

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

Viorel Preoteasa and Ralph-Johan Back

September 13, 2023

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	2
2	Address Graph	3
3	Marking Using a Set	5
3.1	Transitions	8
3.2	Invariants	8
3.3	Diagram	9
3.4	Correctness of the transitions	9
3.5	Diagram correctness	11
4	Marking Using a Stack	12
4.1	Transitions	12
4.2	Invariants	12
4.3	Data refinement relations	13
4.4	Data refinement of the transitions	13
4.5	Diagram data refinement	14
4.6	Diagram correctness	15

5	Generalization of Deutsch-Schorr-Waite Algorithm	15
5.1	Transitions	17
5.2	Invariants	18
5.3	Data refinement relations	18
5.4	Diagram	19
5.5	Data refinement of the transitions	19
5.6	Diagram data refinement	21
5.7	Diagram correctness	22
6	Deutsch-Schorr-Waite Marking Algorithm	22
6.1	Transitions	23
7	Data refinement relation	24
7.1	Data refinement of the transitions	24
7.2	Diagram data refinement	26
7.3	Diagram corectness	27

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 ($x.left := y$) are done by modifying the global pointer function $left := left(x := y)$. 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 polymorphism is used for the state of the computation. In [13] the state of computation was introduced as a type variable, or even more generally, 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 returns `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]: $((nil, x) \in next) = False$
by (simp add: next-not-nil-left)

lemma [simp]: $((x, nil) \in next) = False$
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)

```

definition (in graph)

```

  reach x ≡ {y . (x, y) ∈ next* ∧ y ≠ 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: $b \in reach\ a \implies (b, c) \in next \implies c \in reach\ a$

```

apply (simp add: reach-def)

```

by *auto*

definition (in *graph*)
 $path\ S\ mrk \equiv \{x . (\exists\ s . s \in S \wedge (s, x) \in next\ O\ (next \cap ((-mrk) \times (-mrk)))^*$
 $\})\}$
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 successor of *x* we add *y* to *X*. If $x \in X$ has no unmarked successors it is removed from *X*. The algorithm terminates when *X* is empty.

datatype *I* = *init* | *loop* | *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 ≤ y ↔ x = y ∨ x < y
    by (simp add: less-eq-I-def)
next
  fix P fix a::I
  show P a when ∀x. (∀ y. y < x → P y) → P x
    apply (insert that)
    apply (case-tac P final)
    apply (case-tac P loop)
    apply (simp-all add: less-I-def)
    by blast
qed (simp)

end

```

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

```

lemma trascl-less: x ≠ y ⇒ (a, x) ∈ R* ⇒
  ((a,x) ∈ (R ∩ (-{y})×(-{y}))* ∨ (y,x) ∈ R O (R ∩ (-{y})×(-{y}))*)
  apply (drule-tac
    b = x and a = a and r = R and
    P = λ x. (x ≠ y → ((a,x) ∈ (R ∩ (-{y})×(-{y}))* ∨ (y,x) ∈ R O (R ∩
(-{y})×(-{y}))*)))
  in rtrancl-induct)
  apply (auto simp: Compl-insert)
  apply (case-tac ya = y)
  apply auto
  apply (rule-tac x = a and y = ya and z = z and r = R ∩ ((UNIV - {y}) ×
(UNIV - {y})) in rtrancl-trans)
  apply auto
  apply (case-tac za = y)
  apply auto
  apply (drule-tac x = ya and y = za and z = z and r = (R ∩ (UNIV - {y})
× (UNIV - {y})) in rtrancl-trans)
  by auto

```

