R ] ] ➾ ( ∃ R1 R2. reachable P (Q ∪ D) R1 M ∧ reachable (Succs D M) (Q ∪ D) R2 M ∧ R = R1 ∪ R2 ∪ D )"
apply (subgoal-tac "D ∩ Q = {}")
pref 2
apply (drule reachable-disjoint-border)
apply blast
apply (drule-tac D="D" in reachable-split)
apply clarsimp
apply (rule exI conjI)+
apply assumption
apply (subst(asm) reachable-unfold-step)
back
apply (auto simp add: Int-absorb2 Diff-triv)
done
```

```
lemma reachable-exhibit-node:
"[ [ reachable P Q R M; n ∈ R ] ] ➾ ( ∃ R1 R2. reachable P (Q ∪ {n}) R1 M ∧ reachable (Succs {n} M) (Q ∪ {n}) R2 M ∧ R = R1 ∪ R2 ∪ {n} )"
by (rule reachable-exhibit-nodes) auto
```

3 Schorr-Waite Graph Marking

```
theory SchorrWaite
imports Verify0 ObjectGraphs LinkedList
begin

3.1 Connecting Lists to Objects

We re-use the library on linked lists as is from the case study on the L4 memory allocator [3, 2].
Here, the stack is modelled as a linked list. However, it must also be seen as a collection of objects
manipulated by the marking algorithm. The following lemmata make this connection precise.

```
lemma addr-off-le-imp-less:
"[ [ p ≤ p ⊕ n; 0 < n ] ] ➾ p < p ⊕ n"
by simp
```

```
lemma block-contains-start:
"[ [ block p n S; 0 < n ] ] ➾ p ∈ S"
by (auto simp add: block-def dest: addr-off-le-imp-less)
```

```
sublocale LinkedList < ns: node-set gctx node
apply unfold-locales
apply (rule node-wf-cover)
apply (rule node-contains-base, assumption)
done
```

```
lemma (in LinkedList) list-nodes-accesses-object-nodes:
"accesses (nodes p q s) M (ns.nodes (set s))"
apply (rule accessesI)
apply (rule iffI)
apply (erule mp[rotated])
apply (erule nodes.induct)
apply simp
apply clarsimp
apply (rule nodes.intros(2))
apply simp
```
```
apply (drule mp)
apply (erule eqv-inside-sub, sub)
apply (fastsep use-frames)
apply assumption
apply simp
apply (erule mp[rotated])
apply (erule nodes.induct)
apply simp
apply clarsimp
apply (rule nodes.intros(2))
apply simp
apply (erule nodes.induct)
apply simp
applyclarsimp
apply (rule nodes.intros(2))
apply simp
apply (drule mp)
apply (erule eqv-inside-sub, sub)
apply (fastsep use-frames)
apply assumption
apply simp
done

3.2 Data Structures

structdef "struct cell {
  bool m;
  bool c;
  struct cell *l;
  struct cell *r;
}
"

locale cell-defined =
fixes gctx
assumes cell-defined: "cell-known gctx"
begin

lemmas cell-defined-local[simp] =
same-staticD[where f = cell-known, OF cell-known-ignores-vars, THEN iffD2, OF - cell-defined]

declare cell-defined[simp]

definition "cell-cover ≡ λ p. typed-block gctx p (TStruct "cell")"

declare_cover
"cell-cover p"
apply (clarsimp simp add: wf-cover-def cell-cover-def)
apply wfcover-use
done

lemma cell-cover-unfold[sepunfold]:
"same-static ctx gctx ⇒ cell-cover p = typed-block ctx p (TStruct "cell")"
apply simp add: cell-cover-def
apply localize
done

definition "cell-succs ≡ λ p M. [ to-ptr (cell-l-rd gctx p M), to-ptr (cell-r-rd gctx p M) ]" 

declare_accessor
"cell-succs p" "cell-cover p"
apply (clarsimp simp)
apply accesses
done

lemma cell-succs-unfold:
"same-static ctx gctx ⇒ cell-succs p M = [ to-ptr (cell-l-rd ctx p M), to-ptr (cell-r-rd ctx p M) ]"
by (unfold cell-succs-def) localize
end
3.3 Instantiating the Object Graph Library

Note that this instantiation, using the provisions from §2.2, also sets up the automatic unfolding provers.

