Verification of the Deutsch-Schorr-Waite Marking Algorithm with Modal Logic

Yoshifumi Yuasa[1,4], Yoshinori Tanabe[2,4], Toshifusa Sekizawa[3,4], and Koichi Takahashi[4]

[1] Graduate School of Information Science and Engineering, Tokyo Institute of Technology
[2] Graduate School of Information Science and Technology, University of Tokyo
[3] Graduate School of Information Science and Technology, Osaka University
[4] Research Center for Verification and Semantics (CVS), National Institute of Advanced Industrial Science and Technology (AIST)

Abstract. We have proposed an abstraction technique that uses the formulas of variants of the modal μ-calculus as a method for analyzing pointer manipulating programs. In this paper, the method is applied to verify the correctness of the Deutsch-Schorr-Waite marking algorithm, which is regarded as a benchmark of such analysis. Both the partial correctness and the termination are discussed. For the former, we built a system on top of the proof assistant Agda, with which the user constructs Hoare-style proofs. The system is an optimum combination of automatic and interactive approaches. While a decision procedure for a variant of modal μ-calculus, which is available through the Agda plug-in interface, enables the user to construct concise proofs, the run time is much shorter than for automatic approaches.

1 Introduction

The Deutsch-Schorr-Waite (DSW) algorithm is a marking algorithm for garbage collection. An ordinary marking algorithm based on depth first search uses a stack to remember the node to return to after completing the marking from the current node. The DSW algorithm does not use a stack. Instead, it temporarily alters the pointers of the current node to remember the node to return to. When the marking from the current node is completed, it returns to the remembered node and restores the pointers.

The correctness of the DSW algorithm is not trivial, and it has been regarded as a benchmark of verification methods for pointer manipulating programs. In [9,10], we proposed a method to analyze pointer manipulating programs using modal logics. In this research, we apply the method to verify the correctness of the DSW algorithm.

In our approach, a heap is regarded as a variant of the Kripke structure, which we call a pointer structure. We regard each cell in the heap as a state of the Kripke structure, a "points-to" relation by a pointer-type field of the cell as a labeled transition relation, a boolean-type field of the cell as a predicate symbol (an ordinary atomic formula), and a pointer-type program variable as a nominal (an atomic formula that is satisfied by only one state). We define a variant of modal μ-calculus to describe heap statuses. Due to the fixed-point operators, we can express the various properties of a heap. Moreover, the language has two features that we utilize in verification. First, we can express the weakest preconditions and the strongest postconditions of a formula with respect to the pointer operations by a formula of the logic [10]. Second, the satisfiability problem of the logic is decidable. We have implemented an efficient procedure to solve the problem for the alternation-free fragments of the logic [11], which have a sufficient expressive power for our aim. Combining these two features, we can judge whether a Hoare triple "$A\,\{cs\}\,B$" holds or not, where A and B are formulas of the logic and cs is a basic block of a program.

We built a system to construct Hoare-style proofs on top of the proof assistant Agda [1]. A validity checker of the abovementioned Hoare triples is implemented as a command-line program that the user calls through the plug-in mechanism of Agda. Using the system, we verify the partial correctness of the DSW algorithm.

Our approach is in a sense a middle ground between the automatic and interactive methods. The user of the system manually constructs a proof in Hoare logic, with the help of the validity checker.

The burden on the user is lighter than writing an entire proof since some part of the proof is automatically carried out by the validity checker. Moreover, although invariants of while clauses need to be manually constructed, the system helps to find them: based on the intuition, the user can enumerate statuses of the heap when the while clause is executed and define the transitions between them, as we illustrate in Section 5.2. The correctness of the transitions is verified by the validity checker. Then, the disjunction of the formulas that define the statuses is a suitable invariant for the while clause. Note that even in the automated system, ingredients of the invariants need to be given to the system.

The running time of our system is much shorter than the automatic systems. This is because while the automatic systems need to construct (typically) big transition systems, we calculate only what the user explicitly requests.

As a result, the proof we construct for the partial correctness of the DSW algorithm consists of a few hundred lines in the Agda language and the total running time of the validity checker is less than three minutes.

For a termination proof of the DSW algorithm in the "heaps as Kripke structures" framework, we define three formulas that induce ranking functions. The correctness is reduced to the validity of some formulas constructed from them.

We briefly review related work. The DSW algorithm was presented in its original form in [8]. One of the earliest studies to give a formal proof that can be checked by computer is [2] by Bornat, in which the proof assistant Jape was used. Mehta and Nipkow used the higher-order system Isabelle/HOL in [6]. Yang wrote a formal proof in the logic called Bunched Implications [12], which becomes the basis of separation logic.

In the abovementioned studies, proofs were constructed manually and then checked by proof assistants on computer. Hubert and Marché [4] used a tool called CADUCEUS, which handles C source code directly, and invariants can be embedded into the code as comments in a special form. The tool verifies that they are really invariants.

Loginov *et al.* automated the verification [5]. They use the tool TVLA [7], which performs abstract interpretation of pointer manipulating programs. Since the default predicates in TVLA are insufficient to verify the properties of the algorithm, they added predicates for this verification. TVLA completes the verification in several hours, verifying both the partial correctness and termination; however, the input data is restricted to the shape of a tree.

The remainder of the paper is organized as follows: In Section 2, we define a tiny programming language, which we use throughout the paper, and then show how to describe the DSW marking algorithm in that language. The section includes an explanation of three important sub-processes push, swing and pop of the DSW algorithm. Section 3 introduces the concept of "heaps as Kripke structures," on which our verification framework is based. Two key theorems and a validity checker of Hoare triples based on them are described. In Section 4, we briefly explain the proof assistant Agda, the front-end of our system, and describe a library we provide on the top of Agda to help users reasoning in the Hoare logic. In Section 5, we show the correctness of the DSW algorithm. A proof of the partial correctness built with our system is described in detail, and termination is also discussed. Section 6 concludes the paper.

