Verification of the Deutsch-Schorr-Waite Graph Marking Algorithm using Data Refinement

Viorel Preoteasa and Ralph-Johan Back

July 1, 2010

Abstract

The verification of the Deutsch-Schorr-Waite graph marking algorithm is used as a benchmark in many formalizations of pointer programs. The main purpose of this mechanization is to show how data refinement of invariant based programs can be used in verifying practical algorithms. The verification starts with an abstract algorithm working on a graph given by a relation *next* on nodes. Gradually the abstract program is refined into Deutsch-Schorr-Waite graph marking algorithm where only one bit per graph node of additional memory is used for marking.

Contents

1 Introduction

The verification of the Deutsch-Schorr-Waite (DSW) [14, 10] graph marking algorithm is used as a benchmark in many formalizations of pointer programs [11, 1]. The main purpose of this mechanization is to show how data refinement [12] of invariant based programs [3, 4, 5, 6] can be used in verifying practical algorithms.

The DSW algorithm marks all nodes in a graph that are reachable from a *root* node. The marking is achieved using only one extra bit of memory for every node. The graph is given by two pointer functions, *left* and *right*, which for any given node return its left and right successors, respectively. While marking, the left and right functions are altered to represent a stack that describes the path from the root to the current node in the graph. On completion the original graph structure is restored. We construct the DSW algorithm by a sequence of three successive data refinement steps. One step in these refinements is a generalization of the DSW algorithm to an algorithm which marks a graph given by a family of pointer functions instead of left and right only.

Invariant based programming is an approach to construct correct programs where we start by identifying all basic situations (pre- and post-conditions, and loop invariants) that could arise during the execution of the algorithm. These situations are determined and described before any code is written. After that, we identify the transitions between the situations, which together determine the flow of control in the program. The transitions are verified at the same time as they are constructed. The correctness of the program is thus established as part of the construction process.

Data refinement [9, 2, 7, 8] is a technique of building correct programs working on concrete data structures as refinements of more abstract programs working on abstract data structures. The correctness of the final program follows from the correctness of the abstract program and from the correctness of the data refinement.

Both the semantics and the data refinement of invariant based programs were formalized in [13], and this verification is based on them.

We use a simple model of pointers where addresses (pointers, nodes) are the elements of a set and pointer fields are global pointer functions from addresses to addresses. Pointer updates ($x.\mathit{left} := y$) are done by modifying the global pointer function $\mathit{left} := \mathit{left}(x := y)$. Because of the nature of the marking algorithm where no allocation and disposal of memory are needed we do not treat these operations.

A number of Isabelle techniques are used here. The class mechanism is used for extending the complete lattice theories as well as for introducing well founded and transitive relations. The polimorphism is used for the state of the computation. In [13] the state of computation was introduced as a type variable, or even more generaly, state predicates were introduced as elements of a complete (boolean) lattice. Here the state of the computation is instantiated with various tuples ranging from the abstract data in the first algorithm to the concrete data in the final refinement. The locale mechanism of Isabelle is used to introduce the specification variables and their invariants. These specification variables are used for example to prove that the main variables are restored to their initial values when the algorithm terminates. The locale extension and partial instantiation mechanisms turn out to be also very useful in the data refinements of DSW. We start with a locale which fixes the abstract graph as a relation *next* on nodes. This locale is first partially interpreted into a locale which replaces *next* by a union of a family of pointer functions. In the final refinement step the locale of the pointer functions is interpreted into a locale with only two pointer functions, *left* and *right*.

2 Address Graph

theory *Graph*

imports *Main*

begin

This theory introduces the graph to be marked as a relation next on nodes (addresses). We assume that we have a special node nil (the null address). We have a node root from which we start marking the graph. We also assume that nil is not related by next to any node and any node is not related by next to nil.

locale *node* =
  fixes *nil* :: *'node*
  fixes *root* :: *'node*

locale *graph* = *node* +
  fixes *next* :: (*'node* × *'node*) *set*
  assumes *next-not-nil-left*: (!! *x* . (*nil*, *x*) ∉ *next*)
  assumes *next-not-nil-right*: (!! *x* . (*x*, *nil*) ∉ *next*)
begin

On lists of nodes we introduce two operations similar to existing hd and tl for getting the head and the tail of a list. The new function head applied to a nonempty list returns the head of the list, and it reurns nil when applied to the empty list. The function tail returns the tail of the list when applied to a non-empty list, and it returns the empty list otherwise.

  definition
    head S ≡ (*if S* = [] *then nil else* (*hd S*))

  definition
    tail (*S*::*'a list*) ≡ (*if S* = [] *then* [] *else* (*tl S*))

  lemma [*simp*]: ((*nil*, *x*) ∈ *next*) = *False*
    by (*simp add*: *next-not-nil-left*)

  lemma [*simp*]: ((*x*, *nil*) ∈ *next*) = *False*
    by (*simp add*: *next-not-nil-right*)

  theorem *head-not-nil* [*simp*]:
    (*head S* ≠ *nil*) = (*head S* = *hd S* ∧ *tail S* = *tl S* ∧ *hd S* ≠ *nil* ∧ *S* ≠ [])
    by (*simp add*: *head-def tail-def*)

  theorem *nonempty-head* [*simp*]:
    head (*x* # *S*) = *x*
    by (*simp add*: *head-def*)

  theorem *nonempty-tail* [*simp*]:
    tail (*x* # *S*) = *S*
    by (*simp add*: *tail-def*)

  end

end

3 Marking Using a Set

theory *SetMark*

imports *Graph ../DataRefinementIBP/DataRefinement*

begin

We construct in this theory a diagram which computes all reachable nodes from a given root node in a graph. The graph is defined in the theory Graph and is given by a relation *next* on the nodes of the graph.

The diagram has only three ordered situation ($init > loop > final$). The termination variant is a pair of a situation and a natural number with the lexicographic ordering. The idea of this ordering is that we can go from a bigger situation to a smaller one, however if we stay in the same situation the second component of the variant must decrease.

The idea of the algorithm is that it starts with a set X containing the root element and the root is marked. As long as X is not empty, if $x \in X$ and y is an unmarked sucessor of x we add y to X. If $x \in X$ has no unmarked sucessors it is removed from X. The algorithm terminates when X is empty.

datatype $I = init \mid loop \mid final$

declare *I.split* [*split*]

instantiation I :: *well-founded-transitive*
begin

definition
less-I-def: $i < j \equiv (j = init \wedge (i = loop \vee i = final)) \vee (j = loop \wedge i = final)$

definition
less-eq-I-def: $(i{::}I) \leq (j{::}I) \equiv i = j \vee i < j$

instance proof
fix $x\ y\ z :: I$
  assume $x < y$ **and** $y < z$ **then show** $x < z$
    apply (*simp add*: *less-I-def*)
    by *auto*
  next
fix $x\ y :: I$
  show $x \leq y <-> x = y \vee x < y$
    by (*simp add*: *less-eq-I-def*)
  next
fix P **fix** $a{::}I$
  show $(\forall\ x.\ (\forall\ y.\ y < x \longrightarrow P\ y) \longrightarrow P\ x) \Longrightarrow P\ a$
    apply (*case-tac P final*)
    apply (*case-tac P loop*)
    apply (*simp-all add*: *less-I-def*)
    by *blast*
next

qed (*simp*)

end

definition (**in** *graph*)
  $reach\ x \equiv \{y\ .\ (x,\ y) \in next^* \wedge y \neq nil\}$

theorem (**in** *graph*) *reach-nil* [*simp*]: $reach\ nil = \{\}$
  apply (*simp add*: *reach-def*, *safe*)
  apply (*drule rtrancl-induct*)
  by *auto*

theorem (**in** *graph*) *reach-next*: $b \in reach\ a \Longrightarrow (b,\ c) \in next \Longrightarrow c \in reach\ a$
  apply (*simp add*: *reach-def*)
  by *auto*

definition (**in** *graph*)
  $path\ S\ mrk \equiv \{x\ .\ (\exists\ s\ .\ s \in S \wedge (s,\ x) \in next\ O\ (next \cap ((-mrk)\times(-mrk)))^*$
$)\}$

The set *path S mrk* contains all reachable nodes from S along paths with unmarked nodes.