```
sublocale cell-defined < ns: split-heap-node-set gctx "cell-cover" "cell"
  apply unfold-locales
  apply (rule cell-cover-wf-cover)
  apply (simp-all add: cell-cover-def)
  apply (insert cell-defined)
  apply (simp add: typed-block-def block-contains-start cell-sz-of-ty)
  done

sublocale cell-defined < g: object-graph gctx "cell-cover" "cell-succs"
  apply unfold-locales
  apply (rule cell-succs-accesses)
  done
```

3.4 Instantiating the List Library

Note how the non-standard link structure in the algorithm’s stack is rendered as a singly linked list to access the re-usable formulation in the library [2].

```
context cell-defined
begin

definition
"stack-succ p M ≡
  if to-bool (cell-c-rd gctx p M)
  then to-ptr (cell-r-rd gctx p M)
  else to-ptr (cell-l-rd gctx p M)"

declare accessor
"stack-succ p" "cell-cover p"
apply (insert cell-defined)
apply (accesses unfold: cell-cover-def)
done

sublocale cell-defined < s: LinkedList gctc "cell-cover" stack-succ
  apply unfold-locales
  apply (rule cell-cover-wf-cover)
  prefer 2
  apply (rule stack-succ-accesses)
  apply (clarsimp simp add: cell-cover-def typed-block-def)
  apply (rule block-contains-start, assumption)
  apply (simp add: cell-sz-of-ty[OF cell-defined])
  done

context cell-defined
begin

definition
"stack ≡ λp s M. s.nodes p null s M"