2 Deutsch-Schorr-Waite Marking Algorithm

2.1 Programming Language

Through this paper, we describe pointer algorithms in a tiny procedural programming language. We assume that cells in the heap we manipulate are of the same type, *i.e.*, they all have pointer-type fields indexed by F and bit-fields (boolean-type fields) indexed by B for fixed finite sets F and B of letters. We also fix another finite set V of program variables. All program variables are of pointer-type, *i.e.*, they either hold cells or have the value "null". We introduce a program constant "`Null`" holding the null, and denote $\mathsf{V} \cup \{\texttt{Null}\}$ by V^*.

Here, E_0 and C_0 are the sets of *atomic expressions* and *basic operations* on cells, as listed below.

$$\begin{aligned}\mathsf{E}_0 &::= \mathsf{V} \texttt{ == } \texttt{Null} \mid \mathsf{V} \texttt{ == } \mathsf{V} \mid \mathsf{V}\texttt{.}\mathsf{B} \texttt{ == } \texttt{True} \mid \mathsf{V}\texttt{.}\mathsf{B} \texttt{ == } \texttt{False}\\ \mathsf{C}_0 &::= \mathsf{V} \texttt{ := } \texttt{Null} \mid \mathsf{V} \texttt{ := } \mathsf{V} \mid \mathsf{V}\texttt{.}\mathsf{B} \texttt{ := } \texttt{True} \mid \mathsf{V}\texttt{.}\mathsf{B} \texttt{ := } \texttt{False} \mid\\ &\quad\ \mathsf{V} \texttt{ := } \mathsf{V}\texttt{.}\mathsf{F} \mid \mathsf{V}\texttt{.}\mathsf{F} \texttt{ := } \mathsf{V}\end{aligned}$$

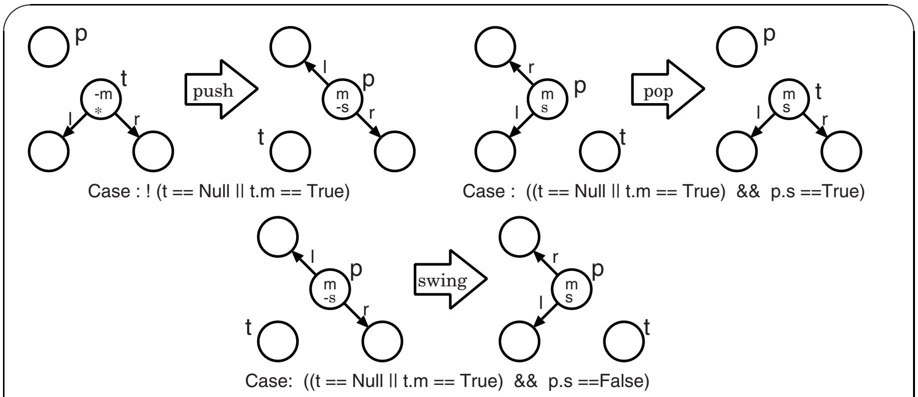
Then we define the *boolean expressions* E, *clauses* C, and *programs* P in the language, where the symbol ϵ represents an empty sequence.

$$\begin{aligned}\mathsf{E} &::= \mathsf{E}_0 \mid (\texttt{!}\mathsf{E}) \mid (\mathsf{E} \texttt{ \&\& } \mathsf{E}) \mid (\mathsf{E} \texttt{ || } \mathsf{E})\\ \mathsf{C} &::= \mathsf{C}_0 \mid \texttt{if } \mathsf{E} \texttt{ then } \{\mathsf{P}\} \texttt{ else } \{\mathsf{P}\} \mid \texttt{while } \mathsf{E} \texttt{ do } \{\mathsf{P}\}\\ \mathsf{P} &::= \epsilon \mid \mathsf{C}\texttt{;}\mathsf{P}\end{aligned}$$

In addition, any white space and comments bracketed by '`/*`' and '`*/`' can be inserted arbitrarily in the program in the usual manner.

```
    t := root; p := Null;                               /* init  */
    while (!(p == Null) || !(t == Null || t.m == True)) do {
      if (!(t == Null || t.m == True)) then {           /* push  */
         x := p; p := t  ; t := t.l; p.l := x;
         p.s := False; p.m := True;
       } else { if (p.s == False) then {                /* swing */
         x := t; t := p.r; y := p.l; p.r := y; p.l := x;
         p.s := True ;
       } else {                                         /* pop   */
         x := t; t := p  ; p := p.r; t.r := x;
       }; };
     };
```

Fig. 1. The DSW algorithm


p
t
-m
*
l
r
push
l
p
m
-s
r
t
Case : ! (t == Null || t.m == True)
r
p
m
s
l
t
pop
p
t
m
s
l
r
Case : ((t == Null || t.m == True) && p.s ==True)
l
p
m
-s
r
t
swing
r
p
m
s
l
t
Case: ((t == Null || t.m == True) && p.s ==False)

Letters m, s in cells represent that the bit-fields indexed by them hold true, $-m$, $-s$ represents they hold false, and "$*$" represents we don't concern the field.