```

lemma (in graph) add-set [simp]: x ≠ y ⇒ x ∈ path S mrk ⇒ x ∈ 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 ∩ (- mrk) × (-
mrk) in trascl-less)
  apply simp-all
  apply (case-tac (ya, x) ∈ (next ∩ (- mrk) × - mrk ∩ (- {y}) × - {y})*)
  apply (rule-tac x = xa in exI)
  apply simp-all
  apply (simp add: relcomp-unfold)
  apply (rule-tac x = ya in exI)

```

apply *simp*
apply (*case-tac* ($next \cap (-\text{mrk}) \times -\text{mrk} \cap (-\{y\}) \times -\{y\}) = (next \cap (-\text{insert } y \text{ mrk}) \times -\text{insert } y \text{ mrk}))$)
apply *simp-all*
apply *safe*
apply *simp-all*
apply (*rule-tac* $x = y$ **in** *exI*)
apply *simp*
apply (*simp add: relcomp-unfold*)
apply (*rule-tac* $x = yaa$ **in** *exI*)
apply *simp*
apply (*case-tac* ($next \cap (-\text{mrk}) \times -\text{mrk} \cap (-\{y\}) \times -\{y\}) = (next \cap (-\text{insert } y \text{ mrk}) \times -\text{insert } y \text{ mrk}))$)
apply *simp-all*
by *auto*

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

lemma (**in** *graph*) *del-stack* [*simp*]: $(\forall y . (t, y) \in next \longrightarrow y \in \text{mrk}) \implies x \notin \text{mrk} \implies x \in \text{path } S \text{ mrk} \implies x \in \text{path } (S - \{t\}) \text{ mrk}$
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** $R = (next \cap (-\text{mrk}) \times -\text{mrk})$ **in** *rtranclD*)
apply *safe*
apply (*drule-tac* $x = y$ **and** $y = x$ **in** *tranclD*)
by *auto*

lemma (**in** *graph*) *init-set* [*simp*]: $x \in \text{reach } root \implies x \neq root \implies x \in \text{path } \{root\} \{root\}$
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 add: Compl-insert*)
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*: $x \in \text{reach } root \implies x \notin \text{path } \{root\} \{root\} \implies x =$

root
apply (*case-tac* $root \neq x$)
apply (*drule* *init-set*)
by *simp-all*

3.1 Transitions

definition (*in graph*)

$Q1-a \equiv [: X, mrk \rightsquigarrow X', mrk'. (root::'node) = nil \wedge X' = \{\} \wedge mrk' = mrk :]$

definition (*in graph*)

$Q2-a \equiv [: X, mrk \rightsquigarrow X', mrk'. (root::'node) \neq nil \wedge X' = \{root::'node\} \wedge mrk' = \{root::'node\} :]$

definition (*in graph*)

$Q3-a \equiv [: X, mrk \rightsquigarrow X', mrk'. (\exists x \in X . \exists y . (x, y) \in next \wedge y \notin mrk \wedge X' = X \cup \{y\} \wedge mrk' = mrk \cup \{y\}) :]$

definition (*in graph*)

$Q4-a \equiv [: X, mrk \rightsquigarrow X', mrk'. (\exists x \in X . (\forall y . (x, y) \in next \longrightarrow y \in mrk) \wedge X' = X - \{x\} \wedge mrk' = mrk) :]$

definition (*in graph*)

$Q5-a \equiv [: X, mrk \rightsquigarrow X', mrk'. X = \{\} \wedge mrk = mrk' :]$

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

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

definition

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

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

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

by *auto*

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

apply (*simp* *add: term-eq-def term-less-def*)

by *auto*

definition (in *graph*)

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

definition (in *graph*)

$Final \equiv \{ (X::('node\ set),\ mrk::('node\ set)) .\ mrk = reach\ root\}$

definition (in *graph*)

$SetMarkInv\ i = (case\ i\ of$
 $I.init \Rightarrow Init\ |$
 $I.loop \Rightarrow Loop\ |$
 $I.final \Rightarrow Final)$

definition (in *graph*)

$SetMarkInvFinal\ i = (case\ i\ of$
 $I.final \Rightarrow Final\ |$
 $- \Rightarrow \{\})$

definition (in *graph*) [*simp*]:

$SetMarkInvTerm\ w\ i = (case\ i\ of$
 $I.init \Rightarrow Init\ |$
 $I.loop \Rightarrow Loop \cap \{s .\ trm\ s = w\}\ |$
 $I.final \Rightarrow Final)$

3.3 Diagram

definition (in *graph*)

$SetMark \equiv \lambda\ (i,\ j) . (case\ (i,\ j)\ of$
 $(I.init,\ I.loop) \Rightarrow Q1-a \sqcap Q2-a\ |$
 $(I.loop,\ I.loop) \Rightarrow Q3-a \sqcap Q4-a\ |$
 $(I.loop,\ I.final) \Rightarrow Q5-a\ |$
 $- \Rightarrow top)$

lemma (in *graph*) *SetMark-dmono* [*simp*]:

dmono SetMark

apply (*unfold dmono-def SetMark-def Q1-a-def Q2-a-def Q3-a-def Q4-a-def Q5-a-def*)

by *simp*

3.4 Correctness of the transitions

lemma (in *graph*) *init-loop-1-a*[*simp*]: $\models Init\ \{\mid Q1-a\ \}\ Loop$

apply (*unfold hoare-demonic Init-def Q1-a-def Loop-def*)

by *auto*

lemma (in *graph*) *init-loop-2-a*[*simp*]: $\models Init\ \{\mid Q2-a\ \}\ Loop$

apply (*simp add: hoare-demonic Init-def Q2-a-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-a* [*simp*]: $\models (Loop \cap \{s . trm\ s = w\}) \{ \mid Q3-a \}$
 $(Loop \cap \{s . trm\ s < w\})$
apply (*simp add: hoare-demonic Q3-a-def Loop-def trm-def*)
apply *safe*
apply (*simp-all*)
apply (*simp-all add: reach-def subset-eq*)
apply *safe*
apply (*simp-all add: Compl-insert*)
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-a*[*simp*]: $\models (Loop \cap \{s . trm\ s = w\}) \{ \mid Q4-a \}$
 $(Loop \cap \{s . trm\ s < w\})$
apply (*simp add: hoare-demonic Q4-a-def Loop-def trm-def*)
apply *auto*
apply (*case-tac card a > 0*)
by *auto*

lemma (*in graph*) *loop-final-a* [*simp*]: $\models (Loop \cap \{s . trm\ s = w\}) \{ \mid Q5-a \}$
Final
apply (*simp add: hoare-demonic Q5-a-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*]: $Sup\ (range\ A)\ i = (\bigcup w . A\ w\ i)$
by (*simp-all add: Sup-fun-def*)

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

lemma *forall-simp2* [*simp*]: $(\forall a\ b . \forall x \in A . \forall y . (a = t\ x\ y) \longrightarrow (h\ x\ y) \longrightarrow (g\ x\ y) \vee b \neq u\ x\ y) = (\forall x \in A . \forall y . h\ x\ y \longrightarrow g\ x\ y)$
