Formal Proof of Abstract Model Checking of Concurrent Garbage Collection

Koichi Takahashi* Masami Hagiya†

March 15, 2002

Abstract

Abstract model checking is a method to verify properties of a state transition system by first making abstraction of the system and then applying model checking on the abstract system. If the employed abstraction relation satisfies certain conditions that guarantee the correctness of the abstraction, verification results of the abstract system can be shifted to the original one. However, the correctness of abstraction is not always easy to verify. In our previous work, we applied abstract model checking on concurrent garbage collection algorithms, but the correctness of the abstraction was not formally proven. In this paper, we formalize the abstraction and the underlying model for concurrent garbage collection algorithms, and formally prove the correctness of the abstraction using the proof assistant, HOL.

1 Introduction

Although model checking is a useful automatic verification method, the state explosion problem is a major drawback of the method. Abstract model checking is a solution to this problem. In abstract model checking, an abstract system is constructed from the original one, often called a concrete system, and the correctness of the abstract one is verified by ordinary exhaustive search [2, 1]. The abstraction relation defines how a state in the concrete system is abstracted to a state in the abstract one.

If the abstraction relation satisfies some conditions, then the safety of the abstract system implies that of the concrete one [1]. We also gave such conditions in our own framework of abstract model checking for safety of programs [9]. Therefore, the entire process of abstract model checking is completed by proving that the abstraction satisfies these conditions.

If an abstract system is constructed in some mechanical way, which guarantees the correctness of the abstract system, then the required conditions are automatically satisfied [1]. However, for practical problems, an abstract system is often manually constructed so that it is coarse enough in order for exhaustive search to be applicable and fine enough in order for verification to succeed. If an abstract system is manually constructed, it is not clear whether the abstraction relation satisfies the required conditions. Using a theorem prover to formally prove the conditions is considered an appropriate approach because of the following reasons:

- Manually constructed abstraction relations are often ad-hoc and error-prone. Formal proofs guarantee their correctness with complete rigor.
- It is usually the case that more than one abstract relation are tried until abstract model checking succeeds.

*National Institute of Advanced Institute for Science and Technology. k.takahashi@aist.go.jp
†Graduate School of Information Science and Technology, University of Tokyo.

- In addition, different abstract relations are defined for verifying different properties. A formal proof for one relation can be modified to a proof for other relations.
- Formally proving the abstraction relation is a light task compared with that of proving the concrete system from scratch (not by abstract model checking).

In our previous work [6], which is also included in the first author's Ph.D. thesis [11], we verified the safety of concurrent garbage collection algorithms by abstract model checking. But we did not formally prove that the abstract relation satisfies the conditions required to guarantee the safety of the concrete system from that of the abstract one. In this paper, we formalize and prove these conditions using the proof assistant, HOL [5], to complete verification.

This work is therefore a case study in which the correctness of a concurrent garbage collection algorithm is completely formally proven by abstract model checking. Since the general framework of abstract model checking has already been established [9], the correctness of the abstraction relation is only formally proven. Consequently, the correctness of the algorithm is guaranteed with complete rigor.

As mentioned above, our approach is light compared with the conventional approach in which the correctness of the concrete algorithm is formally proven using a theorem prover. Such a proof is usually inductive, and finding an appropriate invariant for an inductive proof requires human invention. Defining an abstraction relation, although it is often ad-hoc, requires less human invention, and formally proving its correctness is easy compared with constructing an inductive proof. We will discuss more about the merits of our approach in Section 6.

2 Preliminaries

In this section, we recall our framework of abstract model checking for the safety.

We assume that a finite set of atomic commands is given. Each atomic command is interpreted to make a transition between states. Formally, a transition is a binary relation between states.

A program is in one of the following forms: atomic command, sequential execution of programs, nondeterministic choice of programs, if-statement, and while-statement. Based on the interpretation of atomic commands, a program also makes a transition between states. So, the set of states and a program made of atomic commands defines a state transition system.