Fig. 2. Operations with the three in-line procedures in the while loop

2.2 DSW Algorithm in the Tiny Language

Figure 1 presents the DSW algorithm described in our language. We will restrict ourselves to deal only with the case where a cell has two pointer-indices $l$eft and $r$ight, as in [4] and [5]. The bit-indices are $m$ark and $s$wung. The mark-fields of cells reachable from the root-cell are set to be true during the process and the swung-fields indicate whether the cell has been processed with the swing procedure described below.

At a point in execution time, we call the cell held by the variable p a *current cell*, and the cell held by t a *next cell*. We start with setting the current cell to null and the next cell as the root-cell in the heap, and then go into the while loop, in which one of the three in-line procedures is operated depending on the conditions of the current and next cells (see Fig.2). We exit the while loop if the current cell is null and the next cell is either null or marked.

In the following argument, the three in-line procedures in the while loop are termed *push*, *swing*, and *pop*, as in [4]. Push and pop correspond to the procedures used by typical depth-first marking algorithms with a stack. In the DSW algorithm, we do not use a stack to remember the cell to return to, but use the pointer a cell having been pointed by which we are marking from. The swing procedure exchanges the pointer from left to right, when we have completed marking from the left cell.

3 Heaps as Kripke Structures

3.1 Pointer Structures

We represent a heap status by a directed graph structure with additional information, regarding the cells as the nodes and the pointers as the edges. Formally, we consider a quintuple $(S, nil, \{p_f\}_{f \in \mathsf{F}}, \rho_0, \rho_1)$ consisting of:

– A finite set S of cells and $nil \in S$ for the destination of any null pointers,
– A set of points-to functions $p_f : S \to S$ indexed by F assuming each p_f maps nil to nil,
– A function $\rho_0 : \mathsf{B} \to \mathcal{P}S$ mapping a bit-index to the set of cells holding "true" in the bit-field, and
– A function $\rho_1 : \mathsf{V}^* \to S$ mapping a variable to the cell held by it (or to nil) and the constant `Null` to nil.

We call such a quintuple a *pointer structure*. A pointer structure P induces a Kripke structure $K(P) = (S, \{R_f\}_{f \in \mathsf{F}}, \rho)$ on the propositional variables $\mathsf{B} \cup \mathsf{V}^*$ with R_f's and ρ defined as:

$$s \, R_f \, t \Leftrightarrow p_f \, s = t, \qquad \rho(x) = \begin{cases} \rho_0(x) & (\text{if } x \in \mathsf{B}\ ) \\ \{\rho_1(x)\} & (\text{if } x \in \mathsf{V}^*). \end{cases}$$

We now introduce a formal language to describe heap statuses. This paper defines the language as a variant of *modal μ-calculus*, although the definition below works for various modal languages. The reader is assumed aware of the language and the semantics of modal μ-calculus. We use the basic notions and terms about it following [13]. In the following argument, we call a formula in the modal μ-calculus a *modal formula*.

We call a formula in the form "$@v\,\varphi$" a *basic p-formula*, for a program variable v and a modal formula φ with modalities from F and atomic propositions from $\mathsf{B} \cup \mathsf{V}^*$. Its satisfaction by a pointer structure P is defined as:

$$P \models @v\,\varphi \Leftrightarrow K(P), \rho_1(v) \models \varphi,$$

where the '$\models$' on the right-hand side represents the satisfaction for the modal μ-calculus. In other words, a pointer structure satisfies $@v\,\varphi$, if the induced Kripke structure satisfies φ at $\rho_1(v)$. We call boolean combinations of basic p-formulas *p-formulas*; satisfaction is extended in the usual way.

A basic p-formula $@v\,\varphi$ is called to be *alternation-free* if φ is alternation-free, *i.e.*, of alternation depth less than 2. Note that any CTL formula is equivalent to an alternation-free modal formula. A p-formula is called to be alternation-free if its components are all alternation-free.

3.2 Examples of P-Formulas

Figure 3 shows some examples of p-formulas; the pointer structure satisfies the three p-formulas. There are four schemata defining modal formulas $\boldsymbol{\alpha}\,\varphi$, $\boldsymbol{\beta}\,\varphi$, $\boldsymbol{\gamma}\,\varphi$, and $\boldsymbol{\delta}\,\varphi$ for a modal formula φ, which are taken from the argument to prove the partial correctness of the DSW algorithm in Section 5. Note that they are alternation-free, if φ is alternation-free.

The pointer structure on the right satisfies the following p-formulas:

$$@x\,(\boldsymbol{\beta}\,w),\quad @y\,(\boldsymbol{\gamma}\,z),\quad @x\,(\boldsymbol{\delta}\,z),$$

where :

$$\begin{aligned}
\boldsymbol{\alpha}\,\varphi &= (s \wedge \Diamond_r \varphi) \vee (\neg s \wedge \Diamond_l \varphi)\\
\boldsymbol{\beta}\,\varphi &= \mu X.(\varphi \vee (m \wedge \boldsymbol{\alpha}\,X))\\
\boldsymbol{\gamma}\,\varphi &= \mu X.(\neg m \wedge (\varphi \vee \Diamond_l X \vee \Diamond_r X))\\
\boldsymbol{\delta}\,\varphi &= \boldsymbol{\beta}(\Diamond_r(\boldsymbol{\gamma}\,\varphi))
\end{aligned}$$

. . . .

