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

Viorel Preoteasa and Ralph-Johan Back

August 28, 2014

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 ............................................. 6
  3.2 Invariants ............................................. 6
  3.3 Diagram .................................................. 7
  3.4 Correctness of the transitions ......................... 8
  3.5 Diagram correctness ..................................... 8

4 Marking Using a Stack 9
  4.1 Transitions ............................................. 9
  4.2 Invariants ............................................. 10
  4.3 Data refinement relations ............................... 10
  4.4 Data refinement of the transitions ................... 11
  4.5 Diagram data refinement ................................ 11
  4.6 Diagram correctness .................................... 11
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 =
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**

\[\text{head } S \equiv (\text{if } S = [] \text{ then nil else } \text{hd } S)\]

**definition**

\[\text{tail } (S :: \text{'a list}) \equiv (\text{if } S = [] \text{ then } [] \text{ else } \text{tl } S)\]

**lemma [simp]**: \((\text{nil}, x) \in \text{next} = \text{False}\) (proof)

**lemma [simp]**: \((x, \text{nil}) \in \text{next} = \text{False}\) (proof)

**theorem head-not-nil [simp]**:

\[(\text{head } S \neq \text{nil}) = (\text{head } S = \text{hd } S \land \text{tail } S = \text{tl } S \land \text{hd } S \neq \text{nil} \land S \neq [])\] (proof)

**theorem nonempty-head [simp]**:

\[\text{head } (x \neq S) = x\] (proof)

**theorem nonempty-tail [simp]**:

\[\text{tail } (x \neq S) = S\] (proof)

**definition (in graph)**

\[\text{reach } x \equiv \{y . (x, y) \in \text{next}^* \land y \neq \text{nil}\}\]

**theorem (in graph) reach-nil [simp]**: \(\text{reach } \text{nil} = \{\}\) (proof)

**theorem (in graph) reach-next**: \(b \in \text{reach } a \implies (b, c) \in \text{next} \implies c \in \text{reach } a\) (proof)

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

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 $\text{next}$ on the nodes of the graph.

The diagram has only three ordered situation ($\text{init} > \text{loop} > \text{final}$). 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 $I = \text{init} | \text{loop} | \text{final}$

declare $I$.split [split]

instantiation $I :: \text{well-founded-transitive}$
begin

definition less-I-def: $i < j \equiv (j = \text{init} \land (i = \text{loop} \lor i = \text{final})) \lor (j = \text{loop} \land i = \text{final})$

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

instance ⟨proof⟩

end

The set $\text{path } S \text{ mrk}$ contains all reachable nodes from $S$ along paths with unmarked nodes.

lemma trasc-l-less: $x \neq y \implies (a, x) \in R^* \implies ((a,x) \in (R \cap (-\{y\}) \times (-\{y\}))^* \lor (y,x) \in R O (R \cap (-\{y\}) \times (-\{y\}))^*)$
lemma (in graph) add-set [simp]: \(x \neq y \Rightarrow x \in \text{path } S \text{ mrk } \Rightarrow x \in \text{path } (\text{insert } y \ S)\) (insert y mrk)
(proof)

lemma (in graph) add-set2: \(x \in \text{path } S \text{ mrk } \Rightarrow x \notin \text{path } (\text{insert } y \ S)\) (insert y mrk) \(\Rightarrow x = y\)
(proof)

lemma (in graph) del-stack [simp]: \((\forall \ y \ . \ (t, y) \in \text{next } \Rightarrow y \in \text{mrk}) \Rightarrow x \notin \text{mrk } \Rightarrow x \in \text{path } S \text{ mrk } \Rightarrow x \in \text{path } (S - \{t\}) \text{ mrk}\)
(proof)

lemma (in graph) init-set [simp]: \(x \in \text{reach root } \Rightarrow x \neq \text{root } \Rightarrow x \in \text{path } \{\text{root}\}\)
\{\text{root}\}
(proof)

lemma (in graph) init-set2: \(x \in \text{reach root } \Rightarrow x \notin \text{path } \{\text{root}\}\) \{\text{root}\} \(\Rightarrow x = \text{root}\)
(proof)