We assume two kinds of states, concrete states and abstract states, and give two interpretations to each atomic command. The concrete interpretation of an atomic command is a transition between concrete states, while the abstract interpretation defines a transition between abstract states. The set of concrete states is written C, and that of abstract states A. Formally, the concrete interpretation of an atomic command a is a binary relation on C, which is written $[\![a]\!]_C \subseteq C \times C$. Similarly, the abstract interpretation is written $[\![a]\!]_A \subseteq A \times A$.

The abstraction relation between an abstract state and a concrete state is given by $\alpha \subseteq A \times C$.

The safety of a state transition system means that the system never falls into a unsafe state from the initial states. The set of initial concrete states and that of initial abstract states are written $I_C \subseteq C$ and $I_A \subseteq A$, respectively. The set of safe concrete states and that of safe abstract states are written $S_C \subseteq C$ and $S_A \subseteq A$, respectively.

Our final goal is to show the safety of the concrete system. It can be verified by model checking of the concrete state transition system, which is, however, not feasible if the number of concrete states is too large or infinite. Because the number of abstract states is relatively smaller than that of concrete states, model checking of the abstract system may be feasible. If the safety of the abstract system guarantees that of the concrete one, model checking of the safety of the abstract one is sufficient for the final goal. The following theorem that guarantees this.

If $\alpha, I_C, I_A, S_C, S_A$ satisfy

- $y \in I_C \Rightarrow \exists x \in I_A, (x, y) \in \alpha$,

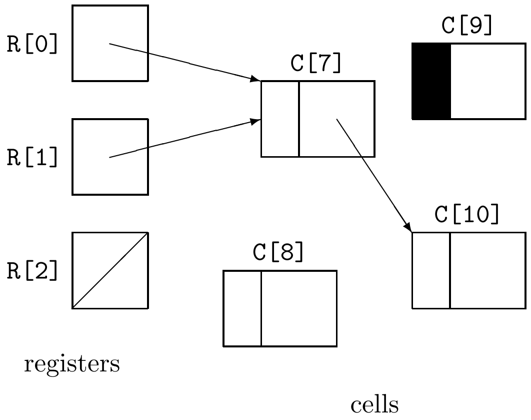

Figure 1: registers and cells

- $x \in S_A, (x, y) \in \alpha \Rightarrow y \in S_C$,

  and $(\alpha; [\![a]\!]_C) \subseteq ([\![a]\!]_A; \alpha)$ holds for every atomic command a, then for any program, the safety of the abstract system implies that of the concrete one.

In this theorem, the operator ";" composes two relations. The last formula can be rewritten as follows.

$$\exists y.((x, y) \in \alpha \land (y, z) \in [\![a]\!]_C) \Rightarrow \exists w.((x, w) \in [\![a]\!]_A \land (w, z) \in \alpha)$$

3 Concurrent Garbage Collection

In this section, we recall our abstract model checking of concurrent garbage collection. We deal with the so-called *on-the-fly garbage collector* formulated by Dijkstra *et al.* [3].

3.1 Cells and Registers

The heap for cells is formally defined as a function (or an array) from cell indices to cells. We do not specify the set of cell indices at the moment. We use `C` to denote the heap, and `C[i]` to denote the `i`-th cell.

```
C : heap = cell_index -> cell
```

A cell consists of its color and its fields that hold pointers to other cells. A pointer to a cell is a cell index or the null pointer `nil`. In this study, for the sake of simplicity, a cell has only one field. The set of cells, denoted by `cell`, is defined as follows.

```
cell = color * (cell_index + {nil})
color = {free, white, gray, black}
```

In the following, the pointer stored in the unique field of the `i`-th cell is denoted by `C[i].f`.

In this model, a list of free cells is not maintained, but a free cell has the color `free`. By this simplification, a cell has one of the four colors.