lemma (**in** *graph*) *trascl-less*: $x \neq y \Longrightarrow (a,\ x) \in R^* \Longrightarrow$
    $((a,x) \in (R \cap (-\{y\})\times(-\{y\}))^* \vee\ (y,x) \in R\ O\ (R \cap (-\{y\})\times(-\{y\}))^*\ )$
  apply (*drule-tac*
    $b = x$ **and** $a = a$ **and** $r = R$ **and**
    $P = \lambda\ x.\ (x \neq y \longrightarrow ((a,x) \in (R \cap (-\{y\})\times(-\{y\}))^* \vee (y,x) \in R\ O\ (R \cap$
$(-\{y\})\times(-\{y\}))^*\ ))$
    in *rtrancl-induct*)
  apply *auto*
  apply (*case-tac* $ya = y$)
  apply *auto*
  apply (*rule-tac* $a = a$ **and** $b = ya$ **and** $c = z$ **and** $r = R \cap ((UNIV\ -\ \{y\})\ \times$
$(UNIV\ -\ \{y\}))$ **in** *rtrancl-trans*)
  apply *auto*
  apply (*case-tac* $za = y$)
  apply *auto*
  apply (*drule-tac* $a = ya$ **and** $b = za$ **and** $c = z$ **and** $r = (R \cap (UNIV\ -\ \{y\})$
$\times\ (UNIV\ -\ \{y\}))$ **in** *rtrancl-trans*)
  by *auto*

lemma (**in** *graph*) *add-set* [*simp*]: $x \neq y \Longrightarrow x \in path\ S\ mrk \Longrightarrow x \in path\ (insert$
$y\ S)\ (insert\ y\ mrk)$
  apply (*simp add*: *path-def*)
  apply *clarify*
  apply (*drule-tac* $x = x$ **and** $y = y$ **and** $a = ya$ **and** $R = next \cap (-\ mrk) \times (-$
$mrk)$ **in** *trascl-less*)
  apply *simp-all*
  apply (*case-tac* $(ya,\ x) \in (next \cap (-\ mrk) \times -\ mrk \cap (-\ \{y\}) \times -\ \{y\})^*$)

apply (*rule-tac* $x = xa$ **in** *exI*)
apply *simp-all*
apply (*simp add*: *rel-comp-def*)
apply (*rule-tac* $x = ya$ **in** *exI*)
apply *simp*
apply (*case-tac* (*next* $\cap$ ($-$ *mrk*) $\times$ $-$ *mrk* $\cap$ ($-$ $\{y\}$) $\times$ $-$ $\{y\}$) $=$ (*next* $\cap$ ($-$ *insert y mrk*) $\times$ $-$ *insert y mrk*))
apply *simp-all*
apply *safe*
apply *simp-all*
apply (*rule-tac* $x = y$ **in** *exI*)
apply *simp*
apply (*simp add*: *rel-comp-def*)
apply (*rule-tac* $x = yaa$ **in** *exI*)
apply *simp*
apply (*case-tac* (*next* $\cap$ ($-$ *mrk*) $\times$ $-$ *mrk* $\cap$ ($-$ $\{y\}$) $\times$ $-$ $\{y\}$) $=$ (*next* $\cap$ ($-$ *insert y mrk*) $\times$ $-$ *insert y mrk*))
apply *simp-all*
by *auto*

lemma (**in** *graph*) *add-set2*: $x \in$ *path S mrk* $\Longrightarrow$ $x \notin$ *path* (*insert y S*) (*insert y mrk*) $\Longrightarrow$ $x = y$
apply (*case-tac* $x \neq y$)
apply (*frule add-set*)
by *simp-all*

lemma (**in** *graph*) *del-stack* [*simp*]: ($\forall$ y . (t, y) $\in$ *next* $\longrightarrow$ $y \in$ *mrk*) $\Longrightarrow$ $x \notin$ *mrk* $\Longrightarrow$ $x \in$ *path S mrk* $\Longrightarrow$ $x \in$ *path* (S $-$ $\{t\}$) *mrk*
apply (*simp add*: *path-def*)
apply *clarify*
apply (*rule-tac* $x = xa$ **in** *exI*)
apply (*case-tac* $x = y$)
apply *auto*
apply (*drule-tac* $a = y$ **and** $b = x$ **and** $R =$ (*next* $\cap$ ($-$ *mrk*) $\times$ $-$ *mrk*) **in** *rtranclD*)
apply *safe*
apply (*drule-tac* $x = y$ **and** $y = x$ **in** *tranclD*)
by *auto*

lemma (**in** *graph*) *init-set* [*simp*]: $x \in$ *reach root* $\Longrightarrow$ $x \neq$ *root* $\Longrightarrow$ $x \in$ *path* $\{root\}$ $\{root\}$
apply (*simp add*: *reach-def path-def*)
apply (*case-tac root* $\neq x$)
apply (*drule-tac* $a =$ *root* **and** $x = x$ **and** $y =$ *root* **and** $R =$ *next* **in** *trascl-less*)
apply *simp-all*
apply *safe*
apply (*drule-tac* $a =$ *root* **and** $b = x$ **and** $R =$ (*next* $\cap$ (*UNIV* $-$ $\{root\}$) $\times$ (*UNIV* $-$ $\{root\}$)) **in** *rtranclD*)
apply *safe*

apply (*drule-tac* $x = root$ **and** $y = x$ **in** *tranclD*)
by *auto*

lemma (**in** *graph*) *init-set2*: $x \in reach\ root \Longrightarrow x \notin path\ \{root\}\ \{root\} \Longrightarrow x = root$
apply (*case-tac* $root \neq x$)
apply (*drule init-set*)
by *simp-all*

3.1 Transitions

definition (**in** *graph*)
$Q1 \equiv \lambda\ (X::('node\ set),\ mrk::('node\ set))\ .\ \{\ (X'::('node\ set),\ mrk')\ .\ (root::'node) = nil \wedge X' = \{\} \wedge mrk' = mrk\}$

definition (**in** *graph*)
$Q2 \equiv \lambda\ (X::('node\ set),\ mrk::('node\ set))\ .\ \{\ (X',\ mrk')\ .\ (root::'node) \neq nil \wedge X' = \{root::'node\} \wedge mrk' = \{root::'node\}\}$

definition (**in** *graph*)
$Q3 \equiv \lambda\ (X,\ mrk)\ .\ \{\ (X',\ mrk')\ .$
$(\exists\ x \in X\ .\ \exists\ y\ .\ (x,\ y) \in next \wedge y \notin mrk \wedge X' = X \cup \{y\} \wedge mrk' = mrk \cup \{y\})\}$

definition (**in** *graph*)
$Q4 \equiv \lambda\ (X,\ mrk)\ .\ \{\ (X',\ mrk')\ .$
$(\exists\ x \in X\ .\ (\forall\ y\ .\ (x,\ y) \in next \longrightarrow y \in mrk) \wedge X' = X - \{x\} \wedge mrk' = mrk)\}$

definition (**in** *graph*)
$Q5 \equiv \lambda\ (X::('node\ set),\ mrk::('node\ set))\ .\ \{\ (X'::('node\ set),\ mrk')\ .\ X = \{\} \wedge mrk = mrk'\}$

3.2 Invariants

definition (**in** *graph*)
$Loop \equiv \{\ (X,\ mrk)\ .$
$finite\ (-mrk) \wedge finite\ X \wedge X \subseteq mrk \wedge$
$mrk \subseteq reach\ root \wedge reach\ root \cap -mrk \subseteq path\ X\ mrk\}$

definition
$trm \equiv \lambda\ (X,\ mrk)\ .\ 2 * card\ (-mrk) + card\ X$

definition
$term\text{-}eq\ t\ w = \{s\ .\ t\ s = w\}$

definition
$term\text{-}less\ t\ w = \{s\ .\ t\ s < w\}$

lemma *union-term-eq*[*simp*]: $(\bigcup\ w\ .\ term\text{-}eq\ t\ w) = UNIV$

apply (*simp add*: *term-eq-def*)
by *auto*

lemma *union-less-term-eq*[*simp*]: $(\bigcup v{\in}\{v.\ v < w\}.\ \textit{term-eq}\ t\ v) = \textit{term-less}\ t\ w$
apply (*simp add*: *term-eq-def term-less-def*)
by *auto*

definition (**in** *graph*)
$\textit{Init} \equiv \{\ (X::('node\ set),\ mrk::('node\ set))\ .\ \textit{finite}\ (-mrk) \wedge mrk = \{\}\}$

definition (**in** *graph*)
$\textit{Final} \equiv \{\ (X::('node\ set),\ mrk::('node\ set))\ .\ mrk = \textit{reach root}\}$

definition (**in** *graph*)
$\textit{SetMarkInv}\ i = (\textit{case}\ i\ \textit{of}$
$\quad I.\textit{init}\ \Rightarrow \textit{Init}\ |$
$\quad I.\textit{loop}\ \Rightarrow \textit{Loop}\ |$
$\quad I.\textit{final} \Rightarrow \textit{Final})$

definition (**in** *graph*)
$\textit{SetMarkInvFinal}\ i = (\textit{case}\ i\ \textit{of}$
$\quad I.\textit{final}\ \Rightarrow \textit{Final}\ |$
$\quad \text{-}\ \Rightarrow \{\})$

definition (**in** *graph*) [*simp*]:
$\textit{SetMarkInvTerm}\ w\ i = (\textit{case}\ i\ \textit{of}$
$\quad I.\textit{init}\ \Rightarrow \textit{Init}\ |$
$\quad I.\textit{loop}\ \Rightarrow \textit{Loop} \cap \{s\ .\ \textit{trm}\ s = w\}\ |$
$\quad I.\textit{final} \Rightarrow \textit{Final})$

definition (**in** *graph*)
$\textit{SetMark-rel} \equiv \lambda\ (i,\ j)\ .\ (\textit{case}\ (i,\ j)\ \textit{of}$
$\quad (I.\textit{init},\ I.\textit{loop})\ \Rightarrow Q1 \sqcup Q2\ |$
$\quad (I.\textit{loop},\ I.\textit{loop})\ \Rightarrow Q3 \sqcup Q4\ |$
$\quad (I.\textit{loop},\ I.\textit{final}) \Rightarrow Q5\ |$
$\quad \text{-} \Rightarrow \bot)$