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 $n \in \text{nat}$.

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

theorem (in *graph*) *SetMark-correct*:

$\models \text{SetMarkInv } \{| \text{pt } \text{SetMark} | \} \text{SetMarkInvFinal}$

proof (rule-tac $X = \text{SetMarkInvTerm}$ in *hoare-diagram3*)

show *dmono SetMark* **by** *simp*

show $\forall u i j. \models \text{SetMarkInvTerm } u i \{| \text{SetMark } (i, j) | \}$

DiagramTermination.SUP-L-P $(\lambda n i. (i, n)) \text{SetMarkInvTerm } (i, u) j$

by (*auto simp add: SUP-L-P-def less-pair-def less-I-def hoare-choice SetMark-def*)

show $\text{SetMarkInv} \leq \text{Sup } (\text{range } \text{SetMarkInvTerm})$

apply (*simp add: le-fun-def, safe*)

apply (*simp-all add: SetMarkInv-def*)

apply (*case-tac x*)

apply *auto*

done

show $\text{Sup } (\text{range } \text{SetMarkInvTerm}) \sqcap - \text{grd } (\text{step } \text{SetMark}) \leq \text{SetMarkInvFinal}$

apply (*simp add: le-fun-def inf-fun-def SetMarkInvFinal-def*)

apply *safe*

apply *simp-all*

apply (*drule-tac x=I.loop in spec*)

apply (*simp add: SetMark-def*)

apply (*simp add: Q1-a-def Q2-a-def*)

apply (*frule-tac x=I.loop in spec*)

apply (*drule-tac x=I.final in spec*)

apply (*simp add: SetMark-def*)

apply (*simp add: Q3-a-def Q4-a-def Q5-a-def*)

apply (*auto*)

done

qed

theorem (in *graph*) *SetMark-correct1* [*simp*]:

Hoare-dgr SetMarkInv SetMark (SetMarkInv \sqcap ($- \text{grd } (\text{step } \text{SetMark})$))

apply (*simp add: Hoare-dgr-def*)

apply (*rule-tac x = SetMarkInvTerm in exI*)

apply (*subgoal-tac SetMarkInv = \sqcup range SetMarkInvTerm*)

apply *simp*

apply *safe*

apply (*simp-all add: SetMark-def SUP-L-P-def*

less-pair-def less-I-def hoare-choice)

apply (*simp-all add: fun-eq-iff*)

apply *safe*

apply (*unfold SetMarkInv-def*)

by *auto*

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

end

4 Marking Using a Stack

theory *StackMark*
imports *SetMark DataRefinementIBP.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 successors, 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.

4.1 Transitions

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

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

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

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

definition
 $Q5'-a \equiv [\lambda (stk, mrk) . \{(stk', mrk') . stk = [] \wedge mrk' = mrk\}:]$

4.2 Invariants

definition
 $Init' \equiv UNIV$

definition

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

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

4.3 Data refinement relations

definition

$$R1-a \equiv \{ : stk, mrk \rightsquigarrow X, mrk' . mrk' = mrk : \}$$
definition

$$R2-a \equiv \{ : stk, mrk \rightsquigarrow X, mrk' . X = set\ stk \wedge (stk, mrk) \in Loop' \wedge mrk' = mrk : \}$$

lemma *[simp]*: $R1-a \in Apply.Disjunctive$
by (*simp add: R1-a-def*)

lemma *[simp]*: $R2-a \in Apply.Disjunctive$ **by** (*simp add: R2-a-def*)

definition *[simp]*:
$$R-a\ i = (case\ i\ of \\ I.init \Rightarrow R1-a \mid \\ I.loop \Rightarrow R2-a \mid \\ I.final \Rightarrow R1-a)$$

lemma *[simp]*: *Disjunctive-fun* $R-a$ **by** (*simp add: Disjunctive-fun-def*)

definition

$$angelic-fun\ r = (\lambda\ i . \{ : r\ i : \})$$
definition (*in graph*)
$$StackMark-a = (\lambda\ (i, j) . (case\ (i, j)\ of \\ (I.init, I.loop) \Rightarrow Q1'-a \sqcap Q2'-a \mid \\ (I.loop, I.loop) \Rightarrow Q3'-a \sqcap Q4'-a \mid \\ (I.loop, I.final) \Rightarrow Q5'-a \mid \\ - \Rightarrow \top))$$

4.4 Data refinement of the transitions

theorem (*in graph*) *init-nil* *[simp]*:

DataRefinement ($\{ .Init. \} o Q1-a$) $R1-a\ R2-a\ Q1'-a$
by (*simp add: data-refinement-hoare hoare-demonic Q1'-a-def Init-def*)

Loop'-def R1-a-def R2-a-def Q1-a-def angelic-def subset-eq)

theorem (in graph) *init-root* [simp]:

DataRefinement ($\{.Init.\} \circ Q2\text{-}a$) *R1-a R2-a Q2'-a*

by (*simp add: data-refinement-hoare hoare-demonic Q2'-a-def Init-def Loop'-def R1-a-def R2-a-def Q2-a-def angelic-def subset-eq*)

theorem (in graph) *step1* [simp]:

DataRefinement ($\{.Loop.\} \circ Q3\text{-}a$) *R2-a R2-a Q3'-a*

apply (*simp add: data-refinement-hoare hoare-demonic Loop-def Loop'-def R2-a-def Q3-a-def Q3'-a-def angelic-def subset-eq*)

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

by (*metis List.set-simps(2) hd-in-set distinct.simps(2)*)

theorem (in graph) *step2* [simp]:

DataRefinement ($\{.Loop.\} \circ Q4\text{-}a$) *R2-a R2-a Q4'-a*

apply (*simp add: data-refinement-hoare hoare-demonic Loop-def Loop'-def R2-a-def Q4-a-def Q4'-a-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 ($\{.Loop.\} \circ Q5\text{-}a$) *R2-a R1-a Q5'-a*

apply (*simp add: data-refinement-hoare hoare-demonic Loop-def*

Loop'-def R2-a-def R1-a-def Q5-a-def Q5'-a-def angelic-def subset-eq)

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

4.5 Diagram data refinement

lemma *assert-comp-choice*: $\{.p.\} \circ (S \sqcap T) = (\{.p.\} \circ S) \sqcap (\{.p.\} \circ T)$

apply (*rule antisym*)

apply (*simp-all add: fun-eq-iff assert-def le-fun-def inf-fun-def inf-assoc*)

apply *safe*

apply (*rule-tac y = S x \sqcap T x in order-trans*)

apply (*rule inf-le2*)

apply *simp*

apply (*rule-tac y = S x \sqcap T x in order-trans*)

apply (*rule inf-le2*)

apply *simp*

apply (*rule-tac y = S x \sqcap (p \sqcap T x) in order-trans*)

apply (*rule inf-le2*)

apply *simp*

apply (*rule-tac y = S x \sqcap (p \sqcap T x) in order-trans*)

apply (*rule inf-le2*)

apply (*rule-tac y = p \sqcap T x in order-trans*)

apply (*rule inf-le2*)

by *simp*