Registers are used by the mutator for manipulating cells. They are also used by the collector as the root for marking cells. In order to make the model closer to real implementations, we introduced a set of registers. In the following, `R` denotes a function (or an array) from register indices to registers, and `R[i]` the `i`-th register. Each register holds a pointer to a cell.

`R[i] := nil`	
`C[R[i]].f := nil`	
`R[i] := j`	(Only when `C[j]` is free.) Make `C[j]` gray.
`R[i] := R[j]`	
`R[i] := C[R[j]].f`	If `C[C[R[j]].f]` is white, make it gray.
`C[R[i]].f := R[j]`	

Table 1: mutator operations

`shade`	Make each directly reachable cell gray, if it is white. Then, go to the `mark` step.
`mark`	Choose a gray cell and make it black. If the cell refers to a white cell, make it gray.
`mark`	If there is no gray cell, go to the `append` step.
`append`	Choose a white cell and make it free.
`append`	If there is no white cell, go to the `unmark` step.
`unmark`	Choose a black or gray cell and make it white.
`unmark`	If there is no black or gray cell, go to the `shade` step.

Table 2: collector operations

```
R : register
register = register_index -> (cell_index + {nil})
```

A cell is called *directly reachable* if its index is stored in some register. A cell is called *reachable* if it is accessible through a chain of pointers from some register.

3.2 Mutator

The mutator has operations in Table 1. Each operation modifies the value of a register or the field of a cell. Some operations also change the color of a cell as specified in the comments in Table 1.

These operations are indivisible in that each causes a single state transition in the entire system.

3.3 Collector

The collector has four states. In order to distinguish the state of the collector from that of the entire system, the collector state is called the *collector step*. The collector has four steps: `shade`, `mark`, `append` and `unmark`. The operations allowed in each step are summarized in Table 2.

Notice that the entire shading operation is indivisible. It makes all the white and directly reachable cells gray at once.

3.4 Abstract Cells

We first define abstractions of cells. An *abstract cell* is a data structure consisting of some attributes about concrete cells. In this paper, the set of attributes is the following, which is enough for verifying the safety of the on-the-fly garbage collector:

(A1) the color of a cell,

(A2) the flag expressing whether a cell is directly reachable from a register,

(A3) the flag expressing whether a cell is (not necessarily directly) reachable from a register, and

(A4) the color of the cell that a cell refers to, or `nil`.

The first attribute is a color. The second and third are boolean values. The last attribute is either a color or the `nil` pointer. If the field of a cell holds the index of some cell, this attribute is the color of the cell having the index. Otherwise, it is `nil`.

3.5 Abstract Heaps and Abstract States

An *abstract heap* is simply a set of abstract cells. An abstract heap is an abstraction of a concrete heap, if for each concrete cell in the concrete heap, there exists an abstract cell in the abstract heap such that the concrete cell satisfies all the attributes of the abstract cell.

An *abstract state* of the entire system is a pair of a collector step and an abstract heap. Note that the values of the registers are not explicitly represented in an abstract state. They are implicit in the attributes of abstract cells.

State transitions within a same collector step does not depend on the *non-existence* of cells satisfying certain attributes. They only depend on the *existence* of some cells. Therefore, at the abstract level, they are monotone with respect to abstract heaps (where heaps are ordered by the set inclusion). Moreover, as we see in the next subsection, abstract transitions within a same collector step does not decrease abstract heaps. Therefore, we can merge the heaps of all abstract states having the same collector step. This means that we should prepare only one abstract heap for each collector step.

3.6 Abstract Transitions

State transitions in the concrete system should be abstracted in such a way that if a concrete transition changes state s to state s' and t is an abstraction of s, then there exists an abstract transition that changes t to some t', where t' is an abstraction of s'.

It is relatively easy to abstract the mutator's transitions and the collector's transitions within a same collector step. For example, consider the mutator operation `R[i]:=nil` of the on-the-fly garbage collector. This operation is abstracted by the following operation on an abstract heap:

> For each directly reachable abstract cell, add a new cell with the same attributes as those of the cell except that it is not directly reachable.
>
> For each reachable abstract cell, add a new cell with same attributes as those of the cell except that it is not reachable (and hence is not directly reachable).