3.3 Diagram

definition (**in** *graph*)
$\textit{SetMark} \equiv \lambda\ (i,\ j)\ .\ (\textit{case}\ (i,\ j)\ \textit{of}$
$\quad (I.\textit{init},\ I.\textit{loop})\ \Rightarrow (\textit{demonic}\ Q1) \sqcap (\textit{demonic}\ Q2)\ |$
$\quad (I.\textit{loop},\ I.\textit{loop})\ \Rightarrow (\textit{demonic}\ Q3) \sqcap (\textit{demonic}\ Q4)\ |$
$\quad (I.\textit{loop},\ I.\textit{final}) \Rightarrow \textit{demonic}\ Q5\ |$
$\quad \text{-} \Rightarrow \textit{top})$

lemma (**in** *graph*) *dgr-dmonic-SetMark* [*simp*]:
dgr-demonic SetMark-rel = *SetMark*
by (*simp add*: *expand-fun-eq SetMark-def dgr-demonic-def SetMark-rel-def demonic-sup-inf*)

lemma (**in** *graph*) *SetMark-dmono* [*simp*]:
dmono SetMark
apply (*unfold dmono-def SetMark-def*)
by *simp*

3.4 Correctness of the transitions

lemma (**in** *graph*) *init-loop-1*[*simp*]: $\models$ *Init* {| *demonic Q1* |} *Loop*
apply (*unfold hoare-demonic Init-def Q1-def Loop-def*)
by *auto*

lemma (**in** *graph*) *init-loop-2*[*simp*]: $\models$ *Init* {| *demonic Q2* |} *Loop*
apply (*simp add*: *hoare-demonic Init-def Q2-def Loop-def*)
apply *auto*
apply (*simp-all add*: *reach-def*)
apply (*rule init-set2*)
by (*simp-all add*: *reach-def*)

lemma (**in** *graph*) *loop-loop-1*[*simp*]: $\models$ (*Loop* $\cap$ {*s* . *trm s* = *w*}) {| *demonic Q3* |} (*Loop* $\cap$ {*s*. *trm s* < *w*})
apply (*simp add*: *hoare-demonic Q3-def Loop-def trm-def*)
apply *safe*
apply (*simp-all*)
apply (*simp-all add*: *reach-def subset-eq*)
apply *safe*
apply *simp-all*
apply (*rule rtrancl-into-rtrancl*)
apply (*simp-all add*: *Int-def*)
apply (*rule add-set2*)
apply *simp-all*
apply (*case-tac card* $(-b) > 0$)
by *auto*

lemma (**in** *graph*) *loop-loop-2*[*simp*]: $\models$ (*Loop* $\cap$ {*s* . *trm s* = *w*}) {| *demonic Q4* |} (*Loop* $\cap$ {*s*. *trm s* < *w*})
apply (*simp add*: *hoare-demonic Q4-def Loop-def trm-def*)
apply *auto*
apply (*case-tac card a* > *0*)
by *auto*

lemma (**in** *graph*) *loop-final*[*simp*]: $\models$ (*Loop* $\cap$ {*s* . *trm s* = *w*}) {| *demonic Q5* |} *Final*
apply (*simp add*: *hoare-demonic Q5-def Loop-def Final-def subset-eq Int-def path-def*)
by *auto*

lemma *union-term-w*[*simp*]: $(\bigcup w.\ \{s.\ t\ s = w\}) = UNIV$
  by *auto*

lemma *union-less-term-w*[*simp*]: $(\bigcup v\in\{v.\ v < w\}.\ \{s.\ t\ s = v\}) = \{s\ .\ t\ s < w\}$
  by *auto*

lemma *sup-union*[*simp*]: $SUP\ A\ i = (\bigcup\ w\ .\ A\ w\ i)$
  by (*simp-all add*: *SUP-fun-eq*)

lemma *empty-pred-false*[*simp*]: $\{\}\ a = False$
  by *blast*

lemma *forall-simp*[*simp*]: $(!a\ b.\ \forall\ x\in A\ .\ (a = (t\ x)) \longrightarrow (h\ x) \vee b \neq u\ x) = (\forall\ x\in A\ .\ h\ x)$
  apply *safe*
  by *auto*

lemma *forall-simp2*[*simp*]: $(!a\ b.\ \forall\ x\in A\ .\ !y\ .\ (a = t\ x\ y) \longrightarrow (h\ x\ y) \longrightarrow (g\ x\ y) \vee b \neq u\ x\ y) = (\forall\ x\in A\ .\ !\ y\ .\ h\ x\ y \longrightarrow g\ x\ y\ )$
  apply *safe*
  by *auto*

3.5 Diagram correctness

The termination ordering for the *SetMark* diagram is the lexicographic ordering on pairs (i, n) where $i \in I$ and $n \in nat$.

interpretation *DiagramTermination* $\lambda\ (n{::}nat)\ (i :: I)\ .\ (i, n)$
  done

theorem (**in** *graph*) *SetMark-correct*:
  $\models$ *SetMarkInv* {|*pt SetMark*|} *SetMarkInvFinal*
  apply (*rule-tac X* = *SetMarkInvTerm* **in** *hoare-diagram3*)
  apply *simp*
  apply *safe*
  apply (*simp-all add*: *SetMark-def SUP-L-P-def*
    less-pair-def less-I-def not-grd-dgr hoare-choice SetMarkInv-def SetMarkInvFinal-def
neg-fun-pred SetMark-def)
  apply *auto*
  apply (*case-tac x*)
  apply *simp-all*
  apply (*drule-tac x=I.loop* **in** *spec*)
  apply (*simp add*: *Q1-def Q2-def*)
  apply *auto*
  apply (*frule-tac x=I.loop* **in** *spec*)
  apply (*drule-tac x=I.final* **in** *spec*)
  apply (*simp add*: *Q3-def Q4-def Q5-def*)
  by *auto*

theorem (**in** *graph*) *SetMark-correct1* [*simp*]:
Hoare-dgr SetMarkInv SetMark (*SetMarkInv* $\sqcap$ (− *grd* (*step SetMark*)))
apply (*simp add*: *Hoare-dgr-def*)
apply (*rule-tac x* = *SetMarkInvTerm* **in** *exI*)
apply *simp*
apply *safe*
apply (*simp-all add*: *SetMark-def SUP-L-P-def*
less-pair-def less-I-def not-grd-dgr hoare-choice SetMarkInv-def SetMarkInvFinal-def
neg-fun-pred SetMark-def)

apply (*simp-all add*: *expand-fun-eq*)
apply *safe*
apply (*unfold SetMarkInv-def*)
apply *auto*
apply (*case-tac x*)
apply *simp-all*
apply (*case-tac x*)
by *simp-all*

theorem (**in** *graph*) *stack-not-nil* [*simp*]:
$(mrk, S) \in Loop \Longrightarrow x \in S \Longrightarrow x \neq nil$
apply (*simp add*: *Loop-def reach-def*)
by *auto*

end

4 Marking Using a Stack

theory *StackMark*

imports *SetMark DataRefinement*

begin

In this theory we refine the set marking diagram to a diagram in which the set is replaced by a list (stack). Iniatially the list contains the root element and as long as the list is nonempty and the top of the list has an unmarked successor y, then y is added to the top of the list. If the top does not have unmarked sucessors, it is removed from the list. The diagram terminates when the list is empty.

