情報処理学会論文誌　プログラミング　Vol. 2　No. 4　13–32 (Aug. 2009)

リターンバリア型実時間ごみ集めの抽象モデル検査

藤川浩光[†1,*1]　馬谷誠二[†1]
八杉昌宏[†1]　湯淺太一[†1]

実時間ごみ集めアルゴリズムの代表的なものとしてスナップショット方式があげられるが，このアルゴリズムのルート挿入を改良したものに，リターンバリアという手法がある．スナップショット方式のルート挿入では，プログラムの実行を止める必要があった．この停止時間はスタックの大きさに依存していたが，リターンバリア方式ではスタックをフレーム単位で分割してルート挿入することにより，より停止時間を短縮することができる．我々は簡略化したリターンバリア型実時間ごみ集めの安全性の証明に成功した．ここでの簡略化とは，通常ならヒープ領域のセルが 2 つ以上のポインタを持てるところを，1 つに限定したということである．さらに，実際の処理系ではルート挿入の間にもスタックが伸び縮みするが，証明を複雑にする可能性があったので，スタックが伸び縮みしない処理系モデルを対象とした．安全性とは使用中のセルを誤ってごみとして回収しないという保証である．現在，様々な実時間ごみ集めアルゴリズムが提案されており，プログラムの停止時間が短くなるよう改良されている．しかしアルゴリズムは複雑さを増しており，一見して安全性があるのか分かりにくいことも多い．モデル検査で安全性を証明されたアルゴリズムであれば，より安心して実用的に使うことができる．先行研究として，簡略化したスナップショット方式ごみ集めの安全性が，抽象モデル検査とよばれる手法で証明されている．本研究ではこれに基づいて抽象モデル検査を行い，リターンバリアを使う場合と使わない場合の比較を行った．また，抽象化が正しく行われていることも定理証明系である Isabele/HOL を用いて証明した．

Abstract Model Checking of Real-time Garbage Collection with Return Barriers

Hiromitsu Fujikawa,[†1,*1] Seiji Umatani,[†1]
Masahiro Yasugi[†1] and Taiichi Yuasa[†1]

Snapshot GC (garbage collection) is one of the most popular real-time GC algorithms and “return barrier” is the method that improves root insertion of the snapshot GC algorithm. The original snapshot GC algorithm required to stop the running program during root insertion. This pause time of programs depends on the size of the stack. GC with return barriers can shorten the pause time by dividing root insertion. We proved the safety of a simplified version of the real-time GC with return barriers. Here, “simplified” means two things. One is that we assume each cell in the heap area contains at most one pointer whereas each cell in general has multiple pointers. The second is that we used a processor model that does not support dynamic changes of the stack size. “Safety” guarantees that GC never collects cells in use as garbage. So far, various real-time GC algorithms have been proposed, which reduce the pause time of programs. However, these algorithms are complex and it is often difficult to judge their safety at a glance. We can safely use such an algorithm if we can prove its safety with model checking. We checked the algorithm based on this method, and compared the abstract systems for GC with return barriers and for the original snapshot GC. We also proved the validity of abstraction using the theorem prover Isabelle/HOL.

1. はじめに

ごみ集めとはメモリ管理技術の 1 つであり，使われることのない不要なオブジェクト（ごみ）を探して，そのメモリ領域を再利用できるようにする処理である．メモリの解放などの処理がプログラマに任されているプログラミング言語では，いちいちメモリの解放を指定しなければならないのはプログラマの負担にもなり，複雑なソースを生み出してエラーを起こす原因にもなる．これに対して Lisp や Java などの処理系にはあらかじめごみ集めの処理が組み込まれており，自動的にごみを回収してその領域を再利用する．これによって，プログラマはメモリ管理を気にすることなくソースコードを記述することができる．

ごみ集めアルゴリズムは様々なものが考案されておりそれぞれ特徴を持っているが，ハードウェア技術の進歩によるアプリケーションの応答速度の向上により，プログラムの中断時間をなるべく短くするようなごみ集めが求められるようになった．機械制御などの実時間性を求められる分野では，数秒間のプログラムの停止が予期せぬエラーにつながることもあり，一般的なごみ集めを実装した処理系ではアプリケーションを記述することができなかった．

そこで実時間ごみ集めというアルゴリズムが提案されている[4),11),12)]．この手法の特徴の

†1 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University
∗1 現在，任天堂株式会社
Presently with Nintendo Co., Ltd.

1つは，ごみ集めの工程をいくつかのフェーズに分けて，ごみ集めとプログラムの実行を交互に行う点である．通常のごみ集めならば，ごみ集めの一連の動作の間はプログラムを停止させていなければならなかったが，実時間ごみ集めの場合は1回の停止時間をより短くすることができる．代表的な実時間ごみ集めとしてスナップショット方式[11),12)]があげられる．この方式は，実時間性を持たない一般的なごみ集めを採用している処理系への応用が可能である，専用のハードウェアを用いなくてもプログラムの実行効率の低下が少ないなどの利点があり，汎用計算機上のシステムに適している．

しかしスナップショット方式では，ルート挿入（ルート領域から直接参照されるすべてのオブジェクトをマーク処理のために登録する処理）の際にプログラムを停止させる必要があり，この停止時間はルート領域の大きさに依存する．ルート領域はレジスタやスタックから構成されているが，スタックはプログラムの実行によってその大きさが変わるため，スナップショット方式では停止時間が大きく変動する可能性がある．この問題点を改良したのがリターンバリア方式[13)]である．このごみ集めはスナップショット方式をもとにした実時間ごみ集めであるが，スタックのルート挿入を分割して行うことで，ルート挿入にともなう停止時間を短くする工夫がなされている．これらのごみ集めの詳細は2章で述べる．

ごみ集めアルゴリズムが必ず満たすべき性質の1つに安全性というものがある．安全性とは，生きている（ルートから参照をたどってアクセスされる可能性のある）オブジェクトを誤って回収してしまうことがないという保証であり，安全性が保証されていないごみ集めはプログラムの処理にエラーを発生させる可能性がある．実時間ごみ集めは通常のごみ集めアルゴリズムよりも複雑であり，一見してその安全性が分かりにくい．これらの安全性を論理的に証明することによって，安心して処理系に実装し，実用的に使うことが可能になる．

本研究では簡略化したリターンバリア方式ごみ集めの安全性を，抽象モデル検査[3),5)]を用いて証明した．ここでの簡略化とは，通常ならヒープ領域のセル（オブジェクト）が2つ以上のポインタを持てるところを，1つに限定したということである．また，実際の処理系ではルート挿入の間にもスタックが伸び縮みするが，証明を複雑にする可能性があったので，スタックが伸び縮みしない処理系モデルを対象とした．抽象モデル検査とはリンク構造などを抽象化したものをモデル検査[2)]する手法で，状態数が多すぎたり，上限がなかったりするシステムの検証に有効である．先行研究として，簡略化されたスナップショット方式ごみ集めの安全性の検証が抽象モデル検査によってなされており[8)–10)]，我々はこの研究をもとにリターンバリア方式ごみ集めの安全性の検証を行った．抽象化や抽象モデル検査については3章と4章に記載する．

仮に抽象モデル検査が成功したとしても，抽象化が間違っていればその検証は意味をなさない．抽象化が妥当かどうかは自明ではなく，定理証明支援システムなどを用いて証明するべきである．実際に我々も，抽象モデル検査が成功した後に，妥当性が証明できず正しい抽象化を考え直すなど，トライアンドエラーを繰り返して証明を成功させた．本研究では抽象化が妥当であることを定理証明系 Isabelle/HOL[1),7)]を用いて証明した．これらのIsabelle/HOL や妥当性の証明についての記述は5章にあり，6章では証明に失敗するもととなった原因やミスなどにも触れる．7章では今後の課題に関することを述べて，簡略化しないリターンバリア方式の安全性の証明へのアプローチを考察する．

2. 実時間ごみ集め

計算機のメモリ領域はルート領域とヒープ領域に分けることができる．ルート領域は実行中のプログラムが直接アクセスできるデータ領域であり，スタックやレジスタが主である．ヒープ領域とはオブジェクトが配置される領域であり，ルートが保持するオブジェクトへのポインタを介して，オブジェクトの内容を間接的にアクセスできる

本章では，まずごみ集めの基本的なアルゴリズムであるマーク・スイープ方式を説明し，次に，マーク・スイープ方式をもとにした実時間ごみ集めであるスナップショット方式について述べ，さらにそのスナップショット方式をもとにしたリターンバリア方式について述べる．なお以下ではごみ集め研究の慣例に従い，オブジェクトをセルとよぶことにする．

2.1 マーク・スイープ方式ごみ集め

マーク・スイープ方式[6)]のアルゴリズムは，次の2つのフェーズに分かれている．

- **Mark**：ルート領域からたどることのできるセルをすべてマークする．
- **Sweep**：マークのついていないセルを回収し，他のセルのマークをすべて消す．

図1 はメモリ領域のリンク構造を表している．左の縦の領域がルート領域であり，ばらばらに散らばっているのがヒープ領域のセルである．各セルの左側の部分はそのセルの状態を表しており，マークされているセルは黒，フリーのセル（ごみ集めによって回収されたセル）は斜線，その他のセルは白で示している．Mark フェーズが終了すると，図1のようにルート領域からたどることができるセルはすべてマークされており，ルート領域からたどることができないセルはマークされないままである．これらのセルはもう使うことができないので，すべてごみである．よってこれらのセルを回収することでごみ集めができる．

2.2 実時間ごみ集め

マーク・スイープ方式のような通常のごみ集めは，プログラムの実行を中断してごみ集め

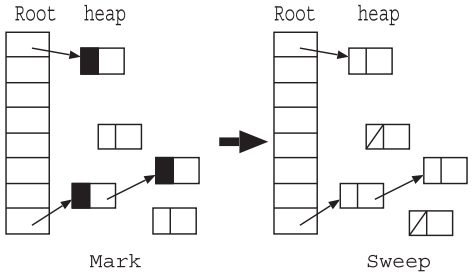

図 1　マーク・スイープ方式ごみ集め
Fig. 1　Mark & Sweep GC.