Abstractions of the collector's transitions that change the collector step are not trivial. For abstracting those transitions, we introduce procedures we call filters. A *filter* is a procedure that deletes inconsistent abstract cells from an abstract heap. In this study, we define one filter for each transition that changes the collector step. We further introduce a general filter that can be combined with those filters.

The filters that are specific to some step changes are listed in Table 3. The concrete transition from `mark` to `append`, for example, requires the condition that there is no gray cell. Therefore, its abstraction deletes gray cells from the abstract heap of the `mark` step before adding its cells to the abstract heap of the `append` step.

These filters are natural abstractions of step changes. But the abstract system that adopts just these filters reaches an error state, i.e., a reachable free abstract cell is generated. The scenario is as follows. We suppose that the following two concrete cells exist in the heap of the `shade` step. One is gray and directly reachable from a register, and refers to the other cell, which is white and refers

`shade` → `mark`	Delete white and directly reachable cells.
`mark` → `append`	Delete gray cells, cells that refer to gray cells.
`append` → `unmark`	Delete white cells and cells that refer to white cells.
`unmark` → `shade`	Delete gray cells, black cells, and cells that refer to gray or black cells.

Table 3: filters specific to step changes

to nothing. The corresponding abstract cells are $\langle \text{gray}, \text{true}, \text{true}, \text{white} \rangle$ and $\langle \text{white}, \text{false}, \text{true}, \text{nil} \rangle$ (values of the tuple correspond to the attributes (A1)–(A4), respectively). Filter "`shade` → `mark`" deletes neither of them, and the abstract collector in the `mark` step generates some cells from them, but it does not delete any cell. Remember that abstract transitions other than filters do not delete abstract cells. Filter "`mark` → `append`" then deletes $\langle \text{gray}, \text{true}, \text{true}, \text{white} \rangle$ and some other cells, but it does not delete $\langle \text{white}, \text{false}, \text{true}, \text{nil} \rangle$, because this abstract cell has no gray attributes. The abstract collector of the `append` step finally generates $\langle \text{free}, \text{false}, \text{true}, \text{nil} \rangle$ from $\langle \text{white}, \text{false}, \text{true}, \text{nil} \rangle$. This is an unsafe cell.

In this scenario, however, the concrete cell corresponding to the abstract cell $\langle \text{white}, \text{false}, \text{true}, \text{nil} \rangle$ should have changed its attributes during the `mark` step, and should no longer correspond to the abstract cell.

Fortunately, this inconsistency can be detected only by examining the abstract heap. The abstract cell $\langle \text{white}, \text{false}, \text{true}, \text{nil} \rangle$ is reachable but is not directly reachable. This means that there should be some other reachable cells that refer to a white cell. In the above scenario, this is not the case because filter "`mark` → `append`" has deleted the abstract cell $\langle \text{gray}, \text{true}, \text{true}, \text{white} \rangle$.

For deleting inconsistent cells such as $\langle \text{white}, \text{false}, \text{true}, \text{nil} \rangle$ in the above scenario, we introduce a general filter concerning reachability. An abstract cell in an abstract heap is called *truely reachable,* if it satisfies one of the following conditions. (Notice that this is an inductive definition.)

- It is directly reachable.
- There exists another truely reachable abstract cell in the abstract heap whose color coincides with the attribute of the cell expressing *the color of the cell it refers to.*

By this filter, an abstract cell whose attribute for reachability is true is deleted if it is not truely reachable. (Note that this notion of true reachability is considered as an approximation of the inductive notion of real reachability defined on concrete cells.)