3.1 Transitions

definition (in graph)
\(Q_1\)-a \(\equiv \): \(X, \text{mrk } \Rightarrow X', \text{mrk}'\). (\text{root::'node}) = \text{nil } \land X' = \{\} \land \text{mrk}' = \text{mrk} :\)

definition (in graph)
\(Q_2\)-a \(\equiv \): \(X, \text{mrk } \Rightarrow X', \text{mrk}'\).
\((\text{root::'node}) \neq \text{nil } \land X' = \{\text{root::'node}\} \land \text{mrk}' = \{\text{root::'node}\} :\)

definition (in graph)
\(Q_3\)-a \(\equiv \): \(X, \text{mrk } \Rightarrow X', \text{mrk}'\).
\((\exists x \in X . \exists y . (x, y) \in \text{next } \land y \notin \text{mrk } \land X' = X \cup \{y\} \land \text{mrk}' = \text{mrk}\) \cup \{y\})\)

definition (in graph)
\(Q_4\)-a \(\equiv \): \(X, \text{mrk } \Rightarrow X', \text{mrk}'\).
\((\exists x \in X . (\forall y . (x, y) \in \text{next } \Rightarrow y \in \text{mrk}) \land X' = X - \{x\} \land \text{mrk}' = \text{mrk}\) :\)

definition (in graph)
\(Q_5\)-a \(\equiv \): \(X, \text{mrk } \Rightarrow X', \text{mrk}'\). \(X = \{\} \land \text{mrk} = \text{mrk}' :\)

3.2 Invariants

definition (in graph)
\(\text{Loop } \equiv \{ (X, \text{mrk}) . \land \text{fin } (\neg \text{mrk}) \land \text{fin } X \land X \subseteq \text{mrk } \land \text{mrk } \subseteq \text{reach root } \land \text{reach root } \cap \neg \text{mrk } \subseteq \text{path } X \text{ mrk}\} \)
definition
\( \text{trm} \equiv \lambda (X, \text{mrk}). 2 \ast \text{card} (-\text{mrk}) + \text{card} X \)

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

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

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

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

definition (in graph)
\( \text{Init} \equiv \{ (X::('node set), \text{mrk}::('node set)) . \text{finite} (-\text{mrk}) \land \text{mrk} = \{} \} \)

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

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

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

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

3.3 Diagram

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

lemma (in graph) \( \text{SetMark-dmono } [\text{simp}]: \)
3.4 Correctness of the transitions

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

\[\text{proof}\]

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

\[\text{proof}\]

**Lemma (in graph)** loop-loop-1-a [simp]: \( \vdash (\text{Loop} \cap \{ s . \text{trm} s = w \}) \{ \mid Q3-a \} \)

\[\text{Loop} \cap \{ s . \text{trm} s < w \}\]

\[\text{proof}\]

**Lemma (in graph)** loop-loop-2-a [simp]: \( \vdash (\text{Loop} \cap \{ s . \text{trm} s = w \}) \{ \mid Q4-a \} \)

\[\text{Loop} \cap \{ s . \text{trm} s < w \}\]

\[\text{proof}\]

**Lemma (in graph)** loop-final-a [simp]: \( \vdash (\text{Loop} \cap \{ s . \text{trm} s = w \}) \{ \mid Q5-a \} \)

Final

\[\text{proof}\]

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

\[\text{proof}\]

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

\[\text{proof}\]

**Lemma** sup-union [simp]: \( \text{Sup} (\text{range} A) i = (\bigcup w . A w i) \)

\[\text{proof}\]

**Lemma** forall-simp [simp]: \( (! a b . \forall x \in A . (a = (t x)) \rightarrow (h x) \vee b \neq u x) = (\forall x \in A . h x) \)

\[\text{proof}\]

**Lemma** forall-simp2 [simp]: \( (! a b . \forall x \in A . ! y . (a = t x y) \rightarrow (h x y) \rightarrow (g x y) \vee b \neq u x y) = (\forall x \in A . ! y . h x y \rightarrow g x y) \)

\[\text{proof}\]

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 \( \lambda (n::\text{nat}) (i :: I) . (i, n) \)

\[\text{proof}\]

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

\( \vdash \text{SetMarkInv} \{ \mid \text{pt SetMark} \} \text{SetMarkInvFinal} \)
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). Initially 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)
\[ Q1' \equiv \lambda (\text{stk}::(\text{node list}), \text{mrk}::(\text{node set})) . ((\text{stk}'::(\text{node list}), \text{mrk}') .\]
\[ \text{root} = \text{nil} \land \text{stk}' = [] \land \text{mrk}' = \text{mrk};] \]

definition (in graph)
\[ Q2' \equiv \lambda (\text{stk}::(\text{node list}), \text{mrk}::(\text{node set})) . ((\text{stk}', \text{mrk}') .\]
\[ \text{root} \neq \text{nil} \land \text{stk}' = [\text{root}] \land \text{mrk}' = \text{mrk} \cup \{\text{root}\};] \]