The data refinement relation of the two diagrams is true if the list has distinct elements and the elements of the list and the set are the same.

consts
dist:: *'a list* ⇒ *bool*

primrec

$dist\ [] \quad = True$
$dist\ (a\ \#\ L) = (\neg\ a\ mem\ L \wedge dist\ L)$

4.1 Transitions

definition (**in** *graph*)
$Q1'\ s \equiv let\ (stk::('node\ list),\ mrk::('node\ set)) = s\ in\ \{(stk'::('node\ list),\ mrk')$
.
$root = nil \wedge stk' = [] \wedge mrk' = mrk\}$

definition (**in** *graph*)
$Q2'\ s \equiv let\ (stk::('node\ list),\ mrk::('node\ set)) = s\ in\ \{(stk',\ mrk')\ .\ root \neq nil$
$\wedge\ stk' = [root] \wedge mrk' = mrk \cup \{root\}\}$

definition (**in** *graph*)
$Q3'\ s \equiv let\ (stk,\ mrk) = s\ in\ \{(stk',\ mrk')\ .\ \ stk \neq [] \wedge (\exists\ y\ .\ (hd\ stk,\ y) \in$
$next\ \wedge$
$y \notin mrk \wedge stk' = y\ \#\ stk \wedge mrk' = mrk \cup \{y\})\}$

definition (**in** *graph*)
$Q4'\ s \equiv let\ (stk,\ mrk) = s\ in\ \{(stk',\ mrk')\ .\ stk \neq [] \wedge$
$(\forall\ y\ .\ (hd\ stk,\ y) \in next \longrightarrow y \in mrk) \wedge stk' = tl\ stk \wedge mrk' = mrk\}$

definition
$Q5'\ s \equiv let\ (stk,\ mrk) = s\ in\ \{(stk',\ mrk')\ .\ stk = [] \wedge mrk' = mrk\}$

4.2 Invariants

definition
$Init' \equiv UNIV$

definition
$Loop' \equiv \{\ (stk,\ mrk)\ .\ dist\ stk\}$

definition
$Final' \equiv UNIV$

definition [*simp*]:
$StackMarkInv\ i = (case\ i\ of$
$I.init\ \Rightarrow Init'\ |$
$I.loop\ \Rightarrow Loop'\ |$
$I.final \Rightarrow Final')$

4.3 Data refinement relations

definition
$R1 \equiv \lambda\ (stk,\ mrk)\ .\ \{(X,\ mrk')\ .\ mrk' = mrk\}$

definition

$R2 \equiv \lambda\ (stk,\ mrk)\ .\ \{(X,\ mrk')\ .\ X = \{x\ .\ x\ mem\ stk\} \wedge (stk,\ mrk) \in Loop' \wedge mrk' = mrk\}$

definition [*simp*]:
R i = (*case i of*
I.init ⇒ *R1* |
I.loop ⇒ *R2* |
I.final ⇒ *R1*)

definition (**in** *graph*)
StackMark-rel = (λ (*i*, *j*) . (*case* (*i*, *j*) *of*
(*I.init*, *I.loop*) ⇒ *Q1′* ⊔ *Q2′* |
(*I.loop*, *I.loop*) ⇒ *Q3′* ⊔ *Q4′* |
(*I.loop*, *I.final*) ⇒ *Q5′* |
\- ⇒ ⊥))

4.4 Data refinement of the transitions

theorem (**in** *graph*) *init-nil* [*simp*]:
DataRefinement Init Q1 R1 R2 (*demonic Q1′*)
by (*simp add*: *DataRefinement-def hoare-demonic Q1′-def Init-def R1-def Loop′-def R1-def R2-def Q1-def angelic-def subset-eq*)

theorem (**in** *graph*) *init-root* [*simp*]:
DataRefinement Init Q2 R1 R2 (*demonic Q2′*)
by (*auto simp*: *DataRefinement-def hoare-demonic Q2′-def Init-def Loop′-def R1-def R2-def Q2-def angelic-def subset-eq*)

theorem (**in** *graph*) *step1* [*simp*]:
DataRefinement Loop Q3 R2 R2 (*demonic Q3′*)
apply (*simp add*: *DataRefinement-def hoare-demonic Loop-def Loop′-def R2-def Q3-def Q3′-def angelic-def subset-eq*)
apply (*simp add*: *simp-eq-emptyset*)
by (*metis Collect-def List.set.simps*(*2*) *hd-in-set mem-def member-set simps*(*2*))

theorem (**in** *graph*) *step2* [*simp*]:
DataRefinement Loop Q4 R2 R2 (*demonic Q4′*)
apply (*simp add*: *DataRefinement-def hoare-demonic Loop-def Loop′-def R2-def Q4-def Q4′-def angelic-def subset-eq*)
apply (*simp add*: *simp-eq-emptyset*)
apply *clarify*
apply (*case-tac a*)
by *auto*

theorem (**in** *graph*) *final* [*simp*]:
DataRefinement Loop Q5 R2 R1 (*demonic Q5′*)
apply (*simp add*: *DataRefinement-def hoare-demonic Loop-def Loop′-def R2-def R1-def Q5-def Q5′-def angelic-def subset-eq*)

by (*simp add*: *simp-eq-emptyset*)

4.5 Diagram data refinement

theorem (**in** *graph*) *StackMark-DataRefinement* [*simp*]:
DgrDataRefinement SetMarkInv SetMark-rel R (*dgr-demonic StackMark-rel*)
by (*simp add*: *DgrDataRefinement-def dgr-demonic-def StackMark-rel-def SetMark-rel-def demonic-sup-inf SetMarkInv-def data-refinement-choice2*)

4.6 Diagram correctness

theorem (**in** *graph*) *StackMark-correct*:
Hoare-dgr (*dangelic R SetMarkInv*) (*dgr-demonic StackMark-rel*) ((*dangelic R SetMarkInv*) $\sqcap$ ($-$ *grd* (*step* ((*dgr-demonic StackMark-rel*)))))
apply (*rule-tac T=SetMark-rel* **in** *Diagram-DataRefinement*)
apply *auto*
by (*rule SetMark-correct1*)

end

5 Generalization of Deutsch-Schorr-Waite Algorithm

theory *LinkMark*

imports *StackMark*

begin

In the third step the stack diagram is refined to a diagram where no extra memory is used. The relation *next* is replaced by two new variables *link* and *label*. The variable *label* : *node* $\rightarrow$ *index* associates a label to every node and the variable *link* : *index* $\rightarrow$ *node* $\rightarrow$ *node* is a collection of pointer functions indexed by the set *index* of labels. For $x \in node$, *link i x* is the successor node of x along the function *link i*. In this context a node x is reachable if there exists a path from the root to x along the links *link i* such that all nodes in this path are not *nil* and they are labeled by a special label *none* $\in$ *index*.

The stack variable S is replaced by two new variables p and t ranging over nodes. Variable p stores the head of S, t stores the head of the tail of S, and the rest of S is stored by temporarily modifying the variables *link* and *label*.

This algorithm is a generalization of the Deutsch-Schorr-Waite graph marking algorithm because we have a collection of pointer functions instead of left and right only.

locale *pointer* = *node* +

fixes *none* :: *'index*
fixes *link0*::*'index* ⇒ *'node* ⇒ *'node*
fixes *label0* :: *'node* ⇒ *'index*

assumes (*nil*::*'node*) = *nil*
begin
definition *next* = {(*a*, *b*) . (∃ *i* . *link0 i a* = *b*) ∧ *a* ≠ *nil* ∧ *b* ≠ *nil* ∧ *label0 a* = *none*}
end

sublocale *pointer* ⊆ *graph nil root next*
apply *unfold-locales*
apply (*unfold next-def*)
by *auto*

The locale pointer fixes the initial values for the variables *link* and *label* and it defines the relation next as the union of all *link i* functions, excluding the mappings to *nil*, the mappings from *nil* as well as the mappings from elements which are not labeled by *none*.