By using this filter, the previous scenario is revised as follows. The set of abstract cells obtained by filter "`mark` → `append`" contains $\langle \text{white}, \text{false}, \text{true}, \text{nil} \rangle$, but it is not truely reachable in this set, because $\langle \text{gray}, \text{true}, \text{true}, \text{white} \rangle$ has been deleted by filter "`mark` → `append`", so it is deleted by the filter concerning reachability. More precisely, the set contains neither $\langle \text{black}, \text{true}, \text{true}, \text{white} \rangle$ nor $\langle \text{black}, \text{false}, \text{true}, \text{white} \rangle$ because the collector in the `mark` step can not generate them in case of the on-the-fly algorithm.

3.7 Results

The abstract model checker was written in Python [13], and run on Pentium II (300MHz). The execution time was 2.794 seconds.

4 Formalization

We formalize the abstraction employed in Section 3 in HOL [5]. As mentioned in Section 2, we have to formalize the following data of the concurrent garbage collection algorithm.

- C : Concrete states of GC.
- I_C, S_C : Initial states and safe states of C.
- $[\![a]\!]_C$: Concrete transitions for all atoms.
- A : Abstract states of GC.
- I_A, S_A : Initial states and safe states of A.
- $[\![a]\!]_A$: Abstract transitions for all atoms.
- α : Abstract relation between A and C.

4.1 Concrete States

In Section 3, we introduced arrays of registers and cells in the concrete model of concurrent garbage collection algorithms. Each register contains a pointer, which is an index of a cell in the heap, or `nil`. Each cell contains a color and a pointer. The kinds of colors are *White*, *Gray*, *Black* and *Free*. The mutator and the collector run in parallel. The collector has four states. In order to distinguish the state of the collector from that of the entire system, the collector state is called the collector step. The collector has four steps: Shade, Mark, Append and Unmark. We formalize the color and the step as datatypes of HOL as follows.

```
Hol_datatype `color =  Free | White | Gray | Black`;
Hol_datatype `step = Shade | Mark | Append | Unmark`;
```

We use the `num` type, the type of natural numbers in HOL, for indices of the arrays. We use an `option` type to formalize the type of pointers. The `option` type of `num` is written `num option` in HOL. A term of the type `num option` is either `NONE`, or of the form `SOME n`. We regard `SOME n` as the index `n`, and `NONE` as `nil`.

The array of registers is a term of the following type.

```
num -> num option
```

It is the type of a function from `num` to `num option`. The array of cells is a term of the following type.

```
num -> color # num option
```

The operator `#` makes the direct product of two types. We use two variables `rb` and `hb` as the upper bounds of indices of registers and cells.

A state of the entire system is represented by values of the following five variables.

```
(rb:num) (hb:num) (s:step) (r:num->num option) (h:num->color#num option)
```

The right side of "`:`" shows the type of each variable. Since the values of `rb` and `hb` never change, the latter three variables are essential.

4.2 Initial States and Safe States

At the initial state of the system, the collector step is *Shade*, the contents of all registers are `nil`, the contents of all cells are `nil`, and the colors of all cells are *Free*. We define the predicate `initial` to judge whether a state is initial or not.

```
val initial_DEF = new_definition("initial_DEF",
``!(rb:num) (hb:num) (s:step) (r:num->num option) (h:num->color#num option).
   initial rb hb s r h = (s=Shade) /\ (!i. r i = NONE) /\ (!k. h k = (Free,NONE))``);
```

In the proof assistant and also in this paper, a term of HOL is surrounded part by "‘‘". The symbol "!" denotes the universal quantifier, and "/\" means conjunction. The juxtaposition `r i` means the application of the function `r` to the argument `i`. The term `(Free,NONE)` denotes the pair of `Free` and `NONE`.

The system is safe if and only if no free cells are reachable from registers by tracing pointers. The reachability is inductively defined as follows.

- If a register contains an index, the cell with the index is reachable.
- If a reachable cell contains an index, the cell with the index is reachable.