definition (in graph)
\[ Q3' \equiv \lambda (\text{stk}, \text{mrk}) . ((\text{stk}', \text{mrk}') . \text{stk} \neq [] \land (\exists y . (\text{hd stk}, y) \in \text{next} \land \]
\[ y \notin \text{mrk} \land \text{stk}' = y \# \text{stk} \land \text{mrk}' = \text{mrk} \cup \{y\};] \]

definition (in graph)
\[ Q4' \equiv \lambda (\text{stk}, \text{mrk}) . ((\text{stk}', \text{mrk}') . \text{stk} \neq [] \land \]
\[ (\forall y . (\text{hd stk}, y) \in \text{next} \rightarrow y \in \text{mrk}) \land \text{stk}' = \text{tl stk} \land \text{mrk}' = \text{mrk};] \]

definition (in graph)
\[ Q5' \equiv \lambda (\text{stk}, \text{mrk}) . ((\text{stk}', \text{mrk}') . \text{stk} = [] \land \text{mrk}' = \text{mrk};] \]

4.2 Invariants

definition 
\(Init' \equiv UNIV\)

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

definition 
\(Final' \equiv UNIV\)

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

4.3 Data refinement relations

definition 
\(R1-a \equiv \{ : stk, mrk \mapsto X, mrk' . mrk' = mrk : \}\)

definition 
\(R2-a \equiv \{ : stk, mrk \mapsto X, mrk' . X = \text{set } stk \land (stk, mrk) \in Loop' \land mrk' = mrk : \}\)

lemma [simp]: \(R1-a \in \text{Apply.Disjunctive}\) (proof)

lemma [simp]: \(R2-a \in \text{Apply.Disjunctive}\) (proof)

definition [simp]:
\(R-a i = (\text{case } i \text{ of}\)
\(I.init \Rightarrow R1-a |\)
\(I.loop \Rightarrow R2-a |\)
\(I.final \Rightarrow R1-a)\)

lemma [simp]: Disjunctive-fun R-a (proof)

definition 
angelic-fun \(r = (\lambda i . \{ :r i : \})\)

definition (in graph)
\(StackMark-a = (\lambda (i, j) . (\text{case } (i, j) \text{ of}\)
\(I.init, I.loop) \Rightarrow Q1'-a \sqcap Q2'-a |\)
\(I.loop, I.loop) \Rightarrow Q3'-a \sqcap Q4'-a |\)
\(I.loop, I.final) \Rightarrow Q5'-a |\)
\(- \Rightarrow T))\)
4.4 Data refinement of the transitions

```
theorem (in graph) init-nil [simp]:
  DataRefinement ({.Init.} o Q1-a) R1-a R2-a Q1'-a
  ⟨proof⟩

theorem (in graph) init-root [simp]:
  DataRefinement ({.Init.} o Q2-a) R1-a R2-a Q2'-a
  ⟨proof⟩

theorem (in graph) step1 [simp]:
  DataRefinement ({.Loop.} o Q3-a) R2-a R2-a Q3'-a
  ⟨proof⟩

theorem (in graph) step2 [simp]:
  DataRefinement ({.Loop.} o Q4-a) R2-a R2-a Q4'-a
  ⟨proof⟩

theorem (in graph) final [simp]:
  DataRefinement ({.Loop.} o Q5-a) R2-a R1-a Q5'-a
  ⟨proof⟩
```

4.5 Diagram data refinement

```
lemma assert-comp-choice: {.p.} o (S ∩ T) = (.p) o S ∩ (.p) o T
  ⟨proof⟩

theorem (in graph) StackMark-DataRefinement [simp]:
  DgrDataRefinement2 SetMarkInv SetMark R-a StackMark-a
  ⟨proof⟩
```

4.6 Diagram correctness

```
theorem (in graph) StackMark-correct:
  Hoare-dgr (R-a .. SetMarkInv) StackMark-a (((R-a .. SetMarkInv) ∩ (¬ grd (step
  (StackMark-a))))
  ⟨proof⟩
```

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 \( \text{label} \). The variable \( \text{label} : \text{node} \rightarrow \text{index} \) associates a label to every node and the variable \( \text{link} : \text{index} \rightarrow \text{node} \rightarrow \text{node} \) is a collection of pointer functions indexed by the set \( \text{index} \) of labels. For \( x \in \text{node} \), \( \text{link} i x \) is the successor node of \( x \) along the function \( \text{link} i \). In this context a node \( x \) is reachable if there exists a path from the root to \( x \) along the links \( \text{link} i \) such that all nodes in this path are not \( \text{nil} \) and they are labeled by a special label \( \text{none} \in \text{index} \).