```

theorem (in graph) StackMark-DataRefinement [simp]:
  DgrDataRefinement2 SetMarkInv SetMark R-a StackMark-a
  by (simp add: DgrDataRefinement2-def StackMark-a-def SetMark-def demonic-sup-inf

    SetMarkInv-def data-refinement-choice2 assert-comp-choice)

```

4.6 Diagram correctness

```

theorem (in graph) StackMark-correct:
  Hoare-dgr (R-a .. SetMarkInv) StackMark-a ((R-a .. SetMarkInv)  $\sqcap$  ( $-$  grd (step
  (StackMark-a))))
  apply (rule-tac D = SetMark in Diagram-DataRefinement2)
  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 $x \in \text{node}$, *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  $\Rightarrow$  'node  $\Rightarrow$  'node
  fixes label0 :: 'node  $\Rightarrow$  '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 ⊆ link?: 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 ∈ set l ∧ stack lnk lbl (lnk (lbl x) x) l)

```

```

lemma label-out-range0 [simp]:
  ¬ x ∈ set S ⇒ label-0 lbl S x = lbl x

```

apply (*rule-tac* $P = \forall \text{ label } . \neg x \in \text{set } S \longrightarrow \text{label-0 label } S x = \text{label } x$ **in** *mp*)
by (*simp*, *induct-tac* *S*, *auto*)

lemma *link-out-range0* [*simp*]:

$\neg x \in \text{set } S \implies \text{link-0 link label } p S i x = \text{link } i x$

apply (*rule-tac* $P = \forall \text{ link } p . \neg x \in \text{set } S \longrightarrow \text{link-0 link label } p S i x = \text{link } i x$ **in** *mp*)

by (*simp*, *induct-tac* *S*, *auto*)

lemma *link-out-range* [*simp*]: $\neg x \in \text{set } S \implies \text{link-0 link (label}(x := y)) p S = \text{link-0 link label } p S$

apply (*rule-tac* $P = \forall \text{ link } p . \neg x \in \text{set } S \longrightarrow \text{link-0 link (label}(x := y)) p S = \text{link-0 link label } p S$ **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*]: $\neg p \in \text{set } S \implies \text{stack (link}(i := (\text{link } i)(p := q))) \text{label } x S = \text{stack link label } x S$

apply (*rule-tac* $P = \forall \text{ link } x . \neg p \in \text{set } S \longrightarrow \text{stack (link}(i := (\text{link } i)(p := q))) \text{label } x S = \text{stack link label } x S$ **in** *mp*)

by (*simp*, *induct-tac* *S*, *auto*)

lemma *stack-out-label-range* [*simp*]: $\neg p \in \text{set } S \implies \text{stack link (label}(p := q)) x S = \text{stack link label } x S$

apply (*rule-tac* $P = \forall \text{ link } x . \neg p \in \text{set } S \longrightarrow \text{stack link (label}(p := q)) x S = \text{stack link label } x S$ **in** *mp*)

by (*simp*, *induct-tac* *S*, *auto*)

definition

$g \text{ mrk } \text{lbl } \text{ptr } x \equiv \text{ptr } x \neq \text{nil} \wedge \text{ptr } x \notin \text{mrk} \wedge \text{lbl } x = \text{none}$

lemma *g-cong* [*cong*]: $\text{mrk} = \text{mrk1} \implies \text{lbl} = \text{lbl1} \implies \text{ptr} = \text{ptr1} \implies x = x1 \implies$

$\text{pointer.g } n \text{ m } \text{mrk } \text{lbl } \text{ptr } x = \text{pointer.g } n \text{ m } \text{mrk1 } \text{lbl1 } \text{ptr1 } x1$

by *simp*

5.1 Transitions

definition

$Q1''\text{-a} \equiv [: p, t, \text{lnk}, \text{lbl}, \text{mrk} \rightsquigarrow p', t', \text{lnk}', \text{lbl}', \text{mrk}' .$

$\text{root} = \text{nil} \wedge p' = \text{nil} \wedge t' = \text{nil} \wedge \text{lnk}' = \text{lnk} \wedge \text{lbl}' = \text{lbl} \wedge \text{mrk}' = \text{mrk} :]$

definition

$Q2''\text{-a} \equiv [: p, t, \text{lnk}, \text{lbl}, \text{mrk} \rightsquigarrow p', t', \text{lnk}', \text{lbl}', \text{mrk}' .$

$\text{root} \neq \text{nil} \wedge p' = \text{root} \wedge t' = \text{nil} \wedge \text{lnk}' = \text{lnk} \wedge \text{lbl}' = \text{lbl} \wedge \text{mrk}' = \text{mrk} \cup \{\text{root}\} :]$

definition

$$\begin{aligned}
Q3''-a \equiv & [: p, t, lnk, lbl, mrk \rightsquigarrow p', t', lnk', lbl', mrk' . \\
& p \neq nil \wedge \\
& (\exists i . g \text{ mrk } lbl (lnk i) p \wedge \\
& p' = lnk i p \wedge t' = p \wedge lnk' = lnk(i := (lnk i)(p := t)) \wedge lbl' = lbl(p \\
:= i) \wedge \\
& mrk' = mrk \cup \{lnk i p\}) :]
\end{aligned}$$

definition

$$\begin{aligned}
Q4''-a \equiv & [: p, t, lnk, lbl, mrk \rightsquigarrow p', t', lnk', lbl', mrk' . \\
& p \neq nil \wedge \\
& (\forall i . \neg g \text{ mrk } lbl (lnk i) p) \wedge t \neq nil \wedge \\
& p' = t \wedge t' = lnk (lbl t) t \wedge lnk' = lnk(lbl t := (lnk (lbl t))(t := p)) \\
& \wedge lbl' = lbl(t := none) \wedge mrk' = mrk:]
\end{aligned}$$

definition

$$\begin{aligned}
Q5''-a \equiv & [: p, t, lnk, lbl, mrk \rightsquigarrow p', t', lnk', lbl', mrk' . \\
& p \neq nil \wedge \\
& (\forall i . \neg g \text{ mrk } lbl (lnk i) p) \wedge t = nil \wedge \\
& p' = nil \wedge t' = t \wedge lnk' = lnk \wedge lbl' = lbl \wedge mrk' = mrk:]
\end{aligned}$$

definition

$$\begin{aligned}
Q6''-a \equiv & [: p, t, lnk, lbl, mrk \rightsquigarrow p', t', lnk', lbl', mrk' . p = nil \wedge \\
& p' = p \wedge t' = t \wedge lnk' = lnk \wedge lbl' = lbl \wedge mrk' = mrk :]
\end{aligned}$$

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

$$R1'-a \equiv \{ : p, t, lnk, lbl, mrk \rightsquigarrow stk, mrk' . (p, t, lnk, lbl, mrk) \in Init'' \wedge mrk' = mrk : \}$$

definition

$$\begin{aligned}
R2'-a \equiv & \{ : p, t, lnk, lbl, mrk \rightsquigarrow stk, mrk' . \\
& p = head \text{ stk} \wedge \\
& t = head (\text{tail } stk) \wedge \\
& stack \text{ lnk } lbl t (\text{tail } stk) \wedge \\
& link0 = link-0 \text{ lnk } lbl p (\text{tail } stk) \wedge \\
& label0 = label-0 \text{ lbl } (\text{tail } stk) \wedge \\
& \neg nil \in set \text{ stk} \wedge
\end{aligned}$$

$mrk' = mrk : \}$

lemma $[simp]$: $R1'-a \in Apply.Disjunctive$ **by** ($simp$ add : $R1'-a-def$)

lemma $[simp]$: $R2'-a \in Apply.Disjunctive$ **by** ($simp$ add : $R2'-a-def$)

definition $[simp]$:

$R'-a$ $i = (case$ i of
 $I.init \Rightarrow R1'-a$ $|$
 $I.loop \Rightarrow R2'-a$ $|$
 $I.final \Rightarrow R1'-a)$

lemma $[simp]$: $Disjunctive-fun$ $R'-a$ **by** ($simp$ add : $Disjunctive-fun-def$)

5.4 Diagram

definition

$LinkMark = (\lambda (i, j) . (case (i, j) of$
 $(I.init, I.loop) \Rightarrow Q1''-a \sqcap Q2''-a$ $|$
 $(I.loop, I.loop) \Rightarrow Q3''-a \sqcap (Q4''-a \sqcap Q5''-a)$ $|$
 $(I.loop, I.final) \Rightarrow Q6''-a$ $|$
 $- \Rightarrow \top))$

definition $[simp]$:

$LinkMarkInv$ $i = (case$ i of
 $I.init \Rightarrow Init''$ $|$
 $I.loop \Rightarrow Loop''$ $|$
 $I.final \Rightarrow Final'')$

5.5 Data refinement of the transitions

theorem $init1-a$ $[simp]$:

$DataRefinement$ $(\{.Init'\} o Q1'-a) R1'-a R2'-a Q1''-a$
by ($simp$ add : $data-refinement-hoare$ $hoare-demonic$ $Q1''-a-def$ $Init'-def$ $Init''-def$
 $Loop''-def$ $R1'-a-def$ $R2'-a-def$ $Q1'-a-def$ $tail-def$ $head-def$ $angelic-def$ $subset-eq$)

theorem $init2-a$ $[simp]$:

$DataRefinement$ $(\{.Init'\} o Q2'-a) R1'-a R2'-a Q2''-a$
by ($simp$ add : $data-refinement-hoare$ $hoare-demonic$ $Q2''-a-def$ $Init'-def$ $Init''-def$
 $Loop''-def$ $R1'-a-def$ $R2'-a-def$ $Q2'-a-def$ $tail-def$ $head-def$ $angelic-def$ $subset-eq$)

theorem $step1-a$ $[simp]$:

$DataRefinement$ $(\{.Loop'\} o Q3'-a) R2'-a R2'-a Q3''-a$
apply ($simp$ add : $data-refinement-hoare$ $hoare-demonic$ $Q3''-a-def$ $Init'-def$ $Init''-def$
 $Loop'-def$ $R1'-a-def$ $Q3'-a-def$ $tail-def$ $head-def$ $angelic-def$ $subset-eq$)
apply ($unfold$ $next-def$)
apply ($simp$ add : $R2'-a-def$)