$\mathsf{F} = \{l\text{eft}, r\text{ight}\}$, $\mathsf{B} = \{m\text{ark}, s\text{wung}\}$,
$\mathsf{V} = \{x, y, z, w\}$

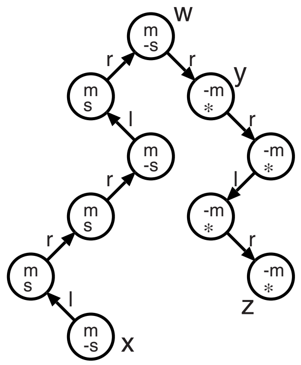


Fig. 3. Examples of p-formulas

The formula $\boldsymbol{\alpha}\,\varphi$ says that the formula φ holds at either the left successor or the right successor cell depending on the swung-field. Then, the formula $\boldsymbol{\beta}\varphi$ represents the reachability to a cell with the property φ, through a path on which all cells have the value "true" in their mark-fields and all pointers to succeeding cells are selected in the same way as in $\boldsymbol{\alpha}$. The formula $\boldsymbol{\gamma}\,\varphi$ represents a more liberal reachability to a cell with φ. One can go through a path selecting any desired left successors and right successors, as long as they have the value "false" in their mark-fields. The schema $\boldsymbol{\delta}$ is a combination of $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$.

3.3 Strongest Post-conditions and Satisfiability Problem

The reason why we choose the language in this paper is that it has two desired properties, which are described in the following key theorems of our framework, where we call a program a *basic block* if it consists of basic operations, and denote by "$cs\,P$" the pointer structure obtained from a pointer structure P by operating a program cs.

Theorem 1. *Let cs be any basic block. There exists a procedure spc which calculates the strongest post condition of a p-formula with respect to cs, i.e.,* $cs\,P \models spc(A, cs)$ *if and only if* $P \models A$ *for any pointer structure P and any p-formula A. Particularly,* $spc(A, cs)$ *is an alternation-free formulas, if A is alternation-free.*

Theorem 2. *There exists a procedure "sat" defined on the alternation-free p-formulas, which determines whether a given p-formula can be satisfied, i.e., sat A is true if and only if there is a pointer structure that satisfies A.*

See [10] and [11] for the proofs of these theorem. The last part of Theorem 1 is not mentioned in [10], but it is clear from the proof shown there.

As suggested by Theorem 2, the expressive power of p-formulas is weaker than languages used in frameworks such as [5] and [12]. Still, all the necessary predicates to prove the correctness of the DSW algorithm can be expressed as we will see in Section 5. Also, note that the procedure mentioned in the theorem judges whether a pointer structure, instead of Kripke structure, exists or not. Although the satisfaction relation of a pointer structure is defined by regarding it as a Kripke structure, there are Kripke structures that cannot be seen as a pointer structure. The procedure is aware of the difference for precise analysis.

Then, we define:

$$\begin{aligned} \mathit{pftrans}(A, cs, B) &= \mathit{sat}\,(\mathit{spc}(A, cs) \wedge B), \\ \mathit{pfvalid}(A, cs, B) &= \neg\mathit{sat}\,(\mathit{spc}(A, cs) \wedge \neg B). \end{aligned}$$

The former determines if the operation *cs* changes *some* pointer structure with property A to a structure with property B, on the other hand, the latter determines if the operation *cs* changes *all* pointer structures with A to structures with B. A Hoare-triple "$A\,\{cs\}\,B$" is clearly valid if "$\mathit{pfvalid}(A, cs, B)$" is true.

The system presented in the paper implements *pfvalid* in Java with BDD library, and uses it to help user's Hoare-style reasoning with a proof assistant.

Based on the concept in this section (but using 2-way CTL instead of the μ-calculus), we developed another system called Modal Logic Abstraction Tool (MLAT), which supports automatic reasoning on pointer algorithms. It implements *pftrans* to create a model by predicate abstraction. See Sekizawa *et al.* [9] for more information about MLAT.

4 Hoare Logic on the Proof Assistant

4.1 Agda, a Proof Assistant

We use an interactive proof assistant *Agda* for the front-end of our tool. We briefly describe the system here. See [1] for further information.

Agda is developed in Chalmers University of Technology. Its input language, called the *Agda language*, is based on Martin-Löf's constructive type theory. The concrete syntax is similar to those of functional programming languages, especially of Haskell. A logic is encoded into Agda language by Curry-Howard isomorphism. Formulas are encoded as types. For example, a conjunction of formulas is encoded as a Cartesian product of types. Users prove theorems by constructing terms, called *proof terms*, inhabiting the theorems encoded as types. For example, a proof term of the distributive law of propositional logic is defined as follows, provided `cnjIntr`, `cnjElimL`, ... are encoded inference rules of the natural deduction system. (`cnjIntr` is a pairing function as which we encode the introduction rule for conjunction, ...*etc.*)

```
dstr (A,B,C :: Prop) :: A && (B || C) -> (A && B) || (A && C)
     = \(p :: A && (B || C)) ->
       let q ::  A       = cnjElimL p ;
           r :: (B || C) = cnjElimR p ;
        in dsjElim r (\(x :: B) -> dsjIntrL (cnjIntr q x))
                     (\(y :: C) -> dsjIntrR (cnjIntr q y))
```

Agda checks types of user's inputs as above. A proof is correct, if it is well-typed.

Users can omit some arguments of a function, called *hidden arguments*, if they are inferred from the rest of arguments. For example, the first two arguments `A` and `(B || C)` of cnjElimL in the proof above are omitted. In fact, arguments of a function are hidden by default. Users declare arguments not hidden by marking them with "`!`" when they define the function. (See Fig.7 for example.)

Agda is equipped with a general plug-in mechanism, through which we execute any external prover that has a command line interface. To invoke an external prover, we put a keyword "external" followed by its plug-in name, on the right-hand side of a defining equation of a proof term. For the example above, we prove the law as:

```
dstr (A,B,C :: Prop) :: A && (B || C) -> (A && B) || (A && C)
     = external "fol"
```

where "fol" is a plug-in name of a prover for first-order logic on the command line. Agda regards the definition to be type correct, if the prover answers positively.