declare accessor
"stack p s" "ns.nodes (set s)"
apply (simp add: stack-def)
apply (rule list-nodes-accesses-object-nodes)
done
```
We now introduce lemmata that model typical stack operations, that will be used in the main proof to keep the script aligned with the program operations.

**lemma pop-stackD:**

\[ \begin{align*}
\text{stack } p & \ S \ M; \ p \neq \text{null} \\
\implies & \exists ! S'. \ S = p \neq S' \land p \notin \text{set } S' \land \text{stack (stack-succ } p \ M) \ S' \ M
\end{align*} \]

by (auto simp add: stack-def stack-succ-def nodes-unfold-front)

**lemma pop-stack-leftE:**

\[ \begin{align*}
\text{stack } p & \ S \ M; \\
p \neq \text{null}; \neg \text{to-bool (cell-c-rd } gctx \ p \ M); \\
\land & \exists ! S'. \ \text{stack (to-ptr (cell-l-rd } gctx \ p \ M)) \ S' \ M; \\
S & = p \neq S' \land p \notin \text{set } S'
\end{align*} \]

by (auto dest: pop-stackD simp add: stack-succ-def)

**lemma pop-stack-rightE:**

\[ \begin{align*}
\text{stack } p & \ S \ M; \\
p \neq \text{null}; \text{to-bool (cell-c-rd } gctx \ p \ M); \\
\land & \exists ! S'. \ \text{stack (to-ptr (cell-r-rd } gctx \ p \ M)) \ S' \ M; \\
S & = p \neq S' \land p \notin \text{set } S'
\end{align*} \]

by (auto dest: pop-stackD simp add: stack-succ-def)

**lemma pop-stack-leftD:**

\[ \begin{align*}
\text{stack } p & \ S \ M; \\
p \neq \text{null}; \neg \text{to-bool (cell-c-rd } gctx \ p \ M) \\
\implies & \exists ! S'. \ \text{stack (to-ptr (cell-l-rd } gctx \ p \ M)) \ S' \ M \land \\
S & = p \neq S' \land p \notin \text{set } S'
\end{align*} \]

by (auto elim: pop-stack-leftE)

**lemma pop-stack-rightD:**

\[ \begin{align*}
\text{stack } p & \ S \ M; \\
p \neq \text{null; to-bool (cell-c-rd } gctx \ p \ M) \\
\implies & \exists ! S'. \ \text{stack (to-ptr (cell-r-rd } gctx \ p \ M)) \ S' \ M \land \\
S & = p \neq S' \land p \notin \text{set } S'
\end{align*} \]

by (auto elim: pop-stack-rightE)

**lemma stack-push:**

\[ \begin{align*}
\text{stack } p' & \ S \ M; p \notin \text{set } S; p \neq \text{null; stack-succ } p \ M = p' \\
\implies & \text{stack } p \ (p \neq S) \ M
\end{align*} \]

apply (simp add: stack-def)

apply (auto intro: nodes.intros(2))

done

**lemma push-stack-leftD:**

\[ \begin{align*}
\text{to-bool (cell-c-rd } gctx \ p \ M) \\
\implies & \text{stack } p \ (p \neq s) \ M'
\end{align*} \]

by (auto intro: nodes.intros(2) simp add: stack-def)

**lemma push-stack-rightD:**

\[ \begin{align*}
\text{to-bool (cell-r-rd } gctx \ p \ M) \\
\implies & \text{stack } p \ (p \neq s) \ M'
\end{align*} \]

by (auto intro: nodes.intros(2) simp add: stack-def)
lemma stack-singleD:
"[ stack p s M; stack p s' M ] \Longrightarrow s = s'"
by (auto simp add: stack-def dest: s.nodes-singleD)

lemma stack-null:
" stack null S M = (S = [])"
by (simp add: stack-def)

lemma head-in-stack:
"[ stack p s M; p \neq null ] \Longrightarrow p \in set s"
by (auto elim: nodes.cases simp add: stack-def)

lemma stack-succ-simps[simp]:
"[ ¬ to-bool (cell-c-rd ctx p M);
  same-static ctx gctx
 ] \Longrightarrow stack-succ p M = to-ptr (cell-l-rd ctx p M)"
"[ to-bool (cell-c-rd ctx p M);
  same-static ctx gctx
 ] \Longrightarrow stack-succ p M = to-ptr (cell-r-rd ctx p M)"
apply (simp-all add: stack-succ-def)
apply localize
apply simp
apply localize
apply simp
done

3.5 Reconstruction of the Original Pointer Structure

This predicate directly reflects the definitions from [4].

fun
stack-reco :: "addr => addr list => memory => memory => bool"
where
"stack-reco t [] M0 M = True"
| "stack-reco t (p #S) M0 M =
  ((if to-bool (cell-c-rd gctx p M)
    then (cell-l-rd gctx p M0 = cell-l-rd gctx p M ∧
      to-ptr (cell-r-rd gctx p M0) = t)
    else (cell-r-rd gctx p M0 = cell-r-rd gctx p M ∧
      to-ptr (cell-l-rd gctx p M0) = t)) ∧
  stack-reco p S M0 M)"

declare_accessor
"stack-reco t S M0" "ns.nodes (set S)"
apply (rule accessesI)
apply (induct S arbitrary: t)
apply simp
apply clarsimp
apply drule meta-spec, drule-tac x=a in meta-spec, drule meta-mp, erule eqv-inside-sub
apply (rule nodes-sub, blast)