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 \( \text{link} \) and \( \text{label} \).

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.

\[
\text{locale} \quad \text{pointer} = \text{node} +
\]

\[
\begin{align*}
\text{fixes} & \quad \text{none} :: \text{'index} \\
\text{fixes} & \quad \text{link}0 :: \text{'index} \Rightarrow \text{'node} \Rightarrow \text{'node} \\
\text{fixes} & \quad \text{label}0 :: \text{'node} \Rightarrow \text{'index}
\end{align*}
\]

\[
\text{assumes} \quad (\text{nil} :: \text{'node}) = \text{nil}
\]

begin
\[
\begin{align*}
\text{definition} \quad \text{next} &= \{(a, b) . (\exists i . \text{link}0 i a = b) \land a \neq \text{nil} \land b \neq \text{nil} \land \text{label}0 a = \text{none}\}
\end{align*}
\]

end

\[
\text{sublocale} \quad \text{pointer} \subseteq \text{link}: \text{graph} \text{nil root next}
\]

\[
\langle \text{proof} \rangle
\]

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

The next two recursive functions, \( \text{label}_0 \), \( \text{link}_0 \) are used to compute the initial values of the variables \( \text{label} \) and \( \text{link} \) from their current values.

\[
\text{context} \quad \text{pointer}
\text{begin}
\text{primrec}
\begin{align*}
\text{label}_0 :: \text{('node} \Rightarrow \text{'index}) \Rightarrow \text{('node list) \Rightarrow ('node} \Rightarrow \text{'index)} \quad \text{where} \\
\text{label}_0 \ 	ext{lbl} \ [] &= \text{lbl} \ |
\text{label}_0 \ 	ext{lbl} \ (x \ # \ l) &= \text{label}_0 \ (\text{lbl}(x := \text{none})) \ l
\end{align*}
\]

\[
\text{lemma} \quad \text{label-cong} \ [\text{cong}]: f = g \implies xs = ys \implies \text{pointer.label}_0 n f xs = \text{pointer.label}_0 n g ys
\]
\[
\langle \text{proof} \rangle
\]

12
The function stack defined below 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.

```plaintext
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)
```

lemmas

- label-out-range0 [simp]: ¬ x ∈ set S =⇒ label-0 lbl S x = lbl x
  ⟨proof⟩
- link-out-range0 [simp]: ¬ x ∈ set S =⇒ link-0 link label p S i x = link i x
  ⟨proof⟩
- link-out-range [simp]: ¬ x ∈ set S =⇒ link-0 link (label(x := y)) p S = link-0 link label p S
  ⟨proof⟩
- empty-stack [simp]: stack link label nil S = (S = [])
  ⟨proof⟩
- stack-out-link-range [simp]: ¬ p ∈ set S =⇒ stack (link(i := (link i)(p := q))) label x S = stack link label x S
  ⟨proof⟩
- stack-out-label-range [simp]: ¬ p ∈ set S =⇒ stack link (label(p := q)) x S = stack link label x S
  ⟨proof⟩