4.2 Hoare Logic Library

We provide a library, consisting of about 5000 lines, for Hoare-style reasoning on our programming language. The following items are encoded into Agda language:

- The programming language (boolean expressions, clauses, and programs),
- Modal formulas, p-formulas, and their sequent-style inference rules,
- Hoare triples and their inference rules.

The encodings are done in the manner of *deep-embedding*; for example, the programming language and the p-formulas are encoded as data types, then, the Hoare triples are encoded as an inductive type-family on these data types. A type constructor HTc makes the Hoare triple of a clause and two p-formulas (pre- and post-conditions). Another type constructor HTp is similar, but its first argument is a program. The inference rules of Hoare logic are shown in

Rules in the Hoare logic:

$$\frac{spc(A,c;)\vdash B}{A\,\{c\}\,B}\text{atmHc}\qquad \frac{A\wedge\bar{s}\,\{cs_0\}\,B\quad A\wedge\neg\bar{s}\,\{cs_1\}\,B}{A\,\{\texttt{if}\,s\,\texttt{then}\,\{cs_0\}\,\texttt{else}\,\{cs_1\}\}\,B}\text{iteHc}\qquad \frac{A\vdash B}{A\,\{\epsilon\}\,B}\text{nilHp}$$

$$\frac{A\vdash C\quad C\wedge\bar{s}\,\{cs\}\,C\quad C\wedge\neg\bar{s}\vdash B}{A\,\{\texttt{while}\,s\,\texttt{do}\,\{cs\}\}\,B}\text{whlHc}\qquad \frac{A\,\{c\}\,C\quad C\,\{cs\}\,B}{A\,\{c;cs\}\,B}\text{cnsHp}$$

Partially automated rules (bold face letters represent finite sets of formulas) :

$$\frac{\bigwedge\boldsymbol{A}_0\vdash B}{\bigwedge\boldsymbol{A}_1\vdash B}\text{cnjSup}(\boldsymbol{A}_0\subseteq\boldsymbol{A}_1)\qquad \frac{A\vdash\bigwedge\boldsymbol{B}_0\quad A\vdash\bigwedge\boldsymbol{B}_1}{A\vdash\bigwedge\boldsymbol{B}_2}\text{cnjSub}(\boldsymbol{B}_2\subseteq\boldsymbol{B}_0\cup\boldsymbol{B}_1)$$

$$\frac{\bigvee\boldsymbol{A}_0\vdash B\quad\bigvee\boldsymbol{A}_1\vdash B}{\bigvee\boldsymbol{A}_2\vdash B}\text{dsjSub}(\boldsymbol{A}_2\subseteq\boldsymbol{A}_0\cup\boldsymbol{A}_1)\qquad \frac{A\vdash\bigvee\boldsymbol{B}_0}{A\vdash\bigvee\boldsymbol{B}_1}\text{dsjSup}(\boldsymbol{B}_0\subseteq\boldsymbol{B}_1)$$

Fig. 4. Some inference rules in the library

Fig.4, where $\bar{s}$ denotes a p-formula with the same meaning as an expression s; for example, by considering the semantics of basic p-formulas, we have :

$$\overline{(x == y)} = @x\,y \quad\text{and}\quad \overline{(x.b == false)} = @x\,\neg b.$$

These rules are encoded as constructors of encoded Hoare triples. For example, the encoded rule for the if-then-else clause is typed as:

```
iteHc (s :: Expr) (cs, ds :: Prog) (!A, B :: PForm)
  :: HTp cs (A && s2f s) B -> HTp cs (A && not (s2f s)) B ->
     HTc (if_then_else s cs ds) A B
```

A user may prove a Hoare triple "$A\,\{cs\}\,B$" by invoking the external prover *pfvalid* through the plug-in mechanism, instead of writing down a proof term, if *cs* is a basic block and A, B are alternation-free. In particular, by taking the empty sequence for *cs*, we prove a logical sequent "$A \vdash B$" by the prover.

We also implement, by the technique called *reflection* [3], a simple "internal" prover on finite sets, depending not on any external prover, but on the type checking mechanism of Agda itself. Although internal provers are not so powerful as external provers, there is an advantage of using them: One can define a function with a type involving calls to internal provers, and then execute the provers every time the function is used, which is not possible with external provers. This way, we partially automate some derived rules in the library. See the lower part of Fig.4, where the conditions in parentheses are tested automatically before the rules are applied. These rules help a user out of many boring applications of associative and commutative rules of the boolean operators. A user can also define new automated rules on the top of the library for one's own sake. In this research, we define the following rule used in Sec. 5.2:

$$\frac{\{\ C\wedge A\wedge\bigvee\boldsymbol{B}_i\ \{cs\}\ A\wedge\bigvee\boldsymbol{D}_i\ \}_{i\in I}}{C\wedge A\wedge\bigvee\boldsymbol{B}\ \ \{cs\}\ A\wedge\bigvee\boldsymbol{D}}\text{lem3}(\boldsymbol{B}\subseteq\textstyle\bigcup_i\boldsymbol{B},\bigcup_i\boldsymbol{D}\subseteq\boldsymbol{D})$$

which combines proofs of the relationships between states in an abstract transition model to prove correctness of the loop invariant obtained from the model.

5 Verification of the DSW Algorithm

5.1 Outline of the Verification

We now verify the partial correctness of the DSW marking algorithm, which consists of the following:

1. The points-to relationships between cells are preserved.
2. A cell reachable from the root-cell is marked.
3. A cell not reachable from the root-cell is not marked.