apply (fastsep use-frames)
done
3.6 Auxiliary Definition for “Marked” Nodes

locale Schorr-Waite = cell-defined
begin

definition
"marked ctx p M ≡ to-bool (cell-m-rd ctx p M)"

declare accessor
"marked ctx p" "field-block ctx p «struct cell» "m"
apply accesses
apply ignores-vars
done

declare marked-def[THEN meta-eq-to-obj-eq, symmetric, lift]
declare field-block-lift[lift]
declare cell-defined-local[lift]

lemma marked-liftD:
"cell-m-rd ctx p M = to-Bool True ⇒ marked ctx p M"
by (simp add: marked-def)

lemma marked-redundant:
"[[ ∀ q ∈ S. marked ctx q M ]] ⇒ marked ctx p M = (p ∈ S ∨ marked ctx p M)"
by blast

lemma marked-redundant’:
"¬ marked ctx p M ⇒ marked ctx n M = (p ≠ n ∧ marked ctx n M)"
by blast

lemma marked-redundant’’:
"[[ marked ctx p M ]] ⇒ marked ctx q M = (p = q ∨ marked ctx q M)"
by blast

lemma sep-disj-cong[sepcong]:
"[[ P = P'; ¬ P ⇒ Q = Q' ]] ⇒ (P ∨ Q) = (P' ∨ Q')"
by blast

lemma split-ball:
"x ∈ A ⇒ (∀ y ∈ A. P y) = (P x ∧ (∀ y ∈ A - { x }. P y))"
by auto

lemma frame-refl’:
"[[ M = M'; wf-cover A; is-valid A ]] ⇒ frame A M' M"
by (auto intro: frame-refl)
lemma SCHORR-WAITE:
"∀ N S M0.
VERIFY
struct cell *root;
struct cell *t;
struct cell *p;
struct cell *q;
|= sm { M ▶ root || t || p || q || ns.nodes N ∧
  same-static ctx gctx ∧
  reachable { root } { null } N M ∧
  (∀ n ∈ N. ¬ marked gctx n M) ∧
  M = M0
} t = root;
p = nil;
[inv ∃ S.
  same-static ctx gctx ∧
  stack p S M ∧
  set S ⊆ N ∧
  reachable { root } { null } N M0 ∧
  (∀ q ∈ set S. marked gctx q M) ∧
  (t = null ∨ t ∈ N) ∧
  reachable
  (((t) ∪ set (map (λ n. to-ptr (cell-r-rd gctx n M)) S))
   { n. n = null ∨ n ∈ N ∧ marked gctx n M})
  { n ∈ N. ¬ marked gctx n M} M ∧
  (∀ p ∈ N. p /∈ set S → cell-succs p M = cell-succs p M0) ∧
  stack-reco t S M0 M ∧
  M ▶ root || t || p || q || ns.nodes N
}
while (p != nil || (t != nil && ! t → m)) {
  if (t == nil || t → m) {
    if (p→c) {
      q = t;
t = p;
p = p→r;
t→r = q;
    }
    else {
      q = t;
t = p→r;
p→r = p→l;
p→l = q;
p→c = true;
    }
  } else {
    q = p;
p = t;
t = t→l;
p→m = true;
p→l = q;
p→c = false;
  }
}
{| (∀ p ∈ N. marked ctx p M) ∧
  (∀ p ∈ N. cell-succs p M = cell-succs p M0)
}"
3.8 Proof

apply (rule all)+
interact
apply (drule-tac A="ns.nodes N" in frame-refl')
apply (rule nodes-wf-cover)
apply layout
proceed
step
step
Prove loop invariant holds initially
apply (simp(no-asm-simp))
apply (rule ex')
apply (rule conjI | assumption)+
apply (simp add: stack-null)
apply (case-tac "to-ptr (rdv ctx "root" M) = null")
apply (simp cong add: conj-cong)
apply (rule conjI)
apply (fastsep use-frames, assumption)
apply (rule conjI)
apply (drule reachable-includes-direct)
apply simp
apply (fastsep use-frames)
apply simp

The loop condition
interact
step
join_exec
apply clarsimp
proceed
step
step
join_exec
apply simp

Enable the inner check on p→c by showing p to be the top of stack
apply (subgoal-tac "to-ptr (rdv ctx "p" M) ≠ null ∧ to-ptr (rdv ctx "p" M) ∈ N")
prefer 2
apply (fast dest: head-in-stack)
apply clarsimp
apply localize
proceed
step
step

The 'pop' branch with t→c = True; simulate operation in proof
apply (erule pop-stack-rightE, simp, simp)
apply localize
apply clarsimp
apply localize
apply clarsimp
proceed
step
step
step
set metis-timeout: 50
step
step
The Swing Branch
Again, simulate the operation
apply (erule pop-stack-leftE, simp, simp)
apply localize
apply simp
apply clarsimp
apply localize
apply simp
proceed
Run through the body of the branch
step
step
step
step
step
step

The 'push' branch
join_exec
apply clarsimp
apply lift

The main point of reachability reasoning: extract one node from the graph

to be able to manipulate its content
apply (drule-tac n="to-ptr (rdv ctx "t" M)" in reachable-exhibit-node) back
apply simp
apply localize
apply simp
apply clarsimp
apply localize
apply (rename-tac S' R1 R2)
apply (simp add: insert-Collect)

Prove a few lemmata that are too hard during memory reasoning
apply (subgoal-tac "to-ptr (rdv ctx "p" M) \neq to-ptr (rdv ctx "t" M) \land to-ptr (rdv ctx "t" M) \notin S")
prefer 2