処理を行う．しかし大規模なシステムの場合，1 回のごみ集めに数秒かかることもあり，インタラクティブな人工知能システムやロボット制御などのアプリケーションにおいては，停止時間の長さが問題となる．プログラムの停止時間がより短いごみ集めとして，実時間ごみ集めのアルゴリズムが提案されている．実時間ごみ集めは，ごみ集めの工程をいくつかの小さな処理に分けて，ごみ集めとプログラムの実行を交互に行い，プログラムの中断時間を短くするというものである．通常ならば「実時間性を持つ」という言葉は「ある一定の時間内に処理が終わることを保障する」と定義する場合が多いが「実時間ごみ集め」における実時間とはこの定義と異なることに注意する．

2.2.1　スナップショット方式

スナップショット方式はマーク・スイープ方式をベースとして，プログラム（ミューテータ）の実行とごみ集め（コレクタ）の処理が交互にできるように改良したものである．

この方式は，ごみ集め開始時にごみであったセルを回収する．ごみ集め開始時に使用中であったセルは，次のごみ集めまでは回収されることはない．ごみ集め中にごみになる場合は，その回のごみ集めでは回収されないが，次のごみ集め開始時にはごみとなっているので，必ず回収される．

スナップショット方式を説明するために，セルの状態として前述の黒，白，フリーに加えて，灰色を追加する．ごみ集め開始時は，フリーでないセルはすべて白である．

スナップショット方式の工程を次の 3 つのフェーズに分ける．

- **Shade**（ルート挿入）：ルート領域から直接参照されているセルをすべて灰色にする．
- **Mark**：灰色のセルが残っている間，そのいずれかを黒にし，そのセルが白いセルを参照しているなら灰色にする．

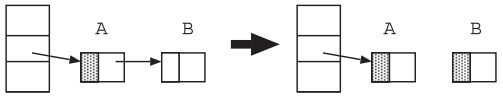

図 2　ライトバリア
Fig. 2　Write barrier.

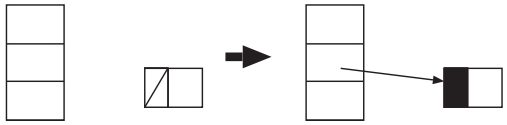

図 3　セルの割り当て
Fig. 3　Cell allocation.

- **Sweep**：ヒープ領域の各セルに対して，それが白ならばフリーにし，黒ならば（次回のごみ集めのために）白にする．

マーク・スイープ方式の Mark フェーズのうち，ルート挿入の処理を Shade という独立したフェーズとしている．スナップショット方式では，ルート挿入の間はミューテータを停止する必要があるためである．Mark フェーズの残りの処理と Sweep フェーズの処理は，少しずつ，ミューテータの処理と交互に行うことができる．

ごみ集め開始時に使用中であったセルを回収しないためには，ミューテータにライトバリア（write barrier）を設ける必要がある．たとえば，**図 2** のようにミューテータによってセル A のポインタが書き換えられたとする．スナップショット方式では，ルート挿入時にルートからセルをたどって参照できるセルは回収してはいけない．よってセル B とそこからたどることができるセルも最終的にはマークしなければいけない．この条件を満たすためには，ポインタのつけ替え時に注意すればよい．図 2 のようにミューテータがセルのポインタを書き換えるときに，もともと指されていたセルが白ならばこのセルを灰色にする．この操作によってこのセル以降のセルを確実にマークすることができる．ポインタつけ替え時のこの処理をライトバリアとよぶ．

新しくセルを割り当てる際にも注意が必要である．Mark フェーズ中にミューテータが新しいセルを要求した場合，フリーセルのいずれかを割り当てる必要があるが，そのセルは白ではなく黒にする（**図 3** 参照）．こうすることによって，そのセルが Sweep フェーズで誤って回収されることを防ぐ．同様の処理が，Sweep フェーズ中にも必要であるが，割り当てられたセルが Sweep フェーズ終了時には白になっていなければならない．通常の実装では，

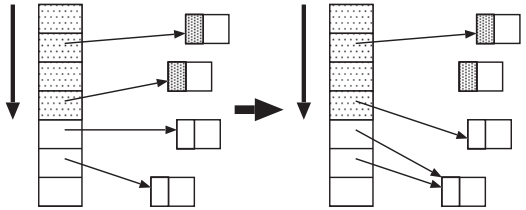

図 4　リターンバリアがない場合の失敗例
Fig. 4　Failure example without return barrier.

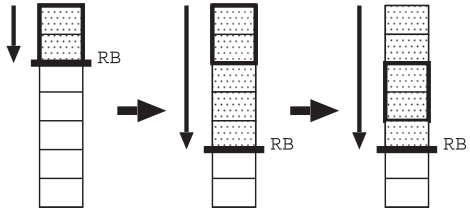

図 5　リターンバリア
Fig. 5　Return barrier.

Sweep フェーズはヒープ領域をスキャンすることによって実現されているので，割り当てるセルがスキャン済みの領域にあれば白に，未スキャンの領域にあれば黒にすればよい．

2.2.2　リターンバリア方式

スナップショット方式は，ルート挿入時にミューテータを中断させる必要があった．ルート領域のうち大部分を占めるのは，レジスタやシステムの大域変数などの固定領域ではなく，プログラムの実行によって大きさが変動するスタックである．ルート挿入の際の停止時間はスタックの大きさによって変動するため，停止時間が長くなることがある．この点を改良するために考えられた機構がリターンバリア（return barrier）である．以下では，リターンバリアを実装したスナップショット方式ごみ集めをリターンバリア方式ごみ集めとよぶことにする．

この方式はルート挿入以外はスナップショット方式と同じ操作をして，ライトバリアもスナップショット方式と同じものを使う．また，スタック以外のルート領域については，スナップショット方式と同様にミューテータを中断させてルート挿入を行う．スタック以外のルート領域は固定領域であり小さいので，この間の停止時間は無視してよい．

スタックのルート挿入について述べる．**図 4** は Shade フェーズ中のスタックを表している．図のスタックは下がボトムで上がトップである．図の左のように矢印の先端位置までルート挿入が進んでいるとする．ルート領域の網掛けの部分は，ルート挿入が終わったことを表している．この状態でミューテータが図の右のようにポインタを書き換えたとすると，Shade フェーズが終わっても，ルートから直接参照される白いセルが残り，Sweep フェーズでごみとして回収されてしまう．スナップショットと同様にライトバリアを用いれば，これは防ぐことができるが，スタックの内容は頻繁に書き換えられるため，オーバヘッドが大きくなってしまう．そこでスタックの特性である関数フレームを利用したリターンバリアという機構を用いて，ミューテータがつねにルート挿入済みの領域のみを参照することを保証するのがリターンバリア方式である．

通常，スタックは関数フレームから構成されており，その最上位に位置するのがカレントフレームである．ミューテータが操作するフレームはカレントフレームだけである．関数がリターンするときには，カレントフレームがポップされ，その直下のフレームが新しいカレントフレームとして使われる．逆に関数が呼び出されるときは，カレントフレームの上にフレームが積まれ，それが新しいカレントフレームとなる．その際に，以前からそこにあったポインタを（言語上の制約あるいは処理系がゼロクリアするなどによって）ミューテータは参照できないものと仮定する．

リターンバリア方式では，ルート挿入はスタックのトップからボトムに向かってフレーム単位で行う．コレクタによる数フレームずつのルート挿入とミューテータの実行を交互に行うことにより，ルート挿入にともなうミューテータの停止時間を短く抑えることができる．カレントフレームがつねにルート挿入済みの領域にあることを保証するために，ルート挿入が終わったフレームとまだ終わっていないフレームの間にバリアを設定する．**図 5** はリターンバリアを用いてルート挿入をしているスタックを表している．太枠はカレントフレームであり，RB の位置にある横棒がバリアである．関数がリターンすることによってカレントフレームがバリアを越えて移行する際には，ミューテータの実行を中断して次の数フレームをルート挿入してからミューテータの実行を再開する．

実際に処理系に実装する際には，入れ子関数がカレントフレーム以外のフレームを参照したり，非局所的脱出によって数多くのフレームを飛び越えてリターンしたりするような場合が生じるが，リターンバリアはこれらの場合にも対応可能である[13]．

3.　抽象モデル検査

スナップショット方式については，抽象モデル検査によって安全性が証明されており[10]，

表 1　ミューテータの操作
Table 1　Operations for the mutator.