```

apply (simp add: data-refinement-hoare)
apply (simp-all add: R2'-a-def angelic-def hoare-demonic simp-eq-emptyset)
apply auto
apply (rule-tac x = aa 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 ≠ y ⇒ (if y = x then a else b) = b
apply (case-tac y ≠ x)
apply simp-all
done

```

```

theorem step2-a [simp]:
  DataRefinement ({.Loop'.} o Q4'-a) R2'-a R2'-a Q4''-a
apply (simp add: data-refinement-hoare hoare-demonic Q4''-a-def Init'-def Init''-def

    Loop'-def Q4'-a-def tail-def head-def angelic-def subset-eq)
apply (unfold next-def)
apply (simp add: R2'-a-def)
apply (simp add: data-refinement-hoare)
apply (simp-all add: R2'-a-def angelic-def hoare-demonic simp-eq-emptyset)
apply (simp-all add: neq-Nil-conv)
apply (unfold g-def)
apply (simp add: head-def)
apply safe
apply auto [1]
apply auto [1]
apply (case-tac ysa)
apply simp-all
apply safe
apply (case-tac ab ya = i)
by auto

```

```

lemma setsimp: a = c ⇒ (x ∈ a) = (x ∈ c)
apply simp
done

```

```

theorem step3-a [simp]:

```

DataRefinement ($\{.Loop'\}$ o $Q4'-a$) $R2'-a$ $R2'-a$ $Q5''-a$
apply (*simp add: data-refinement-hoare hoare-demonic Q5''-a-def Init'-def Init''-def*
Loop'-def Q4'-a-def angelic-def subset-eq)
apply (*unfold R2'-a-def*)
apply (*unfold next-def*)
apply (*simp add: data-refinement-hoare hoare-demonic angelic-def subset-eq*
simp-eq-emptyset g-def head-def tail-def)
by *auto*

theorem *final-a* [*simp*]:
DataRefinement ($\{.Loop'\}$ o $Q5'-a$) $R2'-a$ $R1'-a$ $Q6''-a$
apply (*simp add: data-refinement-hoare hoare-demonic Q6''-a-def Init'-def Init''-def*
Loop'-def R2'-a-def R1'-a-def Q5'-a-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

lemma *apply-fun-index* [*simp*]: $(r \ .. P) i = (r i) (P i)$ **by** (*simp add: apply-fun-def*)

lemma [*simp*]: *Disjunctive-fun* ($r::('c \Rightarrow 'a::\text{complete-lattice} \Rightarrow 'b::\text{complete-lattice})$)
 \implies *mono-fun* r
by (*simp add: Disjunctive-fun-def mono-fun-def*)

theorem *LinkMark-DataRefinement-a* [*simp*]:
DgrDataRefinement2 ($R-a \ .. \text{SetMarkInv}$) *StackMark-a* $R'-a$ *LinkMark*
apply (*rule-tac P = StackMarkInv in DgrDataRefinement-mono*)
apply (*simp add: le-fun-def SetMarkInv-def angelic-def*
R1-a-def R2-a-def Init'-def Final'-def)
apply *safe*
apply *simp*
apply (*simp add: DgrDataRefinement2-def dgr-demonic-def LinkMark-def*
StackMark-a-def demonic-sup-inf data-refinement-choice2 assert-comp-choice)
apply (*rule data-refinement-choice2*)
apply *simp-all*
apply (*rule data-refinement-choice1*)
by *simp-all*

lemma [*simp*]: *mono* $Q1'-a$ **by** (*simp add: Q1'-a-def*)

lemma [*simp*]: *mono* $Q2'-a$ **by** (*simp add: Q2'-a-def*)

lemma [*simp*]: *mono* $Q3'-a$ **by** (*simp add: Q3'-a-def*)

lemma [*simp*]: *mono* $Q4'-a$ **by** (*simp add: Q4'-a-def*)

lemma [*simp*]: *mono* $Q5'-a$ **by** (*simp add: Q5'-a-def*)