We first prove property 3, the easiest among them, to illustrate our method. Properties 1 and 2 are shown in a similar manner. However, the loop invariant needed to prove them is complicated, while the invariant for property 3 is simple enough for our external prover to check directly. In section 5.2, we present an idea for managing such a situation.

To show property 3, we construct a proof term of "*Pre* {*dsw*} *Pos*" encoded to the Agda language, letting:

$$\begin{aligned} \mathit{Pre} &= @a\,(\neg m \wedge \neg\texttt{Null}) \wedge @\mathit{root}\,\boldsymbol{\eta}\,a \\ &\qquad\qquad \text{where } \boldsymbol{\eta}\,\varphi = \neg\mu X.(\varphi \vee \Diamond_l X \vee \Diamond_r X), \\ \mathit{Pos} &= @a\,(\neg m). \end{aligned}$$

The program variable a should be fresh, so as to show property 3 for any cell in the heap. Figure 5 presents the proof, in which we proceed as follows:

i) Show the Hoare triple "*Pre* {*init*} *Inv*", which ensures we enter the while loop with the status *Inv*, a loop invariant,
ii) Check the loop invariant, first for the procedures push, swing, and pop, and then for the loop body as a result,
iii) Show "$\neg C \wedge \mathit{Inv} \vdash \mathit{Pos}$" for the loop condition C, which ensures we exit the loop with the status *Pos*, and then
iv) Combine the results above to construct the proof.

The external prover *pfvalid* helps us proving in steps i)-iii).

In Fig. 5, the function subHp converts a Hoare triple with the empty program to the corresponding logical sequent, and seqHp is the Hoare rule for appending programs which is similar to cnsHp. Both are provided in the library.

5.2 Loop Invariant and State Transition Model

We show properties 1 and 2 simultaneously, *i.e.*, make a proof term of Hoare triple "*Pre* {*dsw*} *Pos*", letting:

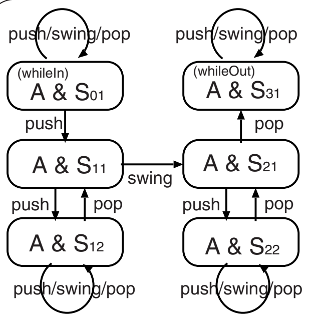
$$\begin{aligned} \mathit{Pre} &= @a\,(\neg m \wedge \Diamond_l b \wedge \Diamond_r c) \wedge @\mathit{root}\,\boldsymbol{\gamma}\,a \\ \mathit{Pos} &= @a\,(\;\; m \wedge \Diamond_l b \wedge \Diamond_r c), \end{aligned}$$

where the program variables a, b, and c are fresh and the modal formula $\gamma\, a$ is as in Fig. 3. Unlike property 3, finding a suitable loop invariant for proving the

```
Pre :: Pform
  = At_ a (not marked_ &&  not NULL_) && At_ root (eta a)
Pos :: Pform
  = At_ a (not marked_)

--   i) Show "Pre {init} Inv"
Inv :: Pform
  = At_ a (not marked_) && At_ p (not (eta a)) && At_ t (not (eta a))
initialize :: HTp init Pre Inv = external "pfvalid"

-- Cwhl is "!(p == Null) || !(t == Null || t.m == True)",
-- Cpsh is "!(t == Null || t.m == True)", and
-- Cswg is "p.s == False".
Gwhl :: Pform =                                    s2f Cwhl
Gpsh :: Pform =                        s2f Cpsh  && s2f Cwhl
Gswg :: Pform =      s2f Cswg  && not (s2f Cpsh) && s2f Cwhl
Gpop :: Pform = not (s2f Cswg) && not (s2f Cpsh) && s2f Cwhl

--  ii) Check the loop invariant
loopInv :: HTp loopBody (Gwhl && Inv) Inv
  = let invPush  :: HTp push  (Gpsh && Inv) Inv = external "pfvalid"
        invSwing :: HTp swing (Gswg && Inv) Inv = external "pfvalid"
        invPop   :: HTp pop   (Gpop && Inv) Inv = external "pfvalid"
    in cnsHp (iteHc (Gwhl && Inv)
                    invPush
                    (cnsHp (iteHc (not (s2f Cpsh) && Gwhl && Inv)
                                  invSwing
                                  invPop)
                           (nilHp idnt)))
             (nilHp idnt)

-- iii) Show "(not Cwhl && Inv) |- Pos"
exitwhl :: (not Gwhl && Inv) |- Pos
  = let exitwhl_ :: HTp [] (not Gwhl && Inv) Pos = external "pfvalid"
    in subHTp exitwhl_

--  iv) Main theorem
dswProp3 :: HTp dsw Pre Pos
  = seqHp initialize
          (cnsHp (whlHc Inv idnt
                            loopInv
                            exitwhl)
                 (nilHp idnt))
```

Lines beginning with “--” are comments.