操作	効果	色の変化
allocate(i,j)	`R[`i`].f := `j （`C[`j`]` がフリーのときのみ）	Unmark フェーズでは `C[`j`]` を白にし， それ以外のフェーズでは黒にする
rclear(i)	`R[`i`].f := nil`	なし
move(i, j)	`R[`i`].f := R[`j`].f`	
load(i, j)	`R[`i`].f := C[R[`j`].f].f`	
cclear(i)	`C[R[`i`].f].f := nil`	Mark フェーズ中（リターンバリア方式の場合は Shade フェーズ中も）でかつ代入前の `C[C[R[`i`].f].f]` が白なら それを灰色にする
store(i, j)	`C[R[`i`].f].f := R[`j`].f`	

その手法をベースにして，本研究はリターンバリア方式の安全性の検証を試みた．この章では，本研究の内容を理解するうえで必要な範囲で，スナップショット方式の抽象モデル検査を概説する．理論的な背景については，原論文[10] を参照されたい．

3.1　処理系のモデル

メモリ領域は，ルート領域 `R` とヒープ領域 `C` からなる．ルート領域は有限個のルートから構成されており，i 番目のルートを `R[`i`]` と表す．各ルート `R[`i`]` は，セルへのポインタまたは空ポインタ `nil` を持ち，その値を `R[`i`].f` で表す．実際の処理系では，ポインタ以外の値を持つこともあるが，ごみ集めの証明にはリンク構造以外は重要でないので，`nil` と見なす．ヒープ領域も有限個のセルからできており，i 番目のセルを `C[`i`]` と表す．簡単のため，各セル `C[`i`]` もポインタを 1 つだけ持てるものとし，その値を `C[`i`].f` と表す．セルへのポインタは，そのセルのインデックスで表す．たとえば，`C[R[`i`].f]` は，i 番目のルートが指しているセルであり，`C[R[`i`].f].f` は，そのセルが持っている値である．

リンク構造を変化させるためにミューテータが利用できる操作は，**表 1** に示す 6 つがある．cclear と store はセルの内容を書き換えるので，ライトバリアの対象となる．allocate は実際には割り当てるフリーセルを指定しないが，検証を容易にするために指定することにする．なんらかの方法でフリーなセルの情報をミューテータが取得できるものと考えるとよい．

処理系の初期状態では，すべてのセルがフリーであり，各セルの持つ値は `nil` である．そして，ルートの値もすべて `nil` である．

コレクタのフェーズは，2.2.1 項で述べた 3 つ（Shade，Mark，Sweep）のうち，Sweep フェーズをさらに次の 2 つに分ける．

- **Append**：白いセルをすべてフリーにする．
- **Unmark**：フリーでないセルをすべて白にする

前述のように，通常の実装では，Sweep フェーズはヒープ領域をスキャンすることによって実現されているが，このように 2 つのフェーズに分けることによって，スキャン済みと未スキャンの領域という概念を使わずに allocate 操作による色の変化を表 1 のように簡潔に記述することが可能になる．

Shade フェーズのコレクタの処理は，ミューテータを停止して一挙に行う必要があった．この意味で，Shade フェーズの処理はアトミック（atomic）である．Mark フェーズでは，1 つの灰色のセルを黒にし，それが白いセルをさしていれば灰色にする操作がアトミックである．Append フェーズでは 1 つの白いセルをフリーにする操作，Unmark フェーズでは 1 つのフリーでないセルを白にする操作がアトミックである．2 つの連続するアトミックな操作の間には，0 個以上の任意のミューテータ操作を実行することができる．

実際の処理系では，起動後しばらくはコレクタが動作せず，フリーセルが少なくなってから動作を開始する．しかし，コレクタ起動の条件を検証に持ち込みたくないので，このモデルでは，起動後ただちにコレクタが動作し始めるものとする．同じ理由で，コレクタは 1 回のごみ集め処理を終了するとただちに次のごみ集めを開始するものとする．また実際の処理系では，コレクタの操作が連続することによってミューテータの停止時間が長くなりすぎたり，逆にミューテータの操作が連続することによってごみの回収が遅れたりしないように，コレクタ操作の実行を制御するのが一般的であるが，このモデルでは，このようなタイミングには配慮しないことにする．

処理系は初期状態に始まり，ミューテータとコレクタの操作を繰り返して次々と状態を変化させていく．コレクタはまず Shade フェーズの処理を行い，Mark フェーズに移行する．Mark フェーズの操作を繰り返して灰色のセルがなくなれば Append フェーズに移行する．Append フェーズの処理を繰り返して白いセルがなくなれば Unmark フェーズに移行する．Unmark フェーズの処理を繰り返してフリーと白いセルだけになれば 1 回のごみ集め処理が完了し，再度 Shade フェーズに移行して次のごみ集め処理を行う．

このように定義した処理系モデルに対して，その安全性を抽象モデル検査の手法を用いて証明する．ここで，処理系が安全であるとは，使用中のセルを誤って回収してしまわないことを意味する．逆に安全でない処理系であれば，ルートからリンクをたどることによって到達できるセルのうち，フリーなものが存在する可能性がある．この状態をエラー状態とよぶ．いい換えれば，安全性の証明とは，処理系がエラー状態にならないことを証明することである．

3.2　抽象モデル

上の処理系モデルを抽象化し，状態数がきわめて少ない抽象モデルを定義する．

処理系の状態は，ヒープ領域の各セルの色，ヒープ領域およびルート領域のリンク構造，コレクタのフェーズ，によって特徴づけられる．このうち，フェーズ以外を，抽象セルの集合として以下のように抽象化する．

抽象セルとは，ルートあるいはセルをリンク構造も考慮して抽象化したものであり，ラベル σ，前方条件 ν，後方条件 β の三つ組 (σ, ν, β) で定義される．ここで，

$\sigma \in \{r, f, w, g, b\}$

$\nu \in \{\mathtt{nil}, f, w, g, b\}$

$\beta \subseteq \{r, [wgb]^*r, w^*g\}$

である．ラベル σ は，この抽象セルに抽象化されるのがルートであるか（$\sigma = r$ の場合）あるいは指定された色（f はフリー，w は白，g は灰色，b は黒）のセルであるかを表す．前方条件 ν は，抽象化されるセル（あるいはルート）の持つ値が空ポインタ nil であるか，あるいは指定された色のセルを指すポインタであるかを表す．後方条件 β は，抽象化されるセルがどのような経路で到達できるものであるかを正規表現の記法で表す．r はルートから直接指されていることを，$[wgb]^*r$ はルートから直接あるいは間接に指されていることを，それぞれ意味する．w^*g は灰色のセルから 0 個以上の白いセルのみを経由して到達できることを意味する．直感的には，ごみ集め開始時に $[wgb]^*r$ であったセルは，Shade フェーズによって w^*g となり，Mark 操作とライトバリアによって，Mark フェーズ終了時には黒になるので，Append フェーズで誤ってごみとして回収されることはない．

3 つの経路のうちのいずれかが後方条件に欠けている場合は，その抽象セルに抽象化されるセルは，その欠けた経路では到達できない．たとえば，後方条件に $[wgb]^*r$ が含まれていて r が含まれていない場合は，ルートから直接には参照されていないが，他のセルを経由して間接に参照されていることを意味する．なお，ルートはどこからも参照されないので，r は前方条件にはなりえず，ルートを抽象化した抽象セルは後方条件がつねに空集合である．

処理系の状態が与えられると，それぞれのセル（あるいはルート）は 1 つしかも唯 1 つの抽象セルに抽象化される．そのような抽象セルからなる集合によって，処理系の状態を抽象化する．この集合のことを抽象状態とよぶ．以下では，セル（あるいはルート）c を抽象化した抽象セルを $\overline{c}$ で表し，処理系の状態 X を抽象化した抽象状態を $\overline{X}$ で表すことにする．たとえば，処理系の初期状態（以下では X_S と記す）においては，すべてのセルがフリーで，どのセルの値もどのルートの値も nil であった，したがって，初期状態 X_S を抽象化した $\overline{X_S}$ は，2 つの抽象セルからなる次の抽象状態となる．