definition

g mrk lbl ptr x ≡ ptr x ≠ nil ∧ ptr x ∉ mrk ∧ lbl x = none

lemmas

- g-cong [cong]: mrk = mrk1 =⇒ lbl = lbl1 =⇒ ptr = ptr1 =⇒ x = x1
  ==> pointer.g n m mrk lbl ptr x = pointer.g n m mrk1 lbl1 ptr1 x1
  ⟨proof⟩
```
5.1 Transitions

definition \[ Q1'' \equiv \{ p, t, \lnk, \lbl, \mrk \to p', t', \lnk', \lbl', \mrk' \mid \]\n\[ \text{root} = \text{nil} \wedge p' = \text{nil} \wedge t' = \text{nil} \wedge \lnk' = \lnk \wedge \lbl' = \lbl \wedge \mrk' = \mrk \} \]

definition \[ Q2'' \equiv \{ p, t, \lnk, \lbl, \mrk \to p', t', \lnk', \lbl', \mrk' \mid \]\n\[ \text{root} \neq \text{nil} \wedge p' = \text{root} \wedge t' = \text{nil} \wedge \lnk' = \lnk \wedge \lbl' = \lbl \wedge \mrk' = \mrk \cup \{ \text{root} \} \} \]

definition \[ Q3'' \equiv \{ p, t, \lnk, \lbl, \mrk \to p', t', \lnk', \lbl', \mrk' \mid \]\n\[ p \neq \text{nil} \wedge \]
\[ (\exists i \cdot g \mrk \lbl (\lnk i) p) \wedge \]
\[ p' = \lnk i \wedge t' = p \wedge \lnk' = \lnk(i := (\lnk i)(p := t)) \wedge \lbl' = \lbl(p := i) \wedge \]
\[ \mrk' = \mrk \cup \{ \lnk i p \} \} \]

definition \[ Q4'' \equiv \{ p, t, \lnk, \lbl, \mrk \to p', t', \lnk', \lbl', \mrk' \mid \]\n\[ p \neq \text{nil} \wedge \]
\[ (\forall i \cdot \neg g \mrk \lbl (\lnk i) p) \wedge t \neq \text{nil} \wedge \]
\[ p' = \text{nil} \wedge t' = t \wedge \lnk' = \lnk \wedge \lbl' = \lbl \wedge \mrk' = \mrk \]}

definition \[ Q5'' \equiv \{ p, t, \lnk, \lbl, \mrk \to p', t', \lnk', \lbl', \mrk' \mid \]\n\[ p \neq \text{nil} \wedge \]
\[ (\forall i \cdot \neg g \mrk \lbl (\lnk i) p) \wedge t = \text{nil} \wedge \]
\[ p' = \text{nil} \wedge t' = t \wedge \lnk' = \lnk \wedge \lbl' = \lbl \wedge \mrk' = \mrk \]}

definition \[ Q6'' \equiv \{ p, t, \lnk, \lbl, \mrk \to p', t', \lnk', \lbl', \mrk' \mid p = \text{nil} \wedge \]
\[ p' = p \wedge t' = t \wedge \lnk' = \lnk \wedge \lbl' = \lbl \wedge \mrk' = \mrk \]}

5.2 Invariants

definition \[ \text{Init}'' \equiv \{ (p, t, \lnk, \lbl, \mrk) \mid \lnk = \lnk 0 \wedge \lbl = \lbl 0 \} \]

definition \[ \text{Loop}'' \equiv \text{UNIV} \]

definition \[ \text{Final}'' \equiv \text{Init}'' \]

5.3 Data refinement relations
\[ R1^{'-a} \equiv \{ : p, t, \text{lnk}, \text{lbl}, \text{mrk} \leadsto \text{stk}, \text{mrk}' : (p, t, \text{lnk}, \text{lbl}, \text{mrk}) \in \text{Init}'' \wedge \text{mrk}' = \text{mrk} : \}
\]

**Definition**
\[ R2^{'-a} \equiv \{ : p, t, \text{lnk}, \text{lbl}, \text{mrk} \leadsto \text{stk}, \text{mrk}' : \]
\begin{align*}
& p = \text{head stk} \wedge \\
& t = \text{head} (\text{tail stk}) \wedge \\
& \text{stack lnk lbl t (tail stk)} \wedge \\
& \text{link0} = \text{link-0 lnk lbl p (tail stk)} \wedge \\
& \text{label0} = \text{label-0 lbl (tail stk)} \wedge \\
& \neg \text{nil} \in \text{set stk} \wedge \\
& \text{mrk}' = \text{mrk} : \}
\end{align*}

**Lemma** \([\text{simp}]: R1^{'-a} \in \text{Apply.Disjunctive (proof)} \]

**Lemma** \([\text{simp}]: R2^{'-a} \in \text{Apply.Disjunctive (proof)} \]

**Definition** \([\text{simp}]: R^{'-a} i = (\text{case } i \text{ of} \]
\begin{align*}
& I.\text{init} \Rightarrow R1^{'-a} | \\
& I.\text{loop} \Rightarrow R2^{'-a} | \\
& I.\text{final} \Rightarrow R1^{'-a})
\end{align*}

**Lemma** \([\text{simp}]: \text{Disjunctive-fun } R^{'-a} \langle \text{proof} \rangle \]

### 5.4 Diagram

**Definition**
\[ \text{LinkMark} = (\lambda (i, j) . (\text{case } (i, j) \text{ of} \]
\begin{align*}