```

lemma [simp]: dmono StackMark-a
  apply (unfold dmono-def StackMark-a-def)
  by simp

```

5.7 Diagram correctness

```

theorem LinkMark-correct:
  Hoare-dgr (R'-a .. (R-a .. SetMarkInv)) LinkMark ((R'-a .. (R-a .. SetMarkInv))
   $\square$  (- grd (step LinkMark))
  apply (rule-tac D = StackMark-a in Diagram-DataRefinement2)
  apply simp-all
  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, $atom : node \rightarrow bool$ 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 initial values of the variables *left*, *right*, and *atom* in *left0*, *right0*, and *atom0* respectively.

```

datatype Index = none | 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  $\subseteq$  dsw?: pointer nil root none::Index link0 label0
proof qed auto

```

context *classical*
begin

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

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

6.1 Transitions

definition
 $QQ1\text{-}a \equiv [: p, t, \text{left}, \text{right}, \text{atom}, \text{mrk} \rightsquigarrow p', t', \text{left}', \text{right}', \text{atom}', \text{mrk}' .$
 $\text{root} = \text{nil} \wedge p' = \text{nil} \wedge t' = \text{nil} \wedge \text{mrk}' = \text{mrk} \wedge \text{left}' = \text{left}$
 $\wedge \text{right}' = \text{right} \wedge \text{atom}' = \text{atom} :]$

definition
 $QQ2\text{-}a \equiv [: p, t, \text{left}, \text{right}, \text{atom}, \text{mrk} \rightsquigarrow p', t', \text{left}', \text{right}', \text{atom}', \text{mrk}' .$
 $\text{root} \neq \text{nil} \wedge p' = \text{root} \wedge t' = \text{nil} \wedge \text{mrk}' = \text{mrk} \cup \{\text{root}\}$
 $\wedge \text{left}' = \text{left} \wedge \text{right}' = \text{right} \wedge \text{atom}' = \text{atom} :]$

definition
 $QQ3\text{-}a \equiv [: p, t, \text{left}, \text{right}, \text{atom}, \text{mrk} \rightsquigarrow p', t', \text{left}', \text{right}', \text{atom}', \text{mrk}' .$
 $p \neq \text{nil} \wedge gg \text{ mrk atom left } p \wedge$
 $p' = \text{left } p \wedge t' = p \wedge \text{mrk}' = \text{mrk} \cup \{\text{left } p\} \wedge$
 $\text{left}' = \text{left}(p := t) \wedge \text{right}' = \text{right} \wedge \text{atom}' = \text{atom} :]$

definition
 $QQ4\text{-}a \equiv [: p, t, \text{left}, \text{right}, \text{atom}, \text{mrk} \rightsquigarrow p', t', \text{left}', \text{right}', \text{atom}', \text{mrk}' .$
 $p \neq \text{nil} \wedge gg \text{ mrk atom right } p \wedge$
 $p' = \text{right } p \wedge t' = p \wedge \text{mrk}' = \text{mrk} \cup \{\text{right } p\} \wedge$
 $\text{left}' = \text{left} \wedge \text{right}' = \text{right}(p := t) \wedge \text{atom}' = \text{atom}(p := \text{True}) :]$

definition
 $QQ5\text{-}a \equiv [: p, t, \text{left}, \text{right}, \text{atom}, \text{mrk} \rightsquigarrow p', t', \text{left}', \text{right}', \text{atom}', \text{mrk}' .$
 $p \neq \text{nil} \wedge \text{--- not needed in the proof}$
 $\neg gg \text{ mrk atom left } p \wedge \neg gg \text{ mrk atom right } p \wedge$
 $t \neq \text{nil} \wedge \neg \text{atom } t \wedge$
 $p' = t \wedge t' = \text{left } t \wedge \text{mrk}' = \text{mrk} \wedge$
 $\text{left}' = \text{left}(t := p) \wedge \text{right}' = \text{right} \wedge \text{atom}' = \text{atom} :]$

definition
 $QQ6\text{-}a \equiv [: p, t, \text{left}, \text{right}, \text{atom}, \text{mrk} \rightsquigarrow p', t', \text{left}', \text{right}', \text{atom}', \text{mrk}' .$
 $p \neq \text{nil} \wedge \text{--- not needed in the proof}$
 $\neg gg \text{ mrk atom left } p \wedge \neg gg \text{ mrk atom right } p \wedge$
 $t \neq \text{nil} \wedge \text{atom } t \wedge$

$$\begin{aligned}
p' &= t \wedge t' = \text{right } t \wedge \text{mrk}' = \text{mrk} \wedge \\
\text{left}' &= \text{left} \wedge \text{right}' = \text{right}(t := p) \wedge \text{atom}' = \text{atom}(t := \text{False}) :]
\end{aligned}$$

definition

$$\begin{aligned}
\text{QQ7-a} &\equiv [: p, t, \text{left}, \text{right}, \text{atom}, \text{mrk} \rightsquigarrow p', t', \text{left}', \text{right}', \text{atom}', \text{mrk}' . \\
& p \neq \text{nil} \wedge \\
& \neg \text{gg } \text{mrk } \text{atom } \text{left } p \wedge \neg \text{gg } \text{mrk } \text{atom } \text{right } p \wedge \\
& t = \text{nil} \wedge \\
& p' = \text{nil} \wedge t' = t \wedge \text{mrk}' = \text{mrk} \wedge \\
& \text{left}' = \text{left} \wedge \text{right}' = \text{right} \wedge \text{atom}' = \text{atom} :]
\end{aligned}$$

definition

$$\begin{aligned}
\text{QQ8-a} &\equiv [: p, t, \text{left}, \text{right}, \text{atom}, \text{mrk} \rightsquigarrow p', t', \text{left}', \text{right}', \text{atom}', \text{mrk}' . \\
& p = \text{nil} \wedge p' = p \wedge t' = t \wedge \text{mrk}' = \text{mrk} \wedge \text{left}' = \text{left} \wedge \text{right}' = \text{right} \wedge \text{atom}' \\
& = \text{atom} :]
\end{aligned}$$