$\overline{X_S} = \{(r, \mathtt{nil}, \{\}), (f, \mathtt{nil}, \{\})\}$

また，処理系のエラー状態を抽象化した抽象状態には，ラベルが f で後方条件に $[wgb]^*r$ を含む抽象セル（以下，エラーセルとよぶ）が必ず含まれる．

3.3　抽象遷移

処理系がある状態 X のときにある操作 θ を実行した結果，状態が Y になったとする．これを次のように表記する．

$X \xrightarrow{\theta} Y$

この状態遷移に対応して，抽象状態は $\overline{X}$ から $\overline{Y}$ に変わる．つまり，θ は抽象状態 $\overline{X}$ を抽象状態 $\overline{Y}$ に遷移させたことになる．このように処理系の操作は，抽象状態を遷移させる操作と見なすことができる．

一般に，ある抽象状態 A に対して，抽象化すると A になる状態は 1 つとは限らない．また，同じ操作を実行しても，対象となるルートやセルが異なれば，結果となる状態は異なる．たとえば，rclear を適用するときに，クリアするルート（つまり rclear への引数）が異なれば，得られる状態は一般的には異なる．そこで，抽象状態 A に対して操作 θ が行う抽象遷移（$\overline{\theta}(A)$ と表記する）を次のように定義する．

$$\overline{\theta}(A) = \bigcup \{\, \overline{Y} \mid \exists X \text{ s.t. } \overline{X} = A \wedge X \xrightarrow{\theta} Y \}$$

つまり，A に抽象化できるすべての状態 X から θ によって遷移するすべての状態 Y を抽象化した抽象状態全体の和集合である．定義から，任意の状態 X，Y と操作 θ に対して，$X \xrightarrow{\theta} Y$ ならば $\overline{\theta}(\overline{X}) \supseteq \overline{Y}$ である．さらに，任意の抽象状態 A，B と操作 θ に対して，$A \supseteq B$ ならば，$\overline{\theta}(A) \supseteq \overline{\theta}(B)$ である．

処理系が安全でない場合，初期状態 X_S で始まり，あるエラー状態 X_E に達する状態列 $X_0, \ldots, X_n$ および操作列 $\theta_1, \ldots, \theta_n$ が存在し，

$X_S = X_0 \xrightarrow{\theta_1} X_1 \cdots X_{n-1} \xrightarrow{\theta_n} X_n = X_E$

が成り立つ．$\overline{X_S}$ で始まって，$\overline{\theta_1}, \ldots, \overline{\theta_n}$ を次々に適用していって得られる抽象状態の列 $A_0, \ldots, A_n$ を考える．

$A_0 = \overline{X_S}$

$A_i = \overline{\theta_i}(A_{i-1}) \quad (0 < i \leq n)$

上に述べたことから，すべての i（$0 \leq i \leq n$）に対して $A_i \supseteq \overline{X_i}$ が成り立つ．最後の $X_n (= X_E)$ はエラー状態だったから，それを抽象化した $\overline{X_n}$ はエラーセルを含み，したがっ

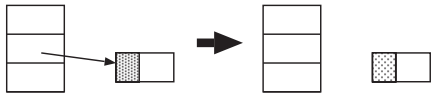

図 6　rclear
Fig. 6　rclear.

て，$A_n(\supseteq \overline{X_n})$ もエラーセルを含む．つまり，安全でない処理系に対しては，初期状態 X_S を抽象化した $\overline{X_S}$ で始まり，抽象遷移を繰り返してエラーセルを含む抽象状態に達することができる．逆に，どのように抽象遷移を繰り返してもエラーセルを含む抽象状態に達することができないことを示せば，その処理系は安全であることが保証される．これが，抽象モデル検査の原理である．

3.4　スナップショット方式の抽象モデル検査

本研究の最終目的は，抽象モデル検査の手法を用いてリターンバリア方式の安全性を検証することにある．この目的のために，まずスナップショット方式の検証を行った．原論文[10]には，検証の過程で必要となる詳細なデータが記載されておらず，検証を行った際の資料も残っていないとのことであった．そこでまず，分かる範囲で原論文に忠実にスナップショット方式の検証を行い，生成された抽象状態の個数など，原論文に記載されている結果と照合することによって，検証が正しく行われていることを確認し，その結果を利用してリターンバリア方式の検証に進むことにした．

3.4.1　抽象遷移の決定

処理系の操作 θ から，対応する抽象遷移 $\overline{\theta}$ を求めることは自明ではない．一般に抽象遷移は，抽象状態 A がある抽象セルを含む場合に，これこれの抽象セルを A に追加する，という形のルールで与えられる．抽象遷移によっては，ルールが 1 つだけの場合もあるし，複数のルールによって定義される場合もある．ルールに誤りや過不足があると検証が正しく行われない．現状では，ルールの導出は人間の能力に頼るしかなく，抽象遷移の決定が抽象モデル検査の最大の難関である．

例として，ミューテータ操作の 1 つである rclear を考える．rclear は，**図 6** に示すように，あるルートの値を `nil` にする操作である．この図を眺めながら，rclear が抽象状態 A にどのような影響を及ぼすかを導きださなければならない．まず明らかなのは，ラベルが r で前方条件が ν である抽象セルが A に含まれていれば，その前方条件を `nil` に替えた抽象セルを A に追加する，というルールである．このルールを次のように表記することにする．

$(r, \nu, \beta) \rightarrow (r, \mathtt{nil}, \beta)$

抽象セルを置き換えるのではなく，追加することに注意する．これは，同じ条件のルートが他にあるかもしれないからである．

上のルール以外に，ルートからのリンクが消えることによって，セルの後方条件が変わる可能性がある．次のルールが考えられる．

$(\sigma, \nu, \{r, [wgb]^*r, w^*g\}) \rightarrow (\sigma, \nu, \{w^*g\})$
$(\sigma, \nu, \{r, [wgb]^*r\}) \rightarrow (\sigma, \nu, \{\})$
$(\sigma, \nu, \{r, [wgb]^*r, w^*g\}) \rightarrow (\sigma, \nu, \{[wgb]^*r, w^*g\})$
$(\sigma, \nu, \{r, [wgb]^*r\}) \rightarrow (\sigma, \nu, \{[wgb]^*r\})$
$(\sigma, \nu, \{[wgb]^*r, w^*g\}) \rightarrow (\sigma, \nu, \{w^*g\})$
$(\sigma, \nu, \{[wgb]^*r\}) \rightarrow (\sigma, \nu, \{\})$

最初の 2 つは，あるセル c がルートから直接参照されており，そのリンクがルートから c に到達するための唯一の経路であり，リンクが切れることによって，c がルートから直接にも間接にも参照できなくなる場合である．次の 2 つは，ただ 1 つのルートから直接参照されているが，他のルートからたどれる経路が存在する場合，最後の 2 つは，ルートからは直接参照されていないが，ルートからたどるための唯一のリンクが切れる場合である．他の 5 つのミューテータ操作については，抽象遷移の一覧を付録に掲載する．

3.4.2　フェーズとフィルタ処理

スナップショット方式の検証は，コレクタのフェーズごとに行われる．各フェーズは，直前のフェーズから受け渡された抽象状態 A_S に対して，抽象遷移を繰り返し適用し，どの抽象遷移を適用してもそれ以上変化しなくなった時点で終了する．抽象遷移は，抽象セルを追加するだけで，抽象セルを削除することはないので，フェーズ終了時の抽象状態 A_F は，抽象遷移の適用順序に依存しない．$\overline{X} \subseteq A_S$ である任意の処理系状態 X に対して，どのように操作を実行しても，フェーズ終了時の状態 Y は，$\overline{Y} \subseteq A_F$ を満たす．

フェーズ終了時の抽象状態 A_F に対して，フェーズが終了することによって存在しえない抽象セルと，A_F において矛盾するセルを除去し，その結果を次のフェーズに受け渡す．この除去する処理をフィルタ処理という．たとえば，Shade フェーズの始まりでは，ルートが指しているセルはすべて白なので，Shade 開始時の抽象状態 A_S には抽象セル $(r, w, \mathtt{nil})$ が含まれている．Shade フェーズが終わったときには，ルートが指しているセルはすべて灰色に変わっているが，抽象セル $(r, w, \mathtt{nil})$ は削除されることがなく，Shade 終了時の抽象状態 A_F に残っている．そこで，この抽象セルを A_F から取り除く．他の 3 つのフェーズについて，フェーズ終了時に取り除く抽象セルは次のとおりである．

- Mark：終了時には灰色セルが存在しないので，ラベルあるいは前方条件が g である抽象セルと，後方条件に w^*g を含む抽象セル．
- Append：白いセルは，終了時にはすべてフリーになっているので，ラベルあるいは前方条件が w である抽象セル．
- Unmark：黒いセルは，次のごみ集めに備えてすべて白になっているので，ラベルあるいは前方条件が b である抽象セル．

抽象状態 A とその要素である抽象セル $a = (\sigma, \nu, \beta)$ に対して，次のいずれかの条件が成り立つとき，a は A において矛盾するといい，各フェーズ終了時のフィルタ処理によって除去する．