apply (rule conjI)
apply (case-tac "to-ptr (rdv ctx "p" M) = null")
apply simp
apply (fast dest: head-in-stack)
apply fast
apply (subgoal-tac "R1 \subseteq N \land R2 \subseteq N")
prefer 2
apply blast
apply clarify
Exhibit node t in the all quantifier to enable congruences
apply (drule-tac x1="to-ptr (rdv ctx "t" M)" in split-ball[THEN iffD1, rotated]) back
apply simp
apply clarsimp
Exhibit the l/r fields of t separately because they will be overwritten
apply (subst(asm) cell-succs-unfold, assumption) back
apply (subst(asm) cell-succs-unfold, assumption) back
apply (subst(asm) marked-redundant', assumption) back back back
apply clarsimp
proceed
Run through the body of the branch
step
step
step
step
step
step
step
step
step
Prove that the loop invariant is maintained in each case separately

split_exec

The pop branch
apply clarsimp
apply localize
Solve existence of the stack
apply (rule ex1_conjI assumption)+
Unmarked nodes remain reachable (by library + set reasoning)
apply (subst(asm) reachable-discard-Q) back
apply (subst(asm) insert-Diff1)
apply fast
apply (subst reachable-discard-Q)
apply simp
apply (subst(asm) reachable-discard-Q)
apply (subst(asm) insert-Diff1) back
apply simp
apply (subst disj-commute, rule disjCI)
apply (blast dest: head-in-stack)
apply simp

Reconstructability of the pointer structure
apply simp
apply clarify
apply (rename-tac M1 M2 M3 p')
apply (case-tac "p' = to-ptr (rdv ctx "p" M2)")
apply (simp add: cell-succs-unfold)
apply blast

The Swing branch
apply (rename-tac M1 M2 M3 M4)
apply clarsimp
Instantiate the stack
apply (rule ex1_conjI)+
apply (rule stack-push)
apply assumption+
apply (simp add: stack-succ-def)
apply localize
apply simp
apply simp
apply localize
apply simp
apply (rule conjI)
New t points to null or to some object
apply (drule-tac M=M0 in reachable-closed, assumption)
apply (simp add: cell-succs-unfold)
apply (rule conjI)
Reachability of unmarked nodes
apply (subst(asm) reachable-discard-Q) back
apply (subst(asm) insert-Diff1)
apply blast
apply (subst insert-commute)
apply (subst reachable-discard-Q)
apply (subst insert-Diff1)
apply simp
apply (subst disj-commute, rule disjCI)
apply (blast dest: head-in-stack)
apply localize

Remaining equality is there except for technical embedding
apply (subst to-ptr-eq-inj[symmetric, rotated -1])
apply (rule tyval-of-rdv, rule sym, assumption)
apply simp
apply simp
The push branch
apply (rename-tac M1 M2 M3 M4)
apply clarsimp
apply (drule marked-liftD)

Instantiate the stack
apply (rule ex!’ conjI)+
apply (rule stack-push, assumption)
apply simp
apply simp
apply (simp add: stack-succ-def)
apply localize
apply simp

Proceed with remaining conditions
apply simp
apply localize
apply simp
apply (rule conjI)
The new t is null or a proper object
apply (drule-tac M=M0 in reachable-closed, assumption)
apply (simp add: cell-succs-unfold)
apply (rule conjI)
Reachability of unmarked nodes
apply (subst(asm) reachable-discard-Q) back
apply (subst(asm) insert-Diff1)
apply blast
apply (subst(asm) cell-succs-unfold, assumption)
apply (drule(1) reachable-joinD)
apply (thin-tac “reachable ?P ?Q R2 ?M”) 
apply (subst(asm) reachable-discard-Q) back
Perform same modifications on the conclusion
apply (subst reachable-discard-Q)
apply (rule reachable-cong[THEN iffD1, rotated -1], assumption)
apply simp
Now the three set arguments of reachability are the same
apply blast
apply blast
apply (rule-tac x="to-ptr (rdv ctx “t” M2)” in insert-ident[THEN iffD1, standard])
apply (drule-tac R="R1 ∪ R2” in reachable-disjoint-border)
apply fast
apply fast
apply (rule trans, drule sym, assumption)
apply fast

Remaining equality is there except for technical embedding
apply (subst to-ptr-eq-inj[symmetric, rotated -1])
apply (rule tyval-of-cell-r-rd, simp)
apply simp
apply simp

After the loop
step
The overall loop condition expresses “done”
join_exec
apply clarsimp
proceed
step
apply (clarsimp simp add: stack-null)
apply (drule reachable-empty[THEN iffD1, rotated])
apply localize
apply blast
apply localize
apply blast
step
done
References