We define the predicate `direct_reachable` to judge whether a cell with the index `k` is contained by some register.

```
val direct_reachable_DEF = new_definition("direct_reachable_DEF",
‘‘!(rb:num) (hb:num) (r:num->num option) (h:num->color#num option) k.
   direct_reachable rb hb r h k = ?i.(i < rb) /\ (k < hb) /\ (r i = SOME k)‘‘);
```

In this definition, "?" denotes the existential quantifier.

We use the inductive definition package of HOL to define the predicate `reachable`, which judges whether the cell with the index `k` is reachable.

```
val (REACHABLE_thm1,REACHABLE_thm2,REACHABLE_thm3) =
  IndDefLib.new_simple_inductive_definition [
  ‘‘direct_reachable r h k
    ==> reachable r h k‘‘,
  ‘‘reachable (r:num->num option) (h:num->color#num option) k /\ IS_SOME (field (h k))
    ==> reachable r h (THE (field (h k)))‘‘];
```

In this definition, the function `field` extracts the contents of a cell. The predicate `IS_SOME` is true when the argument is not `NONE`. The function `THE` returns `x` from `SOME x`.

The predicate `safe` to judge whether the system is safe is defined as follows.

```
val safe_DEF = new_definition("safe_DEF",
‘‘!(rb:num) (hb:num) (s:step) (r:num->num option) (h:num->color#num option).
   safe rb hb s r h = !k. (k < hb) ==> ~ (reachable rb hb r h k)‘‘);
```

In this definition, "==>" means implication, and "~" negation.

4.3 Concrete Interpretation of Atomic Commands

The operations of the mutator and the collector are defined as atomic commands. The concrete interpretation of an atomic command is a relation between states before and after its execution. For each atomic command, we define a predicate that takes two states as arguments.

For example, the mutator's operator Rnew allocates the cell with index `k` and assigns it to the register with index `i`. The predicate `Rnew` is defined as follows.

```
val Rnew_DEF = new_definition("Rnew_DEF",
‘‘!(rb:num) (hb:num) (s:step) (r:num->num option) (h:num->color#num option)
   s’ r’ h’ i k.
   Rnew rb hb s r h s’ r’ h’ i k=
   ((i < rb) /\ (k < hb) /\ ~(s=Shade) /\ (color (h k)=Free)
    =>(s’ = s) /\ (!m. r’ m=((m=i) => (SOME k) | r m))
      /\ (!n. h’ n=((n=k) => (Gray,NONE) | h n))
    | (s’=s) /\ (r’=r) /\ (h’=h))‘‘);
```

The operator takes `i` and `k` as arguments in addition to those of the two states. In this definition, the infix operator " `=> |` " represents the if-then-else expression in HOL.

The predicates for other operations are defined similarly. All definitions are straightforward.

4.4 Abstract States

Now, we mention the abstract states. An abstract state, as defined in Section 3, is a pair of a collector step and an abstract heap. An abstract heap is a set of abstract cells. An abstract cell is a tuple consisting of the following four components.

- the color of the corresponding concrete cell,
- `nil` if the concrete cell contains `nil`, or the color of the cell whose index is contained in the concrete cell,
- the boolean value which represents the direct reachability from registers, and
- the boolean value which represents the reachability.

Therefore, an abstract cell is represented by values of the following four variables.

```
(c:color) (f:color option) (d_r:bool) (reach:bool)
```

A set of abstract cells, i.e., an abstract heap, is represented by a characteristic function. The variable `ah` representing an abstract heap has the following type.

```
color->(color option)->bool->bool->bool
```

Therefore, an abstract state is represented by the following two variables.

```
(as:step) (ah:color->(color option)->bool->bool->bool)
```