- a の後方条件 β が r を含むときに，前方条件が σ であるルートセルが A 中に存在しない．
- β が r を含まず $[wgb]^*r$ を含むときに，前方条件が σ の抽象セルで，後方条件に $[wgb]^*r$ を含むものが存在しない．
- β が w^*g を含むときに，前方条件が σ の抽象セルで，
  - ラベルが g であるもの
  - ラベルが w で後方条件に w^*g を含むもの

  のいずれも存在しない．
- a の前方条件 ν が `nil` 以外のときに，ラベルが ν の抽象セル $a' = (\nu, \nu', \beta')$ で，次の条件をすべて満たすものが存在しない．
  - a がルートセルのときに，β' が r と $[wgb]^*r$ とを含む．
  - β が $[wgb]^*r$ を含むときに，β' も $[wgb]^*r$ を含む．
  - 次のいずれかの条件が成り立つときに，β' が w^*g を含む．
    - a のラベル σ が g である．
    - σ が w で，β が w^*g を含む．
- a のラベルが w，後方条件が $[wgb]^*r$ を含むが w^*g を含まない場合で，A 中のルートセルから到達可能な抽象セルのラベルが w と g だけであり，A 中のすべてのルートセルの前方条件が `nil` または g である．この条件を RFG（Reachable From Gray）条件とよぶことにする．

最後の RFG 条件について補足する．条件を満たす抽象セル a に抽象化されるセル c を考える．c はルートから到達可能であるが，ルートから c に到達する経路は，先頭のルートを除けば白と灰色のセルだけで構成される．しかもルートの直後は灰色のセルなので，その経路には灰色セルから始まって，白いセルだけを経由して c に到達する経路が含まれる．したがって，c を抽象化した a は，後方条件に w^*g を含むはずであり矛盾する．実際に抽象モデル検査を行うと，この条件を満たす抽象セル $(w, \texttt{nil}, \{[wgb]^*r\})$ が Unmark フェーズで出現する．フィルタ処理によってこれを除去しておかないと，次の Append フェーズでエラーセル $(f, \texttt{nil}, \{[wgb]^*r\})$ が生成される．

表 2　飽和状態の抽象セルの個数
Table 2　The numbers of abstract cells on saturation.

	フェーズ終了時	フィルタ処理後
Shade	44	31
Mark	74	17
Append	18	12
Unmark	27	12

あるフェーズが終了したときの処理系の状態 X に対して，抽象状態 A が $A \supseteq \overline{X}$ を満たすとき，A に上記のフィルタ処理を行った結果 A' も $A' \supseteq \overline{X}$ を満たす．したがって，次のフェーズを A の代わりに A' で開始してもかまわない．

3.4.3　結　　果

以上をまとめると，実施した抽象モデル検査の手順は次のとおりである．抽象状態 $\overline{X_S}$（X_S は処理系の初期状態）から出発し，各フェーズごとに抽象状態が飽和するまで抽象遷移を繰り返し，フィルタ処理を行った後，次のフェーズに進む．これを 4 つのフェーズ Shade, Mark, Append, Unmark の順に行い，Unmark フェーズが終わると，Shade フェーズから同じ処理を繰り返す．このサイクルを，いずれかのフェーズの開始時の抽象状態が，同じフェーズの前回のものと一致するまで続ける．抽象セルの個数は有限なので，いつかはこの飽和状態に達する．

抽象モデル検査を実施した結果，エラーセルが抽象状態に現れることはなかった．したがって，スナップショット方式を備えたこの処理系モデルが安全であることが確認された．初期状態から飽和状態に達するまで繰り返したサイクル数は 3 回であった．飽和状態における抽象セルの個数は**表 2** のとおりである．表の数値は，原論文のものと一致しないが，原論文の著者の 1 人である高橋孝一氏によると，同氏の手元にある最新のデータとは一致しているとのことであった．

4. リターンバリア方式の抽象モデル検査

スナップショット方式の抽象モデル検査をベースにして，リターンバリア方式の抽象モデル検査を行った．

リターンバリア方式では，ルート挿入の途中でもミューテータの操作が実行される．実際の処理系では，関数の呼び出しとリターンによってスタックが伸び縮みする．これを忠実に反映する処理系モデルを与え，その抽象モデルを構築することはかなりの困難をともなうことが予想される．そこで本研究では，リターンバリア方式の最も基本的なアイデアである，「ミューテータが参照するルート領域を，ルート挿入済みの部分に限定する」ことを保証することによって，実時間ごみ集めが安全に行われることを検証する．

4.1 処理系のモデル

スナップショット方式の処理系モデル（3.1 節）を拡張する．リターンバリア方式では，ミューテータがアクセスできるルートは，ルート挿入済みのものに制限される．ルート挿入済みのルートとそれ以外のルートを区別するために，ルートにも “色” をつけることにする．挿入済みのルートには scanned を，それ以外のルートには unscanned をつける．unscanned のルートを 1 つ scanned にし，それが直接参照しているセルを灰色にする操作が，Shade フェーズにおけるコレクタのアトミックな操作である．ごみ集め開始時に，すべてのルートは unscanned であり，すべてのルートが scanned となった時点で，Mark フェーズに移行する．その後，Unmark フェーズが終了するまで scanned のままとし，Unmark フェーズが終了したときに，いっせいに unscanned に戻すことにする．実際の処理系では，Shade フェーズにおいて，スタックのトップからルート挿入を行い，ルート挿入の終わったところまでをポインタで覚えておく．そのポインタ位置までが scanned，それ以降が unscanned である．ルートをいっせいに unscanned に戻すには，ポインタをスタックトップを指すように変更するだけでよい．

コレクタのその他の処理は，スナップショット方式の場合と同じである．ミューテータの操作（表 1 参照）については，操作の対象となるルートが scanned に限定されることと，ライトバリアによる色の変化が Shade フェーズにおいても起きる点が，スナップショット方式の場合と異なる．

4.2 抽象モデル

ルートにも “色” がつくようになったので，抽象モデルにおけるルートセルのラベルは r だけでは足りない．ミューテータの操作は，挿入済みのルートに対して行われるので，スナップショットの抽象モデルからの変更が少なくなるように，r は scanned のルートを表し，unscanned を表すために，u という色を導入する．つまり，抽象セルのラベル σ は，

$$\sigma \in \{r, u, f, w, g, b\}$$

である．ルートはどこからも指されないので，前方条件 ν についてはスナップショットと同じで，

$$\nu \in \{\texttt{nil}, f, w, g, b\}$$

である．後方条件 β については自明でないが，最終的に

$$\beta \subseteq \{r, [wgb]^*r, w^*g, u, [wgb]^*u, w^*u\}$$

とした．

処理系の初期状態 X_S では，すべてのルートが unscanned で，すべてのセルがフリーなので，

$$\overline{X_S} = \{(u, \texttt{nil}, \{\}), (f, \texttt{nil}, \{\})\}$$

となる．エラーセルについては，スナップショットにおけるエラーセルに加えて，ラベルが f で後方条件に $[wgb]^*u$ を含む抽象セルもエラーセルとなる．

4.3 抽象モデル検査とその結果

リターンバリア方式では，Shade フェーズ終了時には，すべてのルートが scanned となっている．したがって，ラベルが u である抽象セルと，後方条件に u，$[wgb]^*u$，w^*u のいずれかを含む抽象セルは存在しえない．これらの抽象セルは，Shade フェーズ終了時のフィルタ処理によって削除する．

スナップショット方式の場合は，Shade フェーズ終了時に，前方条件が w であるルートセルを削除したが，リターンバリア方式では，Shade フェーズ中のミューテータの実行によって，そのような抽象セルが Shade フェーズ終了時に存在しうるので，削除してはいけない．たとえば，ルート `R[1]` が灰色のセル `C[1]` を指し，そのセルが白いセル `C[2]` を指していたとする．ミューテータが load(2, 1) を実行すると，`R[2]` というルートが，白いセル `C[2]` を直接指すことになる．

Mark フェーズと Append フェーズの終了時フィルタ操作は，スナップショットの場合と同じである．最後の Unmark フェーズが終わるときには，すべてのルートを scanned から unscanned にいっせいに戻す必要がある．したがって，ラベルが r である抽象セルと，後方条件に r，$[wgb]^*r$，w^*r のいずれかを含む抽象セルがあれば，r を u に置き換えた抽象セルと置き換える．

リターンバリア方式では，抽象状態 A の要素である抽象セル $a = (\sigma, \nu, \beta)$ は，次の条件

表 3　飽和状態の抽象セルの個数（リターンバリアの場合）
Table 3　The numbers of abstract cells on saturation (case return barrier).