The next two recursive functions, *label*_0, *link*_0 are used to compute the initial values of the variables *label* and *link* from their current values.

context *pointer*
begin
primrec
label-0:: (*'node* ⇒ *'index*) ⇒ (*'node list*) ⇒ (*'node* ⇒ *'index*) **where**
label-0 lbl [] = *lbl* |
label-0 lbl (*x* # *l*) = *label-0* (*lbl*(*x* := *none*)) *l*

lemma *label-cong* [*cong*]: *f* = *g* ⟹ *xs* = *ys* ⟹ *pointer.label-0 n f xs* = *pointer.label-0 n g ys*
by *simp*

primrec
link-0:: (*'index* ⇒ *'node* ⇒ *'node*) ⇒ (*'node* ⇒ *'index*) ⇒ *'node* ⇒ (*'node list*) ⇒ (*'index* ⇒ *'node* ⇒ *'node*) **where**
link-0 lnk lbl p [] = *lnk* |
link-0 lnk lbl p (*x* # *l*) = *link-0* (*lnk*((*lbl x*) := ((*lnk* (*lbl x*))(*x* := *p*)))) *lbl x l*

The function *stack* defined bellow is the main data refinement relation connecting the stack from the abstract algorithm to its concrete representation by temporarily modifying the variable *link* and *label*.

primrec
stack:: (*'index* ⇒ *'node* ⇒ *'node*) ⇒ (*'node* ⇒ *'index*) ⇒ *'node* ⇒ (*'node list*) ⇒ *bool* **where**
stack lnk lbl x [] = (*x* = *nil*) |
stack lnk lbl x (*y* # *l*) =
(*x* ≠ *nil* ∧ *x* = *y* ∧ ¬ *x mem l* ∧ *stack lnk lbl* (*lnk* (*lbl x*) *x*) *l*)

lemma *label-out-range0* [*simp*]:
$\neg$ *x mem S* $\Longrightarrow$ *label-0 lbl S x* = *lbl x*
apply (*rule-tac P*=$\forall$ *label* . $\neg$ *x mem S* $\longrightarrow$ *label-0 label S x* = *label x* **in** *mp*)
by (*simp*, *induct-tac S*, *auto*)

lemma *link-out-range0* [*simp*]:
$\neg$ *x mem S* $\Longrightarrow$ *link-0 link label p S i x* = *link i x*
apply (*rule-tac P*=$\forall$ *link p* . $\neg$ *x mem S* $\longrightarrow$ *link-0 link label p S i x* = *link i x* **in** *mp*)
by (*simp*, *induct-tac S*, *auto*)

lemma *link-out-range* [*simp*]: $\neg$ *x mem S* $\Longrightarrow$ *link-0 link* (*label*(*x* := *y*)) *p S* = *link-0 link label p S*
apply (*rule-tac P*=$\forall$ *link p* . $\neg$ *x mem S* $\longrightarrow$ *link-0 link* (*label*(*x* := *y*)) *p S* = *link-0 link label p S* **in** *mp*)
by (*simp*, *induct-tac S*, *auto*)

lemma *empty-stack* [*simp*]: *stack link label nil S* = (*S* = [])
by (*case-tac S*, *simp-all*)

lemma *stack-out-link-range* [*simp*]: $\neg$ *p mem S* $\Longrightarrow$ *stack* (*link*(*i* := (*link i*)(*p* := *q*))) *label x S* = *stack link label x S*
apply (*rule-tac P* = $\forall$ *link x* . $\neg$ *p mem S* $\longrightarrow$ *stack* (*link*(*i* := (*link i*)(*p* := *q*))) *label x S* = *stack link label x S* **in** *mp*)
by (*simp*, *induct-tac S*, *auto*)

lemma *stack-out-label-range* [*simp*]: $\neg$ *p mem S* $\Longrightarrow$ *stack link* (*label*(*p* := *q*)) *x S* = *stack link label x S*
apply (*rule-tac P* = $\forall$ *link x* . $\neg$ *p mem S* $\longrightarrow$ *stack link* (*label*(*p* := *q*)) *x S* = *stack link label x S* **in** *mp*)
by (*simp*, *induct-tac S*, *auto*)

definition
g mrk lbl ptr x $\equiv$ *ptr x* $\neq$ *nil* $\wedge$ *ptr x* $\notin$ *mrk* $\wedge$ *lbl x* = *none*

lemma *g-cong* [*cong*]: *mrk* = *mrk1* $\Longrightarrow$ *lbl* = *lbl1* $\Longrightarrow$ *ptr* = *ptr1* $\Longrightarrow$ *x* = *x1* ==>
pointer.g n m mrk lbl ptr x = *pointer.g n m mrk1 lbl1 ptr1 x1*
by *simp*

5.1 Transitions

definition
Q1'' *s* $\equiv$ *let* (*p*, *t*, *lnk*, *lbl*, *mrk*) = *s in* { (*p'*, *t'*, *lnk'*, *lbl'*, *mrk'*) .
root = *nil* $\wedge$ *p'* = *nil* $\wedge$ *t'* = *nil* $\wedge$ *lnk'* = *lnk* $\wedge$ *lbl'* = *lbl* $\wedge$ *mrk'* = *mrk*}

definition
$Q2''\ s \equiv let\ (p,\ t,\ lnk,\ lbl,\ mrk) = s\ in\ \{\ (p',\ t',\ lnk',\ lbl',\ mrk')\ .$
$root \neq nil \wedge p' = root \wedge t' = nil \wedge lnk' = lnk \wedge lbl' = lbl \wedge mrk' = mrk \cup \{root\}\}$

definition
$Q3''\ s \equiv let\ (p,\ t,\ lnk,\ lbl,\ mrk) = s\ in\ \{\ (p',\ t',\ lnk',\ lbl',\ mrk')\ .$
$p \neq nil \wedge$
$(\exists\ i\ .\ g\ mrk\ lbl\ (lnk\ i)\ p \wedge$
$p' = lnk\ i\ p \wedge t' = p \wedge lnk' = lnk(i := (lnk\ i)(p := t)) \wedge lbl' = lbl(p := i) \wedge$
$mrk' = mrk \cup \{lnk\ i\ p\})\}$

definition
$Q4''\ s \equiv let\ (p,\ t,\ lnk,\ lbl,\ mrk) = s\ in\ \{\ (p',\ t',\ lnk',\ lbl',\ mrk')\ .$
$p \neq nil \wedge$
$(\forall\ i\ .\ \neg\ g\ mrk\ lbl\ (lnk\ i)\ p) \wedge t \neq nil \wedge$
$p' = t \wedge t' = lnk\ (lbl\ t)\ t \wedge lnk' = lnk(lbl\ t := (lnk\ (lbl\ t))(t := p)) \wedge lbl' = lbl(t := none) \wedge$
$mrk' = mrk\}$

definition
$Q5''\ s \equiv let\ (p,\ t,\ lnk,\ lbl,\ mrk) = s\ in\ \{\ (p',\ t',\ lnk',\ lbl',\ mrk')\ .$
$p \neq nil \wedge$
$(\forall\ i\ .\ \neg\ g\ mrk\ lbl\ (lnk\ i)\ p) \wedge t = nil \wedge$
$p' = nil \wedge t' = t \wedge lnk' = lnk \wedge lbl' = lbl \wedge mrk' = mrk\}$

definition
$Q6''\ s \equiv let\ (p,\ t,\ lnk,\ lbl,\ mrk) = s\ in\ \{(p',\ t',\ lnk',\ lbl',\ mrk')\ .\ p = nil \wedge$
$p' = p \wedge t' = t \wedge lnk' = lnk \wedge lbl' = lbl \wedge mrk' = mrk\}$

5.2 Invariants

definition
$Init'' \equiv \{\ (p,\ t,\ lnk,\ lbl,\ mrk)\ .\ lnk = link0 \wedge lbl = label0\}$

definition
$Loop'' \equiv UNIV$

definition
$Final'' \equiv Init''$

5.3 Data refinement relations

definition
$R1' \equiv (\lambda\ (p,\ t,\ lnk,\ lbl,\ mrk)\ .\ \{(stk,\ mrk')\ .\ (p,\ t,\ lnk,\ lbl,\ mrk) \in Init'' \wedge mrk' = mrk\})$

definition

$R2' \equiv (\lambda\ (p,\ t,\ lnk,\ lbl,\ mrk)\ .\ \{(stk,\ mrk')\ .$
$\quad p = head\ stk \wedge$
$\quad t = head\ (tail\ stk) \wedge$
$\quad stack\ lnk\ lbl\ t\ (tail\ stk) \wedge$
$\quad link0 = link\text{-}0\ lnk\ lbl\ p\ (tail\ stk) \wedge$
$\quad label0 = label\text{-}0\ lbl\ (tail\ stk) \wedge$
$\quad \neg\ nil\ mem\ stk \wedge$
$\quad mrk' = mrk\})$

definition [*simp*]:
$R'\ i = (case\ i\ of$
$\quad I.init \Rightarrow R1'\ |$
$\quad I.loop \Rightarrow R2'\ |$
$\quad I.final \Rightarrow R1')$

5.4 Diagram

definition
$LinkMark\text{-}rel = (\lambda\ (i,\ j)\ .\ (case\ (i,\ j)\ of$
$\quad (I.init,\ I.loop) \Rightarrow Q1'' \sqcup Q2''\ |$
$\quad (I.loop,\ I.loop) \Rightarrow Q3'' \sqcup (Q4'' \sqcup Q5'')\ |$
$\quad (I.loop,\ I.final) \Rightarrow Q6''\ |$
$\quad - \Rightarrow \bot))$

definition [*simp*]:
$LinkMarkInv\ i = (case\ i\ of$
$\quad I.init \Rightarrow Init''\ |$
$\quad I.loop \Rightarrow Loop''\ |$
$\quad I.final \Rightarrow Final'')$