Fig. 5. Verification of property 3 by Agda using *pfvalid*



$$
\begin{aligned}
A &= @a\,(\neg\mathtt{Null}) \wedge @p\,(\boldsymbol{\beta}\,\mathtt{Null})\\
S_{01} &= @a\,(\neg m \wedge \Diamond_l b \wedge \Diamond_r c) \wedge (@p\,(\boldsymbol{\delta}\,a) \vee @t\,(\boldsymbol{\gamma}\,a))\\
S_{11} &= @a\,(\;\; m \wedge \;\neg s \wedge \Diamond_r c) \wedge @p\,a \wedge @t\,b\\
S_{12} &= @a\,(\;\; m \wedge \;\neg s \wedge \Diamond_r c) \wedge @p\,(\boldsymbol{\beta}\,b) \wedge @b\,(\boldsymbol{\alpha}\,a)\\
S_{21} &= @a\,(\;\; m \wedge \Diamond_l b \wedge \;\;\; s) \wedge @p\,a \wedge @t\,c\\
S_{22} &= @a\,(\;\; m \wedge \Diamond_l b \wedge \;\;\; s) \wedge @p\,(\boldsymbol{\beta}\,c) \wedge @c\,(\boldsymbol{\alpha}\,a)\\
S_{31} &= @a\,(\;\; m \wedge \Diamond_l b \wedge \Diamond_r c) \wedge @p\,(\neg(\boldsymbol{\beta}\,a))\\
\mathit{Inv} &= A \wedge (S_{01} \vee S_{11} \vee S_{12} \vee S_{21} \vee S_{22} \vee S_{31})
\end{aligned}
$$

Fig. 6. The state transition model

```
invPush :: HTp push (Gpsh && Inv) Inv
 = let (==>) (!pre, !pos :: List Pform) :: Set
           =  HTp push (Gpsh && A & dsj pre) (A & dsj pos)
       p01 :: [S01] ==> [S01, S11]  = external "pfvalid"
       p11 :: [S11] ==> [S12]       = external "pfvalid"
       p12 :: [S12] ==> [S12]       = external "pfvalid"
       p21 :: [S21] ==> [S22]       = external "pfvalid"
       p22 :: [S22] ==> [S22]       = external "pfvalid"
       p31 :: [S31] ==> [S31]       = external "pfvalid"
   in lem3 push Gpsh A unit unit
                    (p01 $ p11 $ p12 $ p21 $ p22 $ p31 $ nilz)
```

Fig. 7. Checking the transitions with "push"

Hoare triple is not a trivial task. To overcome the difficulty, we consider a state transition model as shown in Fig. 6, and define a p-formula *Inv* as the disjunction of the formulas which represent states in the model, where the schemata $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ and $\boldsymbol{\delta}$ are as in Fig. 3.

The p-formula *Inv* is clearly a loop invariant, if transitions in the model are all correct. Unlike the proof of property 3, we show the correctness of the state transition model, because the invariant formula is too large for our external prover *pfvalid* to check at a time. The proof in Fig. 7 witnesses the invariant with respect to the procedure push. It first proves, using *pfvalid*, the operation push changes each state in the model to some state in the model as shown in Fig. 6, and then concludes by combining these proofs using lem3 mentioned in Section 4.2. The "unit"s applied to lem3 are objects calling the internal provers to check the conditions, and the binary operator "$" constructs a sequence of Hoare proofs. Proofs for swing and pop are constructed similarly.

The construction of the state transition model is based on the following observation. We fix an arbitrary cell "a" reachable from the root-cell. Recall that the current cell is the cell that is held by variable p. At the beginning of the marking algorithm, a has not yet become the current cell. The state $A\&S_{01}$

Table 1. Running times of the external prover

	Properties 1,2						Property 3
initialize	10.55						0.50
	$A\&S_{01}$	$A\&S_{11}$	$A\&S_{12}$	$A\&S_{21}$	$A\&S_{22}$	$A\&S_{31}$	
invPush	28.19	0.88	5.36	1.00	6.97	5.02	0.84
invSwing	25.42	1.17	12.95	0.55	11.84	6.70	1.64
invPop	12.67	0.69	6.94	0.95	8.63	2.75	0.88
exits loop	2.98						0.63

represents the heap status at this time. At some time point, a becomes the current cell for the first time and a is marked. The state $A\&S_{11}$ represents the heap status at this time. Then, typically, p leaves a for a while. The heap status is represented by $A\&S_{12}$. The cell a becomes the current cell for the second time. The heap status is represented by $A\&S_{11}$ again. At this time, swing is operated and the swung field of a becomes true. The state $A\&S_{21}$ represents the heap status at this time. Again, typically, p leaves a for a while, with the heap status represented by $A\&S_{22}$, and goes back to a for the third and last time, with the heap status represented by $A\&S_{21}$. Then, pop is operated and the state $A\&S_{31}$ represents the heap status.

Although it is not straightforward to find exact formulas that are suitable for each state, we can use a trial and error approach. We select formulas that may represent the heap status based on the intuition and check whether they make the transitions correct. Due to the prover *pfvalid*, we can perform this procedure effectively.

The entire proof for the partial correctness of the DSW marking algorithm, which we have described, consists of 320 lines in the Agda language, including the encoded DSW algorithm, states in the model, and lemmas and their proofs. There are 25 calls to the external prover; Table 1 summarizes their running times measured in seconds. The tests were performed on a 1.33 GHz Core 2 Duo system running Windows XP.

5.3 Termination

We can apply the same approach as above to show that the DSW algorithm terminates, although we have not yet implemented a system to construct such proofs. We give a sketch of the proof. In this section, we describe heap statuses by modal formulas, not by p-formulas. We say that a pointer structure P satisfies a modal formula φ, if $K(P), s \models \varphi$ for any cell s in P. The notion of the strongest post-condition *spc* is defined based on this satisfaction relation.