	フェーズ終了時	フィルタ処理後
Shade	245	74
Mark	74	17
Append	18	12
Unmark	48	12

を満たすときにも矛盾するセルであり，フィルタ処理において削除する．

- a のラベルが w，後方条件 β が $[wgb]^*u$ を含むが w^*u を含まない場合で，A 中のラベル u のルートセルから到達可能な抽象セルのラベルが w だけであり，ラベルが u である A 中のすべてのルートセルの前方条件が `nil` または w である．

この条件は，3.4.2 項の RFG フィルタと似ている．この条件を満たす白いセル a に対しては，unscanned のルートセルから到達できる経路が存在する．その経路中のセルは，先頭のルートセル以外はすべて白である．したがって，a は後方条件として w^*u を含むはずであり矛盾する．実際に抽象モデル検査を行うと，この条件を満たすセル $(w, \texttt{nil}, \{[wgb]^*u\})$ が Unmark フェーズの直後に出現する．フィルタ処理によってこれを除去しないと，ルートから到達可能にもかかわらず，白いセルなので次の Append フェーズで回収されてしまい，エラーセル $(f, \texttt{nil}, \{[wgb]^*u\})$ を生成する．

抽象モデル検査の実施手順は，スナップショットの場合と同じである．抽象モデル検査を実施した結果，リターンバリア方式においても，エラーセルが抽象状態に現れることはなかった．したがって，リターンバリア方式を備えたこの処理系モデルが安全であることが確認された．初期状態から飽和状態に達するまで繰り返したサイクル数は 3 回であった．飽和状態における抽象セルの個数は**表 3** のとおりである．表 2 と比較すると，リターンバリア方式では Shade フェーズと Unmark フェーズに抽象セルの数が増大しているのが分かる．これはリターンバリア方式の場合，リターンバリア方式特有の後方条件（u，$[wgb]^*u$，w^*u）を含む抽象セルが Shade フェーズと Unmark フェーズで現れることに起因している．また，Shade フェーズでは unscanned のルートを scanned にする遷移が加わるので，scanned ルートと unscanned ルートが混在して抽象セルの数が増える．Unmark フェーズではスキャン済みのルートを未スキャンのルートにする遷移が加わるので，同様に抽象セルの数が増える．逆に Mark フェーズと Append フェーズでは未スキャンのルートは存在しないので，リターンバリアによる影響は存在せず，抽象セルの数も同じになっている．

5. 定理証明系による抽象遷移の妥当性の証明

この章では，抽象遷移の妥当性の証明の手法を述べる．本研究では証明に定理証明系 Isabelle/HOL [7] を使用した．Isabelle/HOL は，ユーザがシステムと対話しながら証明を進めていく証明支援系であり，拡張子が．thy のファイルに記述された定理を，Emacs 上で対話的に証明することができる．

5.1 諸　定　義

証明のために必要な定義を与える．まず，抽象セルのラベルとコレクタのフェーズを，データ型 `color` と `step` で表す．以下はスナップショット方式用の定義である．

```
datatype color = Free | White | Gray | Black | Root
datatype step = Shade | Mark | Append | Unmark
```

リターンバリア方式の場合は，ルートを表す `Root` の代わりに，ルート挿入済みのルートを表す `Scanned` と未挿入のルートを表す `Unscanned` を定義に含める．

後方条件についての定義を与える．以下に定義する `direct_reachable` はルートから直接参照されていることを表す関数である．

```
constdefs direct_reachable
  :: "(nat⇒nat option)⇒(nat⇒color×nat option)⇒nat⇒bool"
  "direct_reachable r h k ≡ ∃i. (r i = Some k)"
```

`constdefs` は関数の宣言と定義を与えるもので，上のようにそのすぐ後に関数名を記述する．この関数はルート領域 `r`，ヒープ領域 `h`，自然数 `k` を引数とし，`k` をインデックスとするセルが，`r` 内のあるルートから指されていれば `True` を返し，そうでなければ `False` を返す．以下では，関数定義の引数として出てくる `r` と `h` は，ルート領域とヒープ領域を意味することにする．`'a option` は，「`Some 'a` もしくは `None`」という型である．`Some` は，`'a option` に `None` 以外の値が入っていることを表すために用意した型である．ポインタの型は `nat option` とし，空ポインタは `None` で，その他のポインタは `Some nat` 型の値で表す．ルート領域 `r` はルートのインデックス `nat` を受け取り，そのルートの値を返す．`h` はセルのインデックス `nat` を受け取り，そのセルのラベルと値の組を返す．ただし上の定義では `h` は使用していない．リターンバリア方式の場合は，ルートが scanned または unscanned の色を持つので `r` は `h` と同じように，指定されたルートのラベル（`Scanned` または `Unscanned`）と値の組を返す．

次にルートからの到達可能性を表す `reachable` と，灰色セルから白いセルたどって到達

可能であることを表す g_reachable を定義する．

```
inductive reachable
   :: "(nat⇒nat option)⇒(nat⇒color×nat option)⇒nat⇒bool"
  for r h where
  rzero[simp]:
    "direct_reachable r h k ⟹ reachable r h k"
  | rstep[simp]:
    "reachable r h k ∧ issome(snd(h k))
     ⟹ reachable r h (the_i (snd(h k)))"

inductive g_reachable
  :: "(nat⇒color×nat option)⇒nat⇒bool"
  for h where
  gzero[simp]:
    "fst(h k)=Gray ∧ issome(snd(h k))
     ⟹ g_reachable h (the_i (snd(h k)))"
  | gstep[simp]:
    "g_reachable h k ∧ fst(h k)=White ∧ issome(snd(h k))
     ⟹ g_reachable h (the_i (snd(h k)))"
```

inductive は関数が帰納的に定義されていることを表す．for r h は，r と h が固定された帰納的定義であること，すなわちこの帰納的定義が k に関するものであることを意味している．[simp] は後で証明を行う際に，simp というコマンドでこの定義も利用できるようにするためで，証明の手間を省くための記述である．これらの定義はリターンバリア方式でもほぼ同じである．

reachable は r，h，k を引数として，インデックスが k であるセルがルートから到達できれば True，そうでないなら False を返す．この定義は rzero と rstep という 2 つに分けられており，前者は「直接到達可能なセルは到達可能である」ということ，後者は「到達可能なセルから参照されているセルは到達可能である」ということを述べている．前者は，direct_reachable r h k が True なら reachable r h k も True であるということを表している．後者は，reachable r h k が True であり，かつ issome(snd(h k)) ならば，インデックス k のセルが指しているセルも到達可能であるということを表している．issome は'a option 型の値を引数とし，引数が Some 'a ならば True を返し，None ならば False を返す関数である．snd(h k) はインデックスが k のセルの値なので，前提条件で空ポインタの場合を除外している．the_i は Some 'a 型の値を引数とし，引数から Some を取り除いた値を返す関数である．

g_reachable の定義は reachable と似ており，gzero と gstep の 2 つに分けられている．前者は「灰色のセルから参照されているセルは g_reachable である」ということ，後者は「g_reachable かつ白いセルから参照されているセルは g_reachable である」ことを述べている．fst(h k) はインデックスが k のセルのラベルを表している．

これらの定義を用いて，セルの抽象化と，抽象状態を定義する．

```
constdefs abstract_cell
  :: "(nat⇒nat option)⇒(nat⇒color×nat option)⇒nat⇒bool
      ⇒color⇒(color option)⇒bool⇒bool⇒bool⇒bool"
  "abstract_cell r h k sel_c c f d_r reach g_r
   ≡ (c = (if(sel_c) then fst(h k) else Root))
     ∧ (f = (if(¬sel_c)
             then (
               if(issome(r k))
               then (Some(fst(h (the_i(r k)))))
               else None
             ) else (
               if(issome(snd(h k)))
               then (Some(fst(h (the_i(snd(h k))))))
               else None)))
     ∧ (d_r = (if(sel_c)
               then (direct_reachable r h k)
               else False))
     ∧ (reach = (if(sel_c)
                 then (reachable r h k)
                 else False))
     ∧ (g_r = (if(sel_c)
               then (g_reachable h k)
```

```
                else False))"
```

この abstract_cell は，インデックスが h であるセル（sel_c が True のとき）あるいはルート（sel_c が False のとき）がどのような抽象セルに抽象化されるかを定義している．c と f は抽象セルのラベルと前方条件である．後方条件は，3 つの変数 d_r，reach，g_r で表されており，それぞれが後方条件に r，$[wgb]^*r$，w^*g が含まれるかどうかを意味する．

リターンバリア方式の場合はルートにも色があるので，c の定義を次のもので置き換える．

```
c = (if(sel_c) then fst(h k) else fst(r k))
```

さらに，後方条件に u，$[wgb]^*u$，w^*u を含む可能性があるので，後方条件にそれらが含まれるかどうかを意味する変数を 3 つ追加する必要がある．

最後に，処理系の状態と抽象状態との関係を定義する．

```
constdefs abstract_relation
  :: "step⇒(nat⇒nat option)⇒(nat⇒color×nat option)
      ⇒step⇒(color⇒(color option)⇒bool⇒bool⇒bool)⇒bool"
  "abstract_relation s r h as ah
   ≡ (s = as)
      ∧ (∀sel_c c f d_r reach g_r.
           (∃ k. abstract_cell r h k sel_c c f d_r reach g_r)
           ⟶ ah c f d_r reach g_r)"
```