& (I.\text{init}, I.\text{loop}) \Rightarrow Q1'''^{'-a} \cap Q2'''^{'-a} | \\
& (I.\text{loop}, I.\text{loop}) \Rightarrow Q3'''^{'-a} \cap (Q4'''^{'-a} \cap Q5'''^{'-a}) | \\
& (I.\text{loop}, I.\text{final}) \Rightarrow Q6'''^{'-a} | \\
& - \Rightarrow \top))
\]

**Definition** \([\text{simp}]: \]
\[ \text{LinkMarkInv } i = (\text{case } i \text{ of} \]
\begin{align*}
& I.\text{init} \Rightarrow \text{Init}'' | \\
& I.\text{loop} \Rightarrow \text{Loop}'' | \\
& I.\text{final} \Rightarrow \text{Final}''
\end{align*}

### 5.5 Data refinement of the transitions

**Theorem** \([\text{init1-a simp}]: \]
\[ \text{DataRefinement } (\{I.\text{init}'\} \circ Q1^{'-a}) R1^{'-a} R2^{'-a} Q1''^{'-a} \langle \text{proof} \rangle \]

**Theorem** \([\text{init2-a simp}]: \]
\[ \text{DataRefinement } (\{I.\text{init}'\} \circ Q2^{'-a}) R1^{'-a} R2^{'-a} Q2''^{'-a} \langle \text{proof} \rangle \]
\textbf{theorem} \textit{step1-a [simp]}: 
\begin{equation*}
\text{DataRefinement} \left( \{ \text{.Loop }. \} \circ \text{Q3}'-a \right) \text{R2}'-a \text{R2}'-a \text{Q3}''-a 
\end{equation*}
\hspace{1em} \langle \text{proof} \rangle

\textbf{lemma} \textit{neqif [simp]}: \( x \neq y \implies (\text{if } y = x \text{ then } a \text{ else } b) = b \) 
\hspace{1em} \langle \text{proof} \rangle

\textbf{theorem} \textit{step2-a [simp]}: 
\begin{equation*}
\text{DataRefinement} \left( \{ \text{.Loop }. \} \circ \text{Q4}'-a \right) \text{R2}'-a \text{R2}'-a \text{Q4}''-a 
\end{equation*}
\hspace{1em} \langle \text{proof} \rangle

\textbf{lemma} \textit{ssetsimp}: \( a = c \implies (x \in a) = (x \in c) \) 
\hspace{1em} \langle \text{proof} \rangle

\textbf{theorem} \textit{step3-a [simp]}: 
\begin{equation*}
\text{DataRefinement} \left( \{ \text{.Loop }. \} \circ \text{Q4}'-a \right) \text{R2}'-a \text{R2}'-a \text{Q5}''-a 
\end{equation*}
\hspace{1em} \langle \text{proof} \rangle

\textbf{theorem} \textit{final-a [simp]}: 
\begin{equation*}
\text{DataRefinement} \left( \{ \text{.Loop }. \} \circ \text{Q5}'-a \right) \text{R2}'-a \text{R1}'-a \text{Q6}''-a 
\end{equation*}
\hspace{1em} \langle \text{proof} \rangle

\subsection{5.6 Diagram data refinement}

\textbf{lemma} \textit{apply-fun-index [simp]}: \( (r .. P) \ i = (r \ i) \ (P \ i) \) 
\hspace{1em} \langle \text{proof} \rangle

\textbf{lemma [simp]}: \( \text{Disjunctive-fun} \ (r::('c \Rightarrow 'a::complete-lattice) \Rightarrow 'b::complete-lattice)) \) 
\hspace{1em} \( \implies \text{mono-fun} \ r \) 
\hspace{1em} \langle \text{proof} \rangle

\textbf{theorem} \textit{LinkMark-DataRefinement-a [simp]}: 
\begin{equation*}
\text{DgrDataRefinement2} \ ((R-a .. \text{SetMarkInv}) \text{StackMark-a} \text{R}'-a \text{LinkMark} 
\end{equation*}
\hspace{1em} \langle \text{proof} \rangle

\textbf{lemma [simp]}: \( \text{mono} \ Q1'-'a \) 
\hspace{1em} \langle \text{proof} \rangle

\textbf{lemma [simp]}: \( \text{mono} \ Q2'-'a \) 
\hspace{1em} \langle \text{proof} \rangle

\textbf{lemma [simp]}: \( \text{mono} \ Q3'-'a \) 
\hspace{1em} \langle \text{proof} \rangle

\textbf{lemma [simp]}: \( \text{mono} \ Q4'-'a \) 
\hspace{1em} \langle \text{proof} \rangle

\textbf{lemma [simp]}: \( \text{mono} \ Q5'-'a \) 
\hspace{1em} \langle \text{proof} \rangle

\textbf{lemma [simp]}: \( \text{dmono} \ \text{StackMark-a} \) 
\hspace{1em} \langle \text{proof} \rangle

\subsection{5.7 Diagram correctness}