We assume a special modality "o" in our modal language, called the *global modality*, which is always interpreted by $S \times S$ in $K(P)$. Furthermore, we have the *inverse modality* $\bar{f}$ for each modality $f \in \mathsf{F}$, that is interpreted by R_f^{-1}. $\mathrm{Sat}_\varphi\, P$ denotes the set $\{s \in S; K(P), s \models \varphi\}$ and we define the modal formulas $\mathrm{NI}_{cs}\, \varphi$ and $\mathrm{DE}_{cs}\, \varphi$ as:

$$\begin{aligned} \mathrm{NI}_{cs}\,\varphi &= \Box_o(\varphi \Rightarrow \mathit{spc}(\varphi, cs)) \\ \mathrm{DE}_{cs}\,\varphi &= \Box_o(\varphi \Rightarrow \mathit{spc}(\varphi, cs)) \wedge \Diamond_o(\neg\varphi \wedge \mathit{spc}(\varphi, cs)), \end{aligned}$$

where P is a pointer structure on the cells S, φ a modal formula, and cs a basic block of a program. It is easy to prove the following lemma:

Lemma 1. *If formula* $\mathrm{NI}_{cs}\,\varphi$ *is valid,* $\mathrm{Sat}_\varphi\, P \supseteq \mathrm{Sat}_\varphi\,(cs\,P)$*. If formula* $\mathrm{DE}_{cs}\,\varphi$ *is valid,* $\mathrm{Sat}_\varphi\, P \supsetneq \mathrm{Sat}_\varphi\,(cs\,P)$*.*

Let $\varphi_1 = \neg m$, $\varphi_2 = \neg s$, and $\varphi_3 = \mu X.(p \vee \Diamond_{\bar{l}}(\neg s \wedge X) \vee \Diamond_{\bar{r}}(s \wedge X))$. The following formulas are valid:

$$\begin{array}{lll} \mathrm{DE}_{\mathrm{push}}\,\varphi_1, & \mathrm{NI}_{\mathrm{push}}\,\varphi_2, & \mathrm{NI}_{\mathrm{push}}\,\varphi_3, \\ & \mathrm{DE}_{\mathrm{swing}}\,\varphi_2, & \mathrm{NI}_{\mathrm{swing}}\,\varphi_3, \\ & & \mathrm{DE}_{\mathrm{pop}}\,\varphi_3. \end{array}$$

If the algorithm does not terminate, at least one of push, swing or pop is operated infinitely. Assume, for example, that swing occurs infinitely but pop is operated for a finite number of times. If we denote the i-th pointer structure at the top of the while loop by P_i, as per Lemma 1, $\mathrm{Sat}_{\varphi_2}\, P_i \supseteq \mathrm{Sat}_{\varphi_2}\, P_{i+1}$ for all $i \geq I$ for some natural number I and $\mathrm{Sat}_{\varphi_2}\, P_i \supsetneq \mathrm{Sat}_{\varphi_2}\, P_{i+1}$ for infinitely many i's, which is impossible. Similar arguments can be applied for other cases.

6 Conclusion and Future Work

In this paper, we showed that the correctness of the DSW marking algorithm can be proved using a framework based on a variant of the modal μ-calculus. For partial correctness, in particular, we built a system for constructing such proofs on top of the proof assistant Agda. Due to the external prover, we made the proofs very concise. We also showed that the termination of the DSW algorithm can be shown in our framework.

As future work, we plan to extend our system for termination proofs. First, the library for Agda should be extended to construct proofs, as in Section 5.3. Second, the command *pfvalid* needs to be extended to handle formulas with backward modalities and global modality. Backward modalities can be processed as shown in [11], and global modality can be handled in a straightforward manner. Although the procedure "*sat*" becomes incomplete in general due to the existence of backward modalities, it should be able to judge the validity of the six formulas in Section 5.3.

Although we have successfully verified the DSW algorithm, our framework is far from complete. The programming language lacks some important features such as function calls, especially recursive calls. The expressive power of the logic might be insufficient – the lack of existential quantifier is one of the problems. We are tackling these issues by trying to extend our framework.

Acknowledgments. This research was supported by the research project "Solving the description explosion problem in verification by means of structure transformation" in the CREST (Core Research for Evolution Science and Technology) program of the Japan Science and Technology Agency.

References

1. Agda Official Home Page, `http://unit.aist.go.jp/cvs/Agda/`
2. Bornat, R.: Proving pointer programs in Hoare logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000)
3. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Center (1995)
4. Hubert, T., Marché, C.: A case study of C source code verification: the Schorr-Waite algorithm. In: Proc. Software Engineering and Formal Methods (SEFM 2005), pp. 190–199. IEEE Computer Society, Los Alamitos (2005)
5. Loginov, A., Reps, T.W., Sagiv, M.: Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 261–279. Springer, Heidelberg (2006)
6. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 121–135. Springer, Heidelberg (2003)
7. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24(3), 217–298 (2002)
8. Schorr, H., Waite, W.M.: An efficient machine-independent procedure for garbage collection in various list structures. Commun. ACM 10(8), 501–506 (1967)
9. Sekizawa, T., Tanabe, Y., Yuasa, Y., Takahashi, K.: MLAT: A tool for heap analysis based on predicate abstraction by modal logic. In: The IASTED International Conference on Software Engineering (SE 2008), pp. 310–317 (2008)
10. Tanabe, Y., Sekizawa, T., Yuasa, Y., Takahashi, K.: Pre- and post-conditions expressed in variants of the modal μ-calculus. CVS/AIST Research Report AIST-PS-2008-009, CVS/AIST (2008)
11. Tanabe, Y., Takahashi, K., Hagiya, M.: A decision procedure for alternation-free modal μ-calculi. In: Advances in Modal Logic (to appear, 2008)
12. Yang, H.: An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In: Proceedings of the 1st Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (2001)
13. Zappe, J.: Modal μ-calculus and alternating tree automata. In: Grädel, E., Thomas, W., Wilke, T. (eds.) Automata, Logics, and Infinite Games. LNCS, vol. 2500, pp. 171–184. Springer, Heidelberg (2001)