処理系におけるコレクタのフェーズ s，ルート領域 r，ヒープ領域 h によって定義される処理系の状態が，フェーズ as と抽象セルの集合 ah で表される抽象モデルの状態に対応するための条件を与えている．定義自体は簡単で，両方のフェーズが一致し，ah 内のすべての抽象セルに対して，その抽象セルに抽象化できるセルあるいはルートが処理系内に存在するかを判断するだけである．

5.2　初期状態とエラー状態の抽象化の妥当性の証明

これらの定義を用いて，初期状態とエラー状態を抽象化したものが，抽象モデル検査で用いる抽象状態の初期状態とエラー状態（エラーセルを含む抽象状態）であることを証明する．

以下にスナップショット方式における処理系モデルの初期状態の定義と，それの抽象状態の定義を述べる．

```
constdefs initial
    :: "step⇒(nat⇒nat option)⇒(nat⇒color×nat option)⇒bool"
    "initial s r h
     ≡ (s = Shade)
        ∧ (∀i. r i = None)
        ∧ (∀k. fst(h k) = Free)
        ∧ (∀k. snd(h k) = None)"

constdefs abs_initial
    :: "step⇒(color⇒color option⇒bool⇒bool⇒bool)⇒bool"
    "abs_initial as ah
     ≡ (as = Shade)
        ∧ (∀c f d_r r g_r.
             ah c f d_r r g_r
             = (c=Free ∧ f=None ∧ d_r=False∧ r=False ∧ g_r=False
                ∨ c=Root ∧ f=None ∧ d_r=False∧ r=False ∧ g_r=False))"
```

initial は，処理系の初期状態 X_S における引数の s，r，h の内容を与えており，abs_initial は，X_S を抽象化した抽象状態 $\overline{X_S}$ における抽象セルの構成を与えている．後者は 3.2 節で与えたものと同じである．これらが正しく抽象化されていることを次の定理として証明する．

```
theorem initial_safe:
    "∀s r h as ah.
       (initial s r h ∧ abs_initial as ah)
       ⟶ abstract_relation s r h as ah"
```

ここで，⟶ は前述の ⇒ とは違い，論理記号の「ならば」を表している．すべての抽象セルの前方条件が nil で，後方条件が空集合であることを示せば，この定理は容易に証明できる．

次にスナップショット方式の処理系における安全な状態（エラー状態でない状態）の定義と，それの抽象状態を定義する．

```
constdefs safe
    :: "step⇒(nat⇒nat option)⇒(nat⇒color×nat option)⇒bool"
    "safe s r h
     ≡ ∀k. (fst(h k)≠Free ∨ ¬reachable r h k)"
```

```
constdefs abs_safe
    :: "step⇒(color⇒color option⇒bool⇒bool⇒bool⇒bool)⇒bool"
    "abs_safe as ah
     ≡ ∀f d_r g_r. ¬(ah Free f d_r True g_r)"
```

safe は，引数の s，r，h によって定義される状態において，どのフリーセルもルートから到達可能でない，つまりその状態がエラー状態ではなければ True を返す．abs_safe は引数の as，ah の定義する抽象状態が，エラーセルを含まなければ True を返す．エラーセルとは，この定義にあるように，ラベルがフリーで後方条件に $[wgb]^*r$ を含む抽象セルである．

これらが正しく抽象化されていることを次の定理で証明する．

```
theorem safety:
  "∀ s r h as ah.
     (abs_safe as ah ∧ abstract_relation s r h as ah)
     ⟶ safe s r h"
```

この定理は，エラー状態とエラーセルの定義をあてはめることにより証明することができる．

リターンバリアでもこれらはほぼ同じであるが，初期状態でルートのラベルがすべて unscanned であることに注意する．

5.3　抽象遷移の妥当性の証明

抽象遷移の妥当性の証明方法について述べる．スナップショット方式の rclear を例にあげ，3.4.1 項であげた rclear の抽象遷移が妥当であることを証明する．

まず処理系における rclear 操作を定義する．

```
constdefs rclear
  :: "step⇒(nat⇒nat option)⇒(nat⇒color×nat option)
      ⇒step⇒(nat⇒nat option)⇒(nat⇒color×nat option)
      ⇒nat⇒bool"
  "rclear s r h s' r' h' i
   ≡ (s'=s)
      ∧ (∀m. r' m = (if(m=i) then None else r m))
      ∧ (h'=h)"
```

s r h は遷移前のフェーズ，ルート領域，ヒープ領域を表しており，s' r' h' は遷移後のフェーズ，ルート領域，ヒープ領域を表している．i は操作の対象となるルートのインデックスである．rclear によってフェーズは変化しないので s'=s である．また，ヒープ領域は変化しないので，h'=h である．rclear によって，インデックス i のルートに空ポインタが代入されるので，r' i は None に変化し，他のルートの値は遷移前と変わらない操作にもよるが，処理系の操作は，このように抽象遷移が簡単に書けることが多い．

リターンバリア方式での rclear の定義は以下のようになる．

```
constdefs rclear
  :: "step⇒(nat⇒color×nat option)⇒(nat⇒color×nat option)
      ⇒step⇒(nat⇒color×nat option)⇒(nat⇒color×nat option)
      ⇒nat⇒bool"
  "rclear s r h s' r' h' i
   ≡ (s'=s)
      ∧ (∀m. fst(r m) = fst(r' m))
      ∧ (∀m. snd(r' m)
              = (if(m=i ∧ fst(r m)=Scanned)
                 then None else snd(r m)))
      ∧ (h'=h)
      ∧ (∀m. fst(r m)=Scanned ∨ fst(r m)=Unscanned)"
```

リターンバリア方式の場合は，rclear の操作対象となるのは，scanned ルートだけである．これを定義に反映させるために，ここでは，unscanned ルートに適用された場合は処理系に影響を与えないものとしている．

次にスナップショット方式の rclear の抽象遷移を定義する．

```
constdefs abs_rclear
  :: "step⇒(color⇒color option⇒bool⇒bool⇒bool⇒bool)
      ⇒step⇒(color⇒color option⇒bool⇒bool⇒bool⇒bool)⇒bool"
  "abs_rclear as ah as' ah'
   ≡ (as'=as)
      ∧ (∀c f d_r r g_r f2.
           ah' c f d_r r g_r
           = (ah c f d_r r g_r
              ∨ ah c f2 d_r r g_r ∧ c=Root ∧ f=None
              ∨ ah c f True True g_r ∧ d_r=False ∧ r=False
```

```
        ∨ ah c f True True g_r ∧ d_r=False ∧ r=True
        ∨ ah c f False True g_r ∧ d_r=False ∧ r=False))"
```

これは，3.4.1 項にあげた抽象遷移を論理式の形に書き換えただけである．等式の右辺は 5 つの式の論理和となっている．最初の式 `ah c f d_r r g_r` は，遷移前に存在した抽象セルは，遷移後にも残っていることを表す．2 つ目の式は，ルートセルで値が空ポインタとなるものができることを表している．残りの 3 つの式は，3.4.1 項であげたヒープセルへの 6 つの抽象遷移を 3 つにまとめたものである．

このように rclear は比較的抽象遷移が少ないが，ライトバリアがある cclear や store の抽象遷移は数十パターンとなる．

リターンバリア方式の場合もほぼ同じであるが，`c=Root` が `c=Scanned` になる点に注意する．遷移によってはスナップショット方式とリターンバリア方式で抽象遷移が大きく異なるものもある．それらについては 6.1 節で述べる．

これらを用いて rclear の抽象遷移の妥当性を証明する．定理は以下のように記述する．

```
theorem rclear_safe:
  "∀s r h s' r' h' i as ah as' ah'.
       rclear s r h s' r' h' i
       ∧ abs_rclear as ah as' ah'
       ∧ abstract_relation s r h as ah
       ⟶ abstract_relation s' r' h' as' ah'"
```

証明手法については何通りかあるが，セルの条件によって場合分けをして，それぞれのケースでの遷移が妥当であることを示す方法が一般的と思われる．この定理に `abstract_relation` と `abstract_cell` の定義を適用して整理すると，次のゴールを得る．

```
goal (1 subgoal):
 1. ⋀r h r' i ah ah'
       d_r sel_c c reach g_r f.
        [∀m. (m = i ⟶ r' i = None)
         ∧ (m ≠ i ⟶ r' m = r m);
         ∀c f d_r r g_r f2.
            ah' c f d_r r g_r =
            (ah c f d_r r g_r
             ∨ ah c f2 d_r r g_r ∧ c = Root ∧ f = None
             ∨ ah c f True True g_r ∧ ¬ d_r ∧ ¬ r
             ∨ ah c f True True g_r ∧ ¬ d_r ∧ r
             ∨ ah c f False True g_r ∧ ¬ d_r ∧ ¬ r);
         ∀d_r sel_c c reach g_r f.
           (∃k. abstract_cell r h k sel_c c f d_r reach g_r)
           ⟶ ah c f d_r reach g_r]
        ⟹ (∃k. abstract_cell r' h k sel_c c f d_r reach g_r)
             ⟶ ah' c f d_r reach g_r
```