\textbf{theorem} \textit{LinkMark-correct}:
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.

locale classical = node +
  fixes left0 :: 'node ⇒ 'node
  fixes right0 :: 'node ⇒ 'node
  fixes atom0 :: 'node ⇒ 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 ⊆ dsw: pointer nil root none::Index link0 label0
⟨proof⟩

context classical begin

lemma [simp]:
  (label0 = (\ x . if atom x then some else none)) = (atom0 = atom)
⟨proof⟩

definition gg mrk atom ptr x ≡ ptr x \neq nil \land ptr x \notin mrk \land \neg atom x
6.1 Transitions

definition
 QQ1-a ≡ [: p, t, left, right, atom, mrk ↼ p', t', left', right', atom', mrk'] .

root = nil ∧ p' = nil ∧ t' = nil ∧ mrk' = mrk ∧ left' = left
∧ right' = right ∧ atom' = atom ;]

definition
 QQ2-a ≡ [: p, t, left, right, atom, mrk ↼ p', t', left', right', atom', mrk'] .

root = nil ∧ p' = root ∧ t' = nil ∧ mrk' = mrk ∪ {root}
∧ left' = left ∧ right' = right ∧ atom' = atom ;]

definition
 QQ3-a ≡ [: p, t, left, right, atom, mrk ↼ p', t', left', right', atom', mrk'] .

p = nil ∧ gg mrk atom left p ∧
 p' = left p ∧ t' = p ∧ mrk' = mrk ∪ {left p} ∧
 left' = left(p := t) ∧ right' = right ∧ atom' = atom ;]

definition
 QQ4-a ≡ [: p, t, left, right, atom, mrk ↼ p', t', left', right', atom', mrk'] .

p = nil ∧ gg mrk atom right p ∧
 p' = right p ∧ t' = p ∧ mrk' = mrk ∪ {right p} ∧
 left' = left(p := t) ∧ right' = right ∧ atom' = atom ;]

definition
 QQ5-a ≡ [: p, t, left, right, atom, mrk ↼ p', t', left', right', atom', mrk'] .

p = nil ∧ (not needed in the proof *)
¬ gg mrk atom left p ∧ ¬ gg mrk atom right p ∧
 t = nil ∧ ¬ atom t ∧
 p' = t ∧ t' = left t ∧ mrk' = mrk ∧
 left' = left(t := p) ∧ right' = right ∧ atom' = atom ;]

definition
 QQ6-a ≡ [: p, t, left, right, atom, mrk ↼ p', t', left', right', atom', mrk'] .

p = nil ∧ (not needed in the proof *)
¬ gg mrk atom left p ∧ ¬ gg mrk atom right p ∧
 t = nil ∧ atom t ∧
 p' = t ∧ t' = right t ∧ mrk' = mrk ∧
 left' = left ∧ right' = right(t := p) ∧ atom' = atom(t := False) ;]

definition
 QQ7-a ≡ [: p, t, left, right, atom, mrk ↼ p', t', left', right', atom', mrk'] .

p = nil ∧
¬ gg mrk atom left p ∧ ¬ gg mrk atom right p ∧
 t = nil ∧
 p' = nil ∧ t' = t ∧ mrk' = mrk ∧
 left' = left ∧ right' = right ∧ atom' = atom ;]
7 Data refinement relation

definition
\[ \text{RR-a} \equiv \{ : \text{p, t, left, right, atom, mrk} \rightarrow \text{p', t', left', right', atom', mrk'} . \]
\[ p = \text{nil} \land p' = p \land t' = t \land \text{mrk'} = \text{mrk} \land \text{left'} = \text{left} \land \text{right'} = \text{right} \land \text{atom'} = \text{atom} : \]  

7.1 Data refinement of the transitions

definition
ClassicMark = (\( \lambda (i, j) . (\text{case} (i, j) \text{ of} \]
\( (\text{I.init, I.loop}) \Rightarrow \text{QQ1-a} \cap \text{QQ2-a} | \)
\( (\text{I.loop, I.loop}) \Rightarrow (\text{QQ3-a} \cap \text{QQ4-a}) \cap ((\text{QQ5-a} \cap \text{QQ6-a}) \cap \text{QQ7-a}) | \)
\( (\text{I.loop, I.final}) \Rightarrow \text{QQ8-a} | \)
\(- \Rightarrow \top) \})

theorem init1-a [simp]:
\[ \text{DataRefinement} (\{ \text{Init}'' \} \circ \text{Q1''-a}) \text{ RR-a RR-a QQ1-a} \]
\langle proof \rangle

theorem init2-a [simp]:
\[ \text{DataRefinement} (\{ \text{Init}'' \} \circ \text{Q2''-a}) \text{ RR-a RR-a QQ2-a} \]
\langle proof \rangle

lemma index-simp:
\[ (u = v) = (u \text{ none} = v \text{ none} \land u \text{ some} = v \text{ some}) \]
\langle proof \rangle