5.5 Data refinement of the transitions

theorem *init1* [*simp*]:
DataRefinement Init' Q1' R1' R2' (*demonic Q1''*)
by (*simp add*: *DataRefinement-def hoare-demonic Q1''-def Init'-def Init''-def Loop''-def R1'-def R2'-def Q1'-def tail-def head-def angelic-def subset-eq*)

theorem *init2* [*simp*]:
DataRefinement Init' Q2' R1' R2' (*demonic Q2''*)
by (*simp add*: *DataRefinement-def hoare-demonic Q2''-def Init'-def Init''-def Loop''-def R1'-def R2'-def Q2'-def tail-def head-def angelic-def subset-eq*)

theorem *step1* [*simp*]:
DataRefinement Loop' Q3' R2' R2' (*demonic Q3''*)
apply (*simp add*: *DataRefinement-def hoare-demonic Q3''-def Init'-def Init''-def*

Loop'-def R1'-def Q3'-def tail-def head-def angelic-def subset-eq)
apply (*simp add*: *simp-eq-emptyset*)
apply *auto*
apply (*unfold next-def*)

apply (*unfold R2'-def*)
apply (*simp add*: *simp-eq-emptyset*)
apply *safe*
apply *simp*
apply (*rule-tac x* = *ac i* (*hd a*) # *a* **in** *exI*)
apply *safe*
apply *simp-all*
apply (*simp add*: *g-def neq-Nil-conv*)
apply *clarify*
apply (*simp add*: *g-def neq-Nil-conv*)
apply (*case-tac a*)
apply (*simp-all add*: *g-def neq-Nil-conv*)
apply (*case-tac a*)
apply *simp-all*
apply (*case-tac a*)
by *auto*

lemma *neqif* [*simp*]: $x \neq y \Longrightarrow (\textit{if } y = x \textit{ then } a \textit{ else } b) = b$
apply (*case-tac* $y \neq x$)
apply *simp-all*
done

theorem *step2* [*simp*]:
DataRefinement Loop' Q4' R2' R2' (*demonic Q4''*)
apply (*simp add*: *DataRefinement-def hoare-demonic Q4''-def Init'-def Init''-def*

Loop'-def Q4'-def angelic-def subset-eq)
apply (*simp add*: *simp-eq-emptyset*)
apply *safe*
apply (*unfold next-def*)
apply (*unfold R2'-def*)
apply (*simp-all add*: *neq-Nil-conv*)
apply *auto* [*1*]
apply (*case-tac ysa*)
apply (*simp add*: *head-def*)
apply (*simp add*: *head-def*)
apply (*case-tac a*)
apply *simp*
apply *simp*
apply (*unfold g-def*)
by *auto*

lemma *setsimp*: $a = c \Longrightarrow (x \in a) = (x \in c)$
apply *simp*
done

theorem *step3* [*simp*]:
DataRefinement Loop' Q4' R2' R2' (*demonic Q5''*)
apply (*simp add*: *DataRefinement-def hoare-demonic Q5''-def Init'-def Init''-def*

Loop′-def Q4′-def angelic-def subset-eq)
apply (*unfold R2′-def*)
apply (*unfold next-def*)
apply (*simp add*: *simp-eq-emptyset*)
apply (*unfold g-def*)
apply *safe*
apply (*simp-all add*: *head-def tail-def*)
by *auto*

theorem *final* [*simp*]:
DataRefinement Loop′ Q5′ R2′ R1′ (*demonic Q6′′*)
apply (*simp add*: *DataRefinement-def hoare-demonic Q6′′-def Init′-def Init′′-def*

Loop′-def R2′-def R1′-def Q5′-def angelic-def subset-eq neq-Nil-conv tail-def head-def)
apply (*simp add*: *simp-eq-emptyset*)
apply *safe*
by *simp-all*

5.6 Diagram data refinement

theorem *LinkMark-DataRefinement* [*simp*]:
DgrDataRefinement (*dangelic R SetMarkInv*) *StackMark-rel R′* (*dgr-demonic LinkMark-rel*)
apply (*rule-tac P* = *StackMarkInv* **in** *DgrDataRefinement-mono*)
apply (*simp add*: *le-fun-def dangelic-def SetMarkInv-def angelic-def R1-def R2-def Init′-def Final′-def mem-def le-bool-def*)
apply *auto*
apply (*unfold simp-set-function*)
apply *auto*
apply (*simp add*: *DgrDataRefinement-def dgr-demonic-def LinkMark-rel-def StackMark-rel-def demonic-sup-inf data-refinement-choice2*)
apply (*rule data-refinement-choice2*)
apply *auto*
apply (*rule data-refinement-choice*)
by *auto*

5.7 Diagram correctness

theorem *LinkMark-correct*:
Hoare-dgr (*dangelic R′* (*dangelic R SetMarkInv*)) (*dgr-demonic LinkMark-rel*)
((*dangelic R′* (*dangelic R SetMarkInv*)) ⊓ (− *grd* (*step* ((*dgr-demonic LinkMark-rel*)))))
apply (*rule-tac T*=*StackMark-rel* **in** *Diagram-DataRefinement*)
apply *auto*
by (*rule StackMark-correct*)

end

end

6 Deutsch-Schorr-Waite Marking Algorithm

theory *DSWMark*

imports *LinkMark*

begin

Finally, we construct the Deutsch-Schorr-Waite marking algorithm by assuming that there are only two pointers (*left*, *right*) from every node. There is also a new variable, $atom : node \to bool$ which associates to every node a Boolean value. The data invariant of this refinement step requires that *index* has exactly two distinct elements *none* and *some*, $left = link\ none$, $right = link\ some$, and $atom\ x$ is true if and only if $label\ x = some$.

We use a new locale which fixes the iniatial values of the variables *left*, *right*, and *atom* in *left0*, *right0*, and *atom0* respectively.

datatype $Index = none \mid some$

locale $classical = node\ +$
  fixes $left0\ :: {}'node \Rightarrow {}'node$
  fixes $right0 :: {}'node \Rightarrow {}'node$
  fixes $atom0\ :: {}'node \Rightarrow bool$
  assumes $(nil{::}'node) = nil$
  begin
    definition
      $link0\ i = (if\ i = (none{::}Index)\ then\ left0\ else\ right0)$

    definition
      $label0\ x = (if\ atom0\ x\ then\ (some{::}Index)\ else\ none)$
  end

sublocale $classical \subseteq pointer\ nil\ root\ none{::}Index\ link0\ label0$
proof qed *auto*

context *classical*
begin

lemma [*simp*]:
  $(label0 = (\lambda\ x\ .\ if\ atom\ x\ then\ some\ else\ none)) = (atom0 = atom)$
  apply (*simp add*: *expand-fun-eq label0-def*)
  by *auto*

definition
  $gg\ mrk\ atom\ ptr\ x \equiv ptr\ x \neq nil \wedge ptr\ x \notin mrk \wedge \neg\ atom\ x$

6.1 Transitions

definition

$QQ1 \equiv \lambda\ (p,\ t,\ left,\ right,\ atom,\ mrk)\ .\ \{(p',\ t',\ left',\ right',\ atom',\ mrk')\ .$
$root = nil \wedge\ \ p' = nil \wedge t' = nil \wedge mrk' = mrk \wedge left' = left \wedge right' = right \wedge atom' = atom\}$

definition
$QQ2 \equiv \lambda\ (p,\ t,\ left,\ right,\ atom,\ mrk)\ .\ \{(p',\ t',\ left',\ right',\ atom',\ mrk')\ .$
$root \neq nil \wedge\ \ p' = root \wedge t' = nil \wedge mrk' = mrk \cup \{root\} \wedge left' = left \wedge right' = right \wedge atom' = atom\}$

definition
$QQ3 \equiv \lambda\ (p,\ t,\ left,\ right,\ atom,\ mrk)\ .\ \{(p',\ t',\ left',\ right',\ atom',\ mrk')\ .$
$p \neq nil \wedge gg\ mrk\ atom\ left\ p \wedge$
$p' = left\ p \wedge t' = p \wedge mrk' = mrk \cup \{left\ p\} \wedge$
$left' = left(p := t) \wedge right' = right \wedge atom' = atom\}$

definition
$QQ4 \equiv \lambda\ (p,\ t,\ left,\ right,\ atom,\ mrk)\ .\ \{(p',\ t',\ left',\ right',\ atom',\ mrk')\ .$
$p \neq nil \wedge gg\ mrk\ atom\ right\ p \wedge$
$p' = right\ p \wedge t' = p \wedge mrk' = mrk \cup \{right\ p\} \wedge$
$left' = left \wedge right' = right(p := t) \wedge atom' = atom(p := True)\}$