4.5 Abstract Initial States and Abstract Safe States

The abstract initial states and the abstract safe states were defined in Section 3. We translate them as follows. We define two predicates, `abs_initial` and `abs_safe`, as follows. The former defines abstract initial states, and the latter safe states.

```
val abs_initial_DEF = new_definition("abs_initial_DEF",
``!(as:step) (ah:color->color option->bool->bool->bool).
   abs_initial as ah =
   (as=Shade) /\
   (!c f d_r r. ah c f d_r r = (c=Free) /\ (f=NONE) /\ (d_r = F) /\ (r = F))``);
val abs_safe_DEF = new_definition("abs_safe_DEF",
``!(as:step) (ah:color->color option->bool->bool->bool).
   abs_safe as ah = (!f d_r. ~ (ah Free f d_r T))``);
```

4.6 Abstract Interpretation of Atomic Commands

For each atomic command, we define the predicate representing its abstract interpretation. For example, the predicate `abs_Rnew`, which represents the abstract interpretation of Rnew, is defined as follows. The variables with dash-mark represent the abstract state after the operator is executed.

```
val abs_Rnew_DEF = new_definition("abs_Rnew_DEF",
``!(as:step) (ah:color->color option->bool->bool->bool) as' ah'.
   abs_Rnew as ah as' ah' =
   (as=as') /\
   (!c f d_r r.
     ah' c f d_r r
     = (ah c f d_r r) \/
       (ah c f T   T /\ (d_r = F) /\ (r = F)) \/
```

```
    (ah c f T   T /\ (d_r = F) /\ (r = T)) \/
    (ah c f F   T /\ (d_r = F) /\ (r = F)) \/
    ((?f' d_r' r'.ah Free f' d_r' r')
     /\ (c = Gray) /\ (f = NONE) /\ (d_r = T) /\ (r = T)))``);
```

The abstract interpretations of some operations are rather complicated. So, the formal definitions of them are long. But that formalization is straightforward.

4.7 Abstract Relation

Finally, we have to define the abstraction relation in HOL. It defines how a concrete state is abstracted to an abstract one. The four attributes of an abstract cell can be calculated from a concrete state. We first define the predicate to judge whether an abstract cell, `c f d_r reach`, is derived from the concrete cell whose index is `k`, where the heap is `h` and the register array is `r`.

```
val abstract_cell_DEF = new_definition("abstract_cell_DEF",
``!(rb:num) (hb:num) (r:num->num option) (h:num->color#num option) k
   (c:color) (f:color option) (d_r:bool) (reach:bool).
   abstract_cell rb hb r h k c f d_r reach=
   (k<hb) /\ (c = color (h k)) /\
   (f= (IS_SOME (field (h k))
        => SOME (color (h (THE (field (h k)))))
        |  NONE))/\
   (d_r = direct_reachable rb hb r h k) /\ (reach = reachable rb hb r h k)``);
```

In this definition, the function `color` denotes the projection that extracts a color from the contents of a cell.

The abstraction relation between a concrete state, `rb hb s r h`, and an abstract state, `as ah`, is now defined as follows.

```
val abstract_relation_DEF = new_definition("abstract_relation",
``!(rb:num) (hb:num) (s:step) (r:num->num option) (h:num->color#num option)
   (as:step) (ah:color->color option->bool->bool->bool).
   abstract_relation rb hb s r h as ah =
   (s = as) /\ (!d_r reach c f.
                (?k. abstract_cell rb hb r h k c f d_r reach)
                ==> ah c f d_r reach)``);
```

5 Formal Proof

We have formalized the necessary data as above. The required conditions to guarantee the validity of abstract model checking are formulas of first order logic as mentioned in Section 2. Of course, formalization of first order formula is easy in HOL. Here are formalized conditions in HOL. The condition about initial states:

```
``!(rb:num) (hb:num) (s:step) (r:num->num option) (h:num->color#num option)
   (as:step) (ah:color->color option->bool->bool->bool).
   initial rb hb s r h
   ==> ?as ah.(abs_initial as ah /\ abstract_relation rb hb s r h as ah)``
```

The formula representing the condition about safe states:

```
``!(rb:num) (hb:num) (s:step) (r:num->num option) (h:num->color#num option)
   (as:step) (ah:color->color option->bool->bool->bool).
   safe rb hb s r h /\ abstract_relation rb hb s r h as ah
   ==> abs_safe as ah``
```