7 Data refinement relation

definition

$$\begin{aligned}
\text{RR-a} &\equiv \{ : p, t, \text{left}, \text{right}, \text{atom}, \text{mrk} \rightsquigarrow p', t', \text{lnk}, \text{lbl}, \text{mrk}' . \\
& \text{lnk } \text{none} = \text{left} \wedge \text{lnk } \text{some} = \text{right} \wedge \\
& \text{lbl} = (\lambda x . \text{if } \text{atom } x \text{ then } \text{some } \text{else } \text{none}) \wedge \\
& p' = p \wedge t' = t \wedge \text{mrk}' = \text{mrk} : \}
\end{aligned}$$

definition [*simp*]:

$$R''\text{-a } i = \text{RR-a}$$

definition

$$\begin{aligned}
\text{ClassicMark} &= (\lambda (i, j) . (\text{case } (i, j) \text{ of} \\
& (I.\text{init}, I.\text{loop}) \Rightarrow \text{QQ1-a} \sqcap \text{QQ2-a} \mid \\
& (I.\text{loop}, I.\text{loop}) \Rightarrow (\text{QQ3-a} \sqcap \text{QQ4-a}) \sqcap ((\text{QQ5-a} \sqcap \text{QQ6-a}) \sqcap \text{QQ7-a}) \mid \\
& (I.\text{loop}, I.\text{final}) \Rightarrow \text{QQ8-a} \mid \\
& - \Rightarrow \top))
\end{aligned}$$

7.1 Data refinement of the transitions

theorem *init1-a* [*simp*]:

$$\begin{aligned}
& \text{DataRefinement } (\{.\text{Init}''.\} \circ \text{Q1}''\text{-a}) \text{RR-a } \text{RR-a } \text{QQ1-a} \\
& \text{by } (\text{simp add: data-refinement-hoare hoare-demonic angelic-def QQ1-a-def Q1}''\text{-a-def} \\
& \text{RR-a-def} \\
& \text{Init}''\text{-def subset-eq})
\end{aligned}$$

theorem *init2-a* [*simp*]:

$$\begin{aligned}
& \text{DataRefinement } (\{.\text{Init}''.\} \circ \text{Q2}''\text{-a}) \text{RR-a } \text{RR-a } \text{QQ2-a} \\
& \text{by } (\text{simp add: data-refinement-hoare hoare-demonic angelic-def QQ2-a-def Q2}''\text{-a-def} \\
& \text{RR-a-def} \\
& \text{Init}''\text{-def subset-eq})
\end{aligned}$$

lemma *index-simp*:

$(u = v) = (u \text{ none} = v \text{ none} \wedge u \text{ some} = v \text{ some})$
by (*safe, rule ext, case-tac x, auto*)

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

theorem *step2-a* [*simp*]:
DataRefinement ($\{.Loop''.\}$ *o* $Q3''\text{-}a$) *RR-a* *RR-a* *QQ4-a*
apply (*simp add: data-refinement-hoare hoare-demonic angelic-def QQ4-a-def*
 $Q3''\text{-}a\text{-def}$ *RR-a-def*
Loop''-def subset-eq g-def gg-def simp-eq-emptyset)
apply *safe*
apply (*rule-tac* $x=\lambda x . \text{if } x = \text{none then } ab \text{ none else } (ab \text{ some})(a := aa)$ **in**
 exI)
apply *simp*
apply (*rule-tac* $x=\text{some}$ **in** exI)
apply (*simp add: index-simp*)
apply (*rule ext*)
apply *auto*
done

theorem *step3-a* [*simp*]:
DataRefinement ($\{.Loop''.\}$ *o* $Q4''\text{-}a$) *RR-a* *RR-a* *QQ5-a*
apply (*simp add: data-refinement-hoare hoare-demonic angelic-def QQ5-a-def*
 $Q4''\text{-}a\text{-def}$ *RR-a-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*: $(x \in (\text{if } b \text{ then } A \text{ else } B)) = ((b \wedge x \in A) \vee (\neg b \wedge x \in B))$
by *auto*

theorem *step4-a* [*simp*]:
DataRefinement ($\{.Loop''.\}$ *o* $Q4''\text{-}a$) *RR-a* *RR-a* *QQ6-a*
apply (*simp add: data-refinement-hoare hoare-demonic angelic-def RR-a-def QQ6-a-def*
 $Q4''\text{-}a\text{-def}$)

```

    Loop''-def subset-eq simp-eq-emptyset g-def gg-def if-set-elim)
apply (simp add: ext)
apply safe
apply (case-tac i)
apply simp-all
apply (case-tac i)
apply simp-all
apply (case-tac i)
apply simp-all
apply (case-tac i)
by simp-all

```

```

theorem step5-a [simp]:
  DataRefinement ({.Loop''.} o Q5''-a) RR-a RR-a QQ7-a
apply (simp add: data-refinement-hoare hoare-demonic angelic-def Q5''-a-def
  QQ7-a-def
    Loop''-def subset-eq RR-a-def simp-eq-emptyset)
apply safe
apply (simp-all add: g-def gg-def)
apply (case-tac i)
by auto

```

```

theorem final-step-a [simp]:
  DataRefinement ({.Loop''.} o Q6''-a) RR-a RR-a QQ8-a
by (simp add: data-refinement-hoare hoare-demonic angelic-def Q6''-a-def QQ8-a-def
    Loop''-def subset-eq RR-a-def simp-eq-emptyset)

```

7.2 Diagram data refinement

```

lemma [simp]: mono RR-a by (simp add: RR-a-def)
lemma [simp]: RR-a ∈ Apply.Disjunctive by (simp add: RR-a-def)
lemma [simp]: Disjunctive-fun R''-a by (simp add: Disjunctive-fun-def)

```

```

lemma [simp]: mono-fun R''-a by simp

```

```

lemma [simp]: mono Q1''-a by (simp add: Q1''-a-def)
lemma [simp]: mono Q2''-a by (simp add: Q2''-a-def)
lemma [simp]: mono Q3''-a by (simp add: Q3''-a-def)
lemma [simp]: mono Q4''-a by (simp add: Q4''-a-def)
lemma [simp]: mono Q5''-a by (simp add: Q5''-a-def)
lemma [simp]: mono Q6''-a by (simp add: Q6''-a-def)

```

```

lemma [simp]: dmono LinkMark
apply (unfold dmono-def LinkMark-def)
by simp

```

```

theorem ClassicMark-DataRefinement-a [simp]:
  DgrDataRefinement2 (R'-a .. (R-a .. SetMarkInv)) LinkMark R''-a ClassicMark