definition
$QQ5 \equiv\ \lambda\ (p,\ t,\ left,\ right,\ atom,\ mrk)\ .\ \{(p',\ t',\ left',\ right',\ atom',\ mrk')\ .$
$p \neq nil \wedge (*not\ needed\ in\ the\ proof*)$
$\neg\ gg\ mrk\ atom\ left\ p \wedge \neg\ gg\ mrk\ atom\ right\ p \wedge$
$t \neq nil \wedge \neg\ atom\ t \wedge$
$p' = t \wedge t' = left\ t \wedge mrk' = mrk \wedge$
$left' = left(t := p) \wedge right' = right \wedge atom' = atom\}$

definition
$QQ6 \equiv \lambda\ (p,\ t,\ left,\ right,\ atom,\ mrk)\ .\ \{(p',\ t',\ left',\ right',\ atom',\ mrk')\ .$
$p \neq nil \wedge (*not\ needed\ in\ the\ proof*)$
$\neg\ gg\ mrk\ atom\ left\ p \wedge \neg\ gg\ mrk\ atom\ right\ p \wedge$
$t \neq nil \wedge atom\ t \wedge$
$p' = t \wedge t' = right\ t \wedge mrk' = mrk \wedge$
$left' = left \wedge right' = right(t := p) \wedge atom' = atom(t := False)\}$

definition
$QQ7 \equiv \lambda\ (p,\ t,\ left,\ right,\ atom,\ mrk)\ .\ \{(p',\ t',\ left',\ right',\ atom',\ mrk')\ .$
$p \neq nil \wedge$
$\neg\ gg\ mrk\ atom\ left\ p \wedge \neg\ gg\ mrk\ atom\ right\ p \wedge$
$t = nil \wedge$
$p' = nil \wedge t' = t \wedge mrk' = mrk \wedge$
$left' = left \wedge right' = right \wedge atom' = atom\}$

definition
$QQ8 \equiv \lambda\ (p,\ t,\ left,\ right,\ atom,\ mrk)\ .\ \{(p',\ t',\ left',\ right',\ atom',\ mrk')\ .$

$p = nil \wedge p' = p \wedge t' = t \wedge mrk' = mrk \wedge left' = left \wedge right' = right \wedge atom' = atom\}$

7 Data refinement relation

definition
$RR \equiv \lambda\ (p,\ t,\ left,\ right,\ atom,\ mrk)\ .\ \{(p',\ t',\ lnk,\ lbl,\ mrk')\ .$
$lnk\ none = left \wedge lnk\ some = right \wedge$
$lbl = (\lambda\ x\ .\ if\ atom\ x\ then\ some\ else\ none) \wedge$
$p' = p \wedge t' = t \wedge mrk' = mrk\}$

definition [*simp*]:
$R''\ i = RR$

definition
$ClassicMark\text{-}rel = (\lambda\ (i,\ j)\ .\ (case\ (i,\ j)\ of$
$(I.init,\ I.loop)\ \Rightarrow QQ1 \sqcup QQ2\ |$
$(I.loop,\ I.loop)\ \Rightarrow (QQ3 \sqcup QQ4) \sqcup ((QQ5 \sqcup QQ6) \sqcup QQ7)\ |$
$(I.loop,\ I.final) \Rightarrow QQ8\ |$
$\text{-} \Rightarrow \bot))$

7.1 Data refinement of the transitions

theorem *init1* [*simp*]:
DataRefinement Init'' Q1'' RR RR (*demonic QQ1*)
by (*simp add*: *DataRefinement-def hoare-demonic angelic-def QQ1-def Q1''-def RR-def*
Init''-def subset-eq)

theorem *init2* [*simp*]:
DataRefinement Init'' Q2'' RR RR (*demonic QQ2*)
by (*simp add*: *DataRefinement-def hoare-demonic angelic-def QQ2-def Q2''-def RR-def*
Init''-def subset-eq)

lemma *index-simp*:
$(u = v) = (u\ none = v\ none \wedge u\ some = v\ some)$
by (*safe*, *rule ext*, *case-tac x*, *auto*)

theorem *step1* [*simp*]:
DataRefinement Loop'' Q3'' RR RR (*demonic QQ3*)
apply (*simp add*: *DataRefinement-def hoare-demonic angelic-def QQ3-def Q3''-def RR-def*
Loop''-def subset-eq g-def gg-def simp-eq-emptyset)
apply *safe*
apply (*rule-tac* $x{=}\lambda\ x\ .\ if\ x = some\ then\ ab\ some\ else\ (ab\ none)(a := aa)$ **in** *exI*)
apply *simp*

apply (*rule-tac x=none* **in** *exI*)
apply (*simp add*: *index-simp*)
done

theorem *step2* [*simp*]:
DataRefinement Loop″ Q3″ RR RR (*demonic QQ4*)
apply (*simp add*: *DataRefinement-def hoare-demonic angelic-def QQ4-def Q3″-def RR-def*
Loop″-def subset-eq g-def gg-def simp-eq-emptyset)
apply *safe*
apply (*rule-tac x*=λ *x* . *if x* = *none then ab none else* (*ab some*)(*a* := *aa*) **in** *exI*)
apply *simp*
apply (*rule-tac x=some* **in** *exI*)
apply (*simp add*: *index-simp*)
apply (*rule ext*)
apply *auto*
done

theorem *step3* [*simp*]:
DataRefinement Loop″ Q4″ RR RR (*demonic QQ5*)
apply (*simp add*: *DataRefinement-def hoare-demonic angelic-def QQ5-def Q4″-def RR-def*
Loop″-def subset-eq g-def gg-def simp-eq-emptyset)
apply *clarify*
apply (*case-tac i*)
apply *auto*
done

lemma *if-set-elim*: ($x \in$ (*if b then A else B*)) = (($b \wedge x \in A$) $\vee$ ($\neg$ $b \wedge x \in B$))
by *auto*

theorem *step4* [*simp*]:
DataRefinement Loop″ Q4″ RR RR (*demonic QQ6*)
apply (*simp add*: *DataRefinement-def hoare-demonic angelic-def QQ6-def Q4″-def*
Loop″-def subset-eq simp-eq-emptyset)

apply *safe*
apply (*simp add*: *RR-def*)
apply *auto*
apply (*rule ext*)
apply (*simp-all*)
apply (*simp-all add*: *RR-def g-def gg-def if-set-elim*)
apply (*case-tac i*)
by *auto*

theorem *step5* [*simp*]:
DataRefinement Loop″ Q5″ RR RR (*demonic QQ7*)

apply (*simp add*: *DataRefinement-def hoare-demonic angelic-def Q5''-def QQ7-def*

Loop''-def subset-eq RR-def simp-eq-emptyset)
apply *safe*
apply (*simp-all add*: *g-def gg-def*)
apply (*case-tac i*)
by *auto*

theorem *final-step* [*simp*]:
DataRefinement Loop'' Q6'' RR RR (*demonic QQ8*)
by (*simp add*: *DataRefinement-def hoare-demonic angelic-def Q6''-def QQ8-def Loop''-def subset-eq RR-def simp-eq-emptyset*)

7.2 Diagram data refinement

theorem *ClassicMark-DataRefinement* [*simp*]:
DgrDataRefinement (*dangelic R'* (*dangelic R SetMarkInv*)) *LinkMark-rel R''* (*dgr-demonic ClassicMark-rel*)
apply (*rule-tac P* = *LinkMarkInv* **in** *DgrDataRefinement-mono*)
apply (*simp add*: *le-fun-def dangelic-def SetMarkInv-def angelic-def R1'-def R2'-def Init''-def Loop''-def Final''-def mem-def le-bool-def*)
apply *auto*
apply (*unfold simp-set-function*)
apply *auto*
apply (*simp add*: *DgrDataRefinement-def dgr-demonic-def ClassicMark-rel-def LinkMark-rel-def demonic-sup-inf data-refinement-choice2*)
apply (*rule data-refinement-choice2*)
apply *auto*
apply (*rule data-refinement-choice*)
apply *auto*
apply (*rule data-refinement-choice2*)
apply *auto*
apply (*rule data-refinement-choice*)
by *auto*

7.3 Diagram corectness

theorem *ClassicMark-correct* [*simp*]:
Hoare-dgr (*dangelic R''* (*dangelic R'* (*dangelic R SetMarkInv*))) (*dgr-demonic ClassicMark-rel*)
((*dangelic R''* (*dangelic R'* (*dangelic R SetMarkInv*))) $\sqcap$ ($-$ *grd* (*step* ((*dgr-demonic ClassicMark-rel*)))))
apply (*rule-tac T*=*LinkMark-rel* **in** *Diagram-DataRefinement*)
apply *auto*
by (*rule LinkMark-correct*)

We have proved the correctness of the final algorithm, but the pre and the post conditions involve the angelic choice operator and they depend on all data refinement steps we have used to prove the final diagram. We simplify these conditions and we show that we obtained indead the corectness of the marking algorithm.