The formula for atomic command Rnew:

```
``!(rb:num) (hb:num) (s:step) (r:num->num option) (h:num->color#num option) s' r' h'
   (as:step) (ah:color->color option->bool->bool->bool) as' ah'.
  ?s'' r'' h'' i k.
   abstract_relation rb hb s'' r'' h'' as ah /\ Rnew rb hb s'' r'' h'' s' r' h' i k
  ==>
  ?as'' ah''. abs_Rnew as ah as'' ah'' /\ abstract_relation rb hb s' r' h' as'' ah''``
```

The formulas for other atomic commands are given by replacing "Rnew" with the corresponding atomic command name.

We prove these formulas by backward reasoning. We use Boomborg-HOL interface [12]. The formula of the condition of initial states is proven by the next short tactic.

```
ZAP_TAC our_ss [] THEN
EXISTS_TAC
``\c (f: color option) d_r r. (c=Free) /\ (f=NONE) /\ (d_r = F) /\ (r = F)`` THEN
ZAP_TAC bool_ss [] THEN
ONCE_REWRITE_TAC[REACHABLE_thm3] THEN
ZAP_TAC our_ss []
```

The `ZAP_TAC` is a strong semi-automatic reasoning tactic in HOL. The tactics for other formulas are longer. But the construction of tactics was straightforward.

6 Discussion

We formalized the concrete model and the abstract model of the on-the-fly GC algorithm in HOL. The formalization was straightforward thanks to many datatypes of HOL and its inductive definition library. The construction of proofs was also straightforward.

Because we formally proved that the abstraction used in Section 3 satisfies the conditions that guarantee the validity of abstract model checking, the safety of the on-the-fly GC algorithm is now completely formally verified.

The concurrent GC algorithms have been formally verified in several studies [7, 8, 4, 10]. Those except [7] did not use model checking. They found some invariants by inspecting the concurrent GC algorithm, and then constructed the proof on theorem provers. Havelund [7] verified the algorithms by model checking only with a finite model.

Our verification was done by the following three steps.

1. An abstract model was designed.
2. The abstract model was verified by model checking.
3. The validity of our abstraction was formally proven.

We felt that these steps were light tasks in the following sense.

1. The design was simple.
2. Model checking was fully automatic.
3. Formal proving was straightforward.

As a result of our experience, we can conclude that abstract model checking is an effective verification technique.

References

[1] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[3] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[4] G. Gonthier. Verifying the safety of a practical concurrent garbage collector. In *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 462–465. Springer-Verlag, 1996.

[5] M. Gordon and T. Melham. *Introduction to HOL: A Theorem-proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[6] M. Hagiya and K. Takahashi. A verification of concurrent garbage collection by abstract model checking. In *Second Workshop on Systems for Programming and Applications*, 1999. `http://www.osss.is.tsukuba.ac.jp/spa99proc`. in japanese.

[7] K. Havelund. Mechanical verification of a garbage collector. In *Fourth International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMTPTA '99)*, pages 1258–1283, 1999.

[8] P. B. Jackson. Verifying a garbage collection algorithm. In *Theorem Proving in High Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 225–244. Springer-Verlag, 1998.

[9] Y. Kinoshita and K. Takahashi. Cumulatives for safety. *Electronic Notes on Theoretical computer Science*. to appear. available from `http://staff.aist.go.jp/k.takahashi/`.

[10] D. M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6:359–390, 1994.

[11] K. Takahashi. *Abstraction and Search in Verification by State Exploration*. PhD thesis, The University of Tokyo, 2002. available from `http://staff.aist.go.jp/k.takahashi/`.

[12] K. Takahashi and M. Hagiya. Proving as edition HOL tactics. *Formal Aspects of Computing*, 11:343–357, 1999.

[13] G. van Rossum. Python tutorial, 1998. `http://www.python.org/doc/tut/tut.html`.