```

```

apply (rule-tac  $P = \text{LinkMarkInv}$  in DgrDataRefinement-mono)
apply (simp add: le-fun-def SetMarkInv-def
  angelic-def  $R1'-a$ -def  $R2'-a$ -def  $\text{Init}''$ -def  $\text{Loop}''$ -def  $\text{Final}''$ -def)
apply auto
apply (simp add: DgrDataRefinement2-def dgr-demonic-def ClassicMark-def LinkMark-def
  demonic-sup-inf data-refinement-choice2 assert-comp-choice)
apply (rule data-refinement-choice2)
apply simp
apply (rule data-refinement-choice1)
apply simp-all
apply (rule data-refinement-choice2)
apply simp-all
apply (rule data-refinement-choice1)
by simp-all

```

7.3 Diagram corectness

theorem *ClassicMark-correct-a* [simp]:
 $\text{Hoare-dgr } (R''-a \dots (R'-a \dots (R-a \dots \text{SetMarkInv}))) \text{ } \text{ClassicMark}$
 $((R''-a \dots (R'-a \dots (R-a \dots \text{SetMarkInv}))) \sqcap (\neg \text{grd } (\text{step } \text{ClassicMark})))$
apply (rule-tac $D = \text{LinkMark}$ **in** *Diagram-DataRefinement2*)
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 indeed 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 easely 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

$$\begin{aligned} \text{ClassicInit} &= \{(p, t, \text{left}, \text{right}, \text{atom}, \text{mrk}) . \\ &\quad \text{atom} = \text{atom0} \wedge \text{left} = \text{left0} \wedge \text{right} = \text{right0} \wedge \\ &\quad \text{finite } (\neg \text{mrk}) \wedge \text{mrk} = \{\}\} \end{aligned}$$

definition

$$\text{ClassicFinal} = \{(p, t, \text{left}, \text{right}, \text{atom}, \text{mrk}) .$$

$$\text{atom} = \text{atom0} \wedge \text{left} = \text{left0} \wedge \text{right} = \text{right0} \wedge \\ \text{mrk} = \text{reach root}\}$$

theorem [simp]:

$\text{ClassicInit} \subseteq (\text{RR-a } (R1'-a (R1-a (\text{SetMarkInv } \text{init}))))$
apply (simp add: SetMarkInv-def)
apply (simp add: ClassicInit-def angelic-def RR-a-def R1'-a-def R1-a-def Init-def Init''-def)
apply safe
apply (unfold simp-eq-emptyset)
apply (simp add: link0-def label0-def)
apply (simp add: fun-eq-iff)
by (simp add: label0-def)

theorem [simp]:

$(\text{RR-a } (R1'-a (R1-a (\text{SetMarkInv } \text{final})))) \leq \text{ClassicFinal}$
apply (simp add: SetMarkInv-def)
apply (simp add: ClassicFinal-def angelic-def RR-a-def R1'-a-def R1-a-def Final-def Final''-def Init''-def label0-def link0-def)
apply (simp add: simp-eq-emptyset inf-fun-def)
apply auto
by (simp-all add: link0-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 $\text{ClassicPre } \text{loop} = \text{ClassicPre } \text{final} = \emptyset$.

definition [simp]:

$\text{ClassicPre } i = (\text{case } i \text{ of}$
 $I.\text{init} \Rightarrow \text{ClassicInit} \mid$
 $I.\text{loop} \Rightarrow \{\} \mid$
 $I.\text{final} \Rightarrow \{\})$

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]:

$\text{ClassicPost } i = (\text{case } i \text{ of}$
 $I.\text{init} \Rightarrow \{\} \mid$
 $I.\text{loop} \Rightarrow \{\} \mid$
 $I.\text{final} \Rightarrow \text{ClassicFinal})$

lemma exists-or:

$(\exists x . p x \vee q x) = ((\exists x . p x) \vee (\exists x . q x))$
by auto

lemma [simp]:

$(- \text{grd } (\text{step } \text{ClassicMark})) \text{init} = \{\}$

```

apply (simp add: grd-def step-def)
apply safe
apply simp
apply (drule-tac x = loop in spec)
by (simp add: ClassicMark-def QQ1-a-def QQ2-a-def demonic-def)

```

```

lemma [simp]: grd  $\top = \perp$ 
by (simp add: grd-def top-fun-def)

```

```

lemma [simp]:
  (– grd (step ClassicMark)) loop = {}
apply safe
apply simp
apply (frule-tac x = final in spec)
apply (drule-tac x = loop in spec)
apply (unfold ClassicMark-def QQ1-a-def QQ2-a-def QQ3-a-def QQ4-a-def QQ5-a-def
  QQ6-a-def QQ7-a-def QQ8-a-def)
apply simp
apply (case-tac a  $\neq$  nil)
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.

```

lemma [simp]: mono QQ1-a by (simp add: QQ1-a-def)
lemma [simp]: mono QQ2-a by (simp add: QQ2-a-def)
lemma [simp]: mono QQ3-a by (simp add: QQ3-a-def)
lemma [simp]: mono QQ4-a by (simp add: QQ4-a-def)
lemma [simp]: mono QQ5-a by (simp add: QQ5-a-def)
lemma [simp]: mono QQ6-a by (simp add: QQ6-a-def)
lemma [simp]: mono QQ7-a by (simp add: QQ7-a-def)
lemma [simp]: mono QQ8-a by (simp add: QQ8-a-def)

```

```

lemma [simp]: dmono ClassicMark
apply (unfold dmono-def ClassicMark-def)
by simp

```

```

theorem  $\models$  ClassicPre { | pt ClassicMark | } ClassicPost
apply (rule-tac P = (R''-a .. (R'-a .. (R-a .. SetMarkInv))) in hoare-pre)
apply (subst le-fun-def)
apply simp
apply (rule-tac Q = ((R''-a .. (R'-a .. (R-a .. SetMarkInv)))  $\sqcap$  (– grd (step
  ((ClassicMark)))))) in hoare-mono)
apply (simp-all add: hoare-dgr-correctness)
apply (rule le-funI)
apply (case-tac x)
apply (simp-all add: inf-fun-def del: uminus-apply)
apply (rule-tac y = RR-a (R1'-a (R1-a (SetMarkInv final))) in order-trans)

```

by *auto*
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, *Archive of Formal Proofs*. <http://isa-afp.org/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.