The predicate *ClassicInit* which is true for the *init* situation states that initially the variables *left*, *right*, and *atom* are equal to their initial values and also that no node is marked.

The predicate *ClassicFinal* which is true for the $final$ situation states that at the end the values of the variables *left*, *right*, and *atom* are again equal to their initial values and the variable mrk records all reachable nodes. The reachable nodes are defined using our initial *next* relation, however if we unfold all locale interpretations and definitions we see easeily that a node x is reachable if there is a path from *root* to x along *left* and *right* functions, and all nodes in this path have the atom bit false.

definition
ClassicInit = {(*p*, *t*, *left*, *right*, *atom*, *mrk*) .
atom = *atom0* ∧ *left* = *left0* ∧ *right* = *right0* ∧
finite (− *mrk*) ∧ *mrk* = {}}

definition
ClassicFinal = {(*p*, *t*, *left*, *right*, *atom*, *mrk*) .
atom = *atom0* ∧ *left* = *left0* ∧ *right* = *right0* ∧
mrk = *reach root*}

theorem [*simp*]:
ClassicInit ⊆ (*angelic RR* (*angelic R1'* (*angelic R1* (*SetMarkInv init*))))
apply (*simp add*: *SetMarkInv-def*)
apply (*simp add*: *ClassicInit-def angelic-def RR-def R1'-def R1-def Init-def Init''-def*)
apply (*simp add*: *mem-def simp-eq-emptyset inf-fun-eq inf-bool-eq*)
apply *clarify*
apply (*simp add*: *simp-set-function*)
apply (*simp-all add*: *expand-fun-eq link0-def label0-def simp-set-function*)
done

theorem [*simp*]:
ClassicInit ⊆ (*angelic* (*R'' init*) (*angelic* (*R' init*) (*angelic* (*R init*) (*SetMarkInv init*))))
by (*simp add*: *R''-def R'-def R-def*)

theorem [*simp*]:
(*angelic RR* (*angelic R1'* (*angelic R1* (*SetMarkInv final*)))) ≤ *ClassicFinal*
apply (*simp add*: *SetMarkInv-def*)
apply (*simp add*: *ClassicFinal-def angelic-def RR-def R1'-def R1-def Final-def Final''-def Init''-def label0-def link0-def*)
apply (*simp add*: *mem-def simp-eq-emptyset inf-fun-eq inf-bool-eq*)
apply (*simp-all add*: *simp-set-function link0-def*)
apply *safe*
apply (*simp-all add*: *simp-set-function*)

apply (*simp-all add*: *link0-def*)
by *auto*

theorem [*simp*]:
($angelic$ (R'' *final*) ($angelic$ (R' *final*) ($angelic$ (R *final*) (*SetMarkInv final*)))) $\leq$ *ClassicFinal*
by (*simp add*: R''-*def* R'-*def* R-*def*)

The indexed predicate *ClassicPre* is the precondition of the diagram, and since we are only interested in starting the marking diagram in the *init* situation we set *ClassicPre loop* = *ClassicPre final* = $\emptyset$.

definition [*simp*]:
ClassicPre i = (*case i of*
I.init $\Rightarrow$ *ClassicInit* |
I.loop $\Rightarrow$ {} |
I.final $\Rightarrow$ {})

We are interested on the other hand that the marking diagram terminates only in the *final* situation. In order to achieve this we define the postcondition of the diagram as the indexed predicate *ClassicPost* which is empty on every situation except $final$.

definition [*simp*]:
ClassicPost i = (*case i of*
I.init $\Rightarrow$ {} |
I.loop $\Rightarrow$ {} |
I.final $\Rightarrow$ *ClassicFinal*)

definition [*simp*]:
ClassicMark = *dgr-demonic ClassicMark-rel*

lemma *exists-or*:
$(\exists\ x\ .\ p\ x \vee q\ x) = ((\exists\ x\ .\ p\ x) \vee (\exists\ x\ .\ q\ x))$
by *auto*

lemma [*simp*]:
(− *grd* (*step* (*dgr-demonic ClassicMark-rel*))) *init* = {}
apply (*unfold expand-fun-eq*)
apply (*unfold fun-Compl-def*)
apply *simp*
apply (*unfold dgr-demonic-def*)
apply *safe*
apply (*unfold grd-demonic*)
apply *auto*
apply (*unfold ClassicMark-rel-def*)
apply *auto*
apply (*rule-tac x* = *loop* **in** *exI*)
apply *auto*

apply (*simp add*: *QQ1-def QQ2-def sup-fun-eq sup-bool-eq*)
by (*simp add*: *mem-def simp-set-function exists-or*)

lemma [*simp*]:
(− *grd* (*step* (*dgr-demonic ClassicMark-rel*))) *loop* = {}
apply (*unfold expand-fun-eq*)
apply (*unfold fun-Compl-def*)
apply *simp*
apply (*unfold dgr-demonic-def*)
apply *safe*
apply (*unfold grd-demonic*)
apply *auto*
apply (*unfold ClassicMark-rel-def*)
apply *auto*
apply (*simp add*: *bot-fun-eq bot-bool-eq*)
apply (*simp add*: *QQ3-def QQ4-def QQ5-def QQ6-def QQ7-def QQ8-def sup-fun-eq sup-bool-eq*)
apply (*simp add*: *mem-def simp-set-function*)
apply (*case-tac* $a \neq nil$)
apply *auto*
apply (*rule-tac* $x = loop$ **in** *exI*)
apply *auto*
apply (*simp add*: *exists-or*)
apply (*unfold gg-def*)
by *auto*

The final theorem states the correctness of the marking diagram with respect to the precondition *ClassicPre* and the postcondition *ClassicPost*, that is, if the diagram starts in the initial situation, then it will terminate in the final situation, and it will mark all reachable nodes.

theorem $\models$ *ClassicPre* {| *pt ClassicMark* |} *ClassicPost*
apply (*rule-tac* P=(*dangelic* R'' (*dangelic* R' (*dangelic R SetMarkInv*))) **in** *hoare-pre*)
apply (*subst le-fun-def*)
apply (*simp add*: *dangelic-def*)
apply (*rule-tac* Q = ((*dangelic* R'' (*dangelic* R' (*dangelic R SetMarkInv*))) $\sqcap$ (− *grd* (*step* ((*dgr-demonic ClassicMark-rel*))))) **in** *hoare-mono*)
apply *simp*
apply (*rule le-funI*)
apply (*case-tac x*)
apply (*rule le-funI*)
apply (*simp add*: *inf-fun-eq mem-def*)
apply (*rule le-funI*)
apply (*simp add*: *inf-fun-eq mem-def*)
apply (*subst inf-fun-eq*)
apply (*simp-all add*: *dangelic-def*)
apply (*rule-tac* y = *angelic RR* (*angelic R1'* (*angelic R1* (*SetMarkInv final*))) **in** *order-trans*)
apply *auto* [*1*]

by (*simp-all add*: *hoare-dgr-correctness*)
end

end

References

[1] J.-R. Abrial. Event based sequential program development: Application to constructing a pointer program. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 51–74. Springer, 2003.

[2] R. J. Back. *Correctness preserving program refinements: proof theory and applications*, volume 131 of *Mathematical Centre Tracts*. Mathematisch Centrum, Amsterdam, 1980.

[3] R.-J. Back. Semantic correctness of invariant based programs. In *International Workshop on Program Construction*, Chateau de Bonas, France, 1980.

[4] R.-J. Back. Invariant based programs and their correctness. In W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 223–242. MacMillan Publishing Company, 1983.

[5] R.-J. Back. Invariant based programming: Basic approach and teaching experience. *Formal Aspects of Computing*, 2008.

[6] R.-J. Back and V. Preoteasa. Semantics and proof rules of invariant based programs. Technical Report 903, TUCS, Jul 2008.

[7] R. J. Back and J. von Wright. Encoding, decoding and data refinement. *Formal Aspects of Computing*, 12:313–349, 2000.

[8] W. DeRoever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison.* Cambridge University Press, New York, NY, USA, 1999.

[9] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4), Dec. 1972.

[10] D. E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[11] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.

[12] V. Preoteasa and R.-J. Back. Data refinement of invariant based programs. *Electronic Notes in Theoretical Computer Science*, 259:143 – 163, 2009. Proceedings of the 14th BCS-FACS Refinement Workshop (REFINE 2009).

[13] V. Preoteasa and R.-J. Back. Semantics and data refinement of invariant based programs. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. http://afp.sf.net/entries/DataRefinementIBP.shtml, May 2010. Formal proof development. Submitted.

[14] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.