theorem step1-a [simp]:
\[ \text{DataRefinement} (\{ \text{Loop}'' \} \circ \text{Q3''-a}) \text{ RR-a RR-a QQ3-a} \]
\langle proof \rangle

theorem step2-a[simp]:
\[ \text{DataRefinement} (\{ \text{Loop}'' \} \circ \text{Q4''-a}) \text{ RR-a RR-a QQ4-a} \]
\langle proof \rangle

theorem step3-a [simp]:
\[ \text{DataRefinement} (\{ \text{Loop}'' \} \circ \text{Q5''-a}) \text{ RR-a RR-a QQ5-a} \]
\langle proof \rangle
lemma if-set-elim: \((x \in (\text{if } b \text{ then } A \text{ else } B)) = ((b \land x \in A) \lor (\neg b \land x \in B))\)

\langle proof \rangle

theorem step4-a [simp]:
DataRefinement (\{.Loop".\} \circ Q4"-a) RR-a RR-a QQ6-a
\langle proof \rangle

theorem step5-a [simp]:
DataRefinement (\{.Loop".\} \circ Q5"-a) RR-a RR-a QQ7-a
\langle proof \rangle

theorem final-step-a [simp]:
DataRefinement (\{.Loop".\} \circ Q6"-a) RR-a RR-a QQ8-a
\langle proof \rangle

7.2 Diagram data refinement

lemma [simp]: mono RR-a \langle proof \rangle
lemma [simp]: RR-a \in Apply.Disjunctive \langle proof \rangle
lemma [simp]: Disjunctive-fun R"-a \langle proof \rangle

lemma [simp]: mono-fun R"-a \langle proof \rangle

lemma [simp]: mono Q1"-a \langle proof \rangle
lemma [simp]: mono Q2"-a \langle proof \rangle
lemma [simp]: mono Q3"-a \langle proof \rangle
lemma [simp]: mono Q4"-a \langle proof \rangle
lemma [simp]: mono Q5"-a \langle proof \rangle
lemma [simp]: mono Q6"-a \langle proof \rangle

lemma [simp]: dmono LinkMark
\langle proof \rangle

theorem ClassicMark-DataRefinement-a [simp]:
DgrDataRefinement2 (R'-a .. (R-a .. SetMarkInv)) LinkMark R"-a ClassicMark
\langle proof \rangle

7.3 Diagram correctness

theorem ClassicMark-correct-a [simp]:
Hoare-dgr (R'-a .. (R-a .. (R-a .. SetMarkInv))) ClassicMark
((R"-a .. (R'-a ..(R-a .. SetMarkInv))) \cap (\neg grd (step ClassicMark)))
\langle proof \rangle

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 correctness of the
marking algorithm.
The predicate \textit{ClassicInit} which is true for the \textit{init} situation states that initially the variables \textit{left}, \textit{right}, and \textit{atom} are equal to their initial values and also that no node is marked.

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

\begin{verbatim}
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}
\end{verbatim}

\textbf{theorem [simp]}:
\begin{verbatim}
ClassicInit ⊆ (RR-a (R1′-a (R1-a (SetMarkInv init))))
\end{verbatim}
\textit{proof}

\textbf{theorem [simp]}:
\begin{verbatim}
(RR-a (R1′-a (R1-a (SetMarkInv final)))) ≤ ClassicFinal
\end{verbatim}
\textit{proof}

The indexed predicate \textit{ClassicPre} is the precondition of the diagram, and since we are only interested in starting the marking diagram in the \textit{init} situation we set \textit{ClassicPre loop} = \textit{ClassicPre final} = ∅.

\begin{verbatim}
definition [simp]:
 ClassicPre i = (case i of
  I.init ⇒ ClassicInit |
  I.loop ⇒ {} |
  I.final ⇒ {})
\end{verbatim}

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

\begin{verbatim}
definition [simp]:
 ClassicPost i = (case i of
  I.init ⇒ {} |
  I.loop ⇒ {} |
  I.final ⇒ ClassicFinal)
\end{verbatim}
The final theorem states the correctness of the marking diagram with respect to the precondition $\text{ClassicPre}$ and the postcondition $\text{ClassicPost}$, 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.

**References**


matisch Centrum, Amsterdam, 1980.