⋀ は次のピリオドまでの記号がメタ変数であることを表している．これらのメタ変数のうち，`d_r`，`sel_c`，`c`，`reach`，`g_r`，`f` が遷移後のセルの条件となる．これを場合分けして（たとえば `sel_c` が `True` の場合と `False` の場合で分けて抽象セルがルートセルの場合とそうでない場合に分けるなど），状態遷移が可能な遷移は，抽象遷移でもすべて可能であることを証明する．帰納的な証明も多く出てくるが（rclear で w^*g が変化しないのはなぜか？など），`induct_tac` などの帰納法証明のために用意されたルールを使えば証明は可能である．

6. 結果と考察

6.1 妥当性の証明

スナップショット方式とリターンバリア方式のそれぞれについて以下の妥当性を証明した（ただし Rootunscan はリターンバリア方式の場合のみ）．

- 抽象化の妥当性
  - 初期状態
  - エラー状態
- 抽象遷移の妥当性
  - ミューテータの操作
    rclear，cclear，allocate，move，load，store
  - コレクタのアトミック操作
    Shade，Mark，Append，Unmark，Rootunscan

Rootunscan はリターンバリア方式の Unmark フェーズのうち，ルートのセルを Unscanned に戻す操作を分けたものである．証明をしやすいよう便宜上分けただけであり，特に新しい抽象遷移などは追加していない．

5.2 節で示したように，抽象化の妥当性の証明は比較的容易であり，リターンバリア方式

の場合でも 200 行程度以下で証明できたのに対し，抽象遷移の妥当性は 1,000 行を超えるものも多く，最も長いもの（リターンバリア方式の store）は 7,782 行であった．抽象遷移の妥当性の証明は，すべての抽象セルをラベルや後方条件などで場合分けをしてそれぞれのケースで妥当性を満たすことを証明するため，cclear や store など抽象遷移の数が多いものは証明が長くなり，反対に allocate や Append など抽象遷移の数が少ないものは証明が短くなる傾向があった．また，抽象遷移の中にはリターンバリア方式でもスナップショット方式でも抽象遷移がほとんど変わらないものもある．たとえば Mark や Append フェーズにおけるコレクタのアトミック操作は，2 つの方式で抽象遷移は変わらない．これはこの 2 つの遷移が次のことを満たしているからである．

- セルのポインタを変えない．
- ルートのラベルを変更しない．
- ルートのラベルによって挙動を変えない．

このためこの 2 つの遷移はリターンバリア方式の場合でも，スナップショット方式の証明を利用して比較的容易に証明することができる．ただし，リターンバリア方式に固有の後方条件とルートのラベルを，これらの遷移が変更しないことを証明しなければいけない．ミューテータの遷移に関しても，これらのことを証明できる場合は，抽象遷移の証明は比較的容易であった．

6.2 落 と し 穴

今回の検証作業では，まず抽象遷移を推定し，抽象モデル検査をした後に，それらの妥当性の証明にとりかかった．妥当性が証明できない場合には抽象遷移を再び検討し，抽象モデル検査を行ってエラー状態が出ないよう抽象遷移を練り上げ，再び妥当性の証明を行った．これらの繰返しによりエラー状態が生成されず，かつ妥当である抽象遷移を見つけることができて安全性を証明することができた．

大きくつまずいたポイントは以下の 3 つである．

- 後方条件 w^*u
- リンクがループする場合の抽象遷移
- リターンバリア方式特有の，後方条件を変化させるミューテータの遷移

この後方条件と抽象遷移が抜けていたため抽象モデル検査と妥当性の証明に失敗しうまくいかなかった．以下でこれらについて詳しく述べる．

6.2.1 後方条件 w^*u

当初はリターンバリア方式の後方条件の集合は $\{r, [wgb]^*r, w^*g, u, [wgb]^*u\}$ であった．これはスナップショット方式に u と $[wgb]^*u$ を加えたものである．一般に，ごみ集めの検証では，ルートからの到達可能性が重要になる．スナップショット方式の場合は Shade フェーズがあるので，到達可能性に加えて直接到達可能であるかどうかも考慮する必要がある．リターンバリア方式でもこれらの条件が必要であると判断し，u と $[wgb]^*u$ を加えて後方条件とした．しかし実際には抽象モデル検査はうまくいかず，どうやってもエラー状態が出てしまったため，再び後方条件を見直し，w^*g に相当する条件を付け加えることが必要であることに気づいた．w^*g は，セルがこれからマークされるであろうことを意味しており，回収されないセルにつく後方条件である．よってリターンバリア方式でも回収されないセルの条件を考え，w^*u を追加することにした．unscanned のルートから白いセルのみをたどって参照できるセルは，ルート挿入により必ず w^*g が後方条件に含まれることになる．この後方条件によって，抽象モデル検査に成功した．

図 7　Unmark フェーズ
Fig. 7　Unmark phase.

図 8　ループするセルの Unmark フェーズ
Fig. 8　A looping cell in unmark phase.

6.2.2 リンクがループする場合の遷移

例として Unmark フェーズのコレクタの遷移をあげる．Unmark では**図 7** のように黒いセルを白に変化させる．したがって抽象遷移として，前方条件が黒であるセルを白にするとか，ラベルが黒であるセルを白にするなどが考えられる．しかし前方条件とラベルが同時に白になるような抽象遷移を，当初は想定していなかった．これは，ミューテータやコレクタは 1 度に 1 つのセルしか操作しないため，1 度の遷移で変化するのはラベルか前方条件どちらか一方だろうと考えたためである．しかし実際に妥当性の証明をするうちに，どうしても証明できないパターンがあり，抽象遷移が足りないことに気づいた．それは**図 8** のように，リンクがループしている場合である．

このような場合は，このセルのラベルが変わることにより前方条件も同時に変わる．自分自身を直接指している場合でなくても，間接的に自分を指す場合なども考慮しないと，見逃

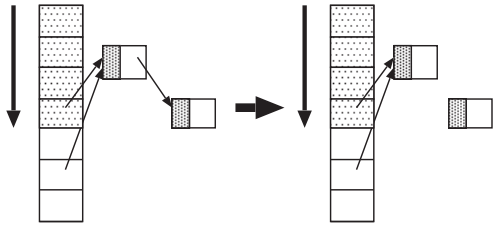

図 9　リターンバリア特有の後方条件を変化させる cclear
Fig. 9　Cnullwrite chaging the rear condition peculiar to return barrier.

してしまう抽象遷移があった．これらを新たに加えて再び抽象モデル検査をすると，エラー状態は出なかった．これらの遷移は入れなくても，飽和状態の抽象セルの集合は変化しなかった．これは，抽象セルを 2 回の遷移で変化させるところを，新しい遷移では 1 回で変化させているだけの違いであったためである．

6.2.3　リターンバリア特有の後方条件を変化させるミューテータの遷移

当初我々はリターンバリアを導入しても，ミューテータの操作に対応する抽象遷移への影響はないと考えていた．リターンバリア方式のミューテータは scanned ルートから参照されているセルしか参照せず，このルートをスナップショット方式でのルートと見なせばスナップショット方式とまったく同じ抽象遷移でよいと考えたためである．しかし妥当性を証明するうちに，先ほど同様どうしても証明できないパターンが，いくつかのミューテータの操作でみつかった．**図 9** は cclear の例である．

このような場合はセルは scanned ルートから到達できなくなるだけでなく，unscanned ルートからも到達できなくなってしまう．よってリターンバリア方式特有の後方条件を変化させる抽象遷移が必要であることが分かった．これらの存在によって一気に組合せの数が増え，非常に証明が大変になった．

7. 今後の課題

本研究では簡略化した処理系モデルに対して，リターンバリア型実時間ごみ集めの安全性の証明に成功した．課題として考えられるのが，簡略化していない処理系モデルでの検証である．

まず，スタックの伸縮を反映する処理系モデルについては，ある程度の目処が立っており，別の機会に報告したい．

ヒープセルが 2 つ以上のポインタを持つ処理系モデルについては，次のように考えている．簡略化したごみ集めを実装したシステムのモデルを G_{simp} とし，通常のシステムのモデルを G_{norm} とする．G_{simp} をリンク構造に変換したものを X_{simp}，リンク構造の状態遷移を t_{simp} とし，X_{simp} を抽象化したものを $\overline{X_{\mathrm{simp}}}$，抽象遷移を t_{Asimp} とする．G_{norm} についても同様に X_{norm}，t_{norm}，$\overline{X_{\mathrm{norm}}}$，$t_{\mathrm{Anorm}}$ とする．

先行研究でのスナップショット方式を実装したシステムのモデルが簡略化されていたものであったため，我々はこれに準拠して検証ができるように簡略化したモデルで証明をした．このモデルを通常のモデルに戻して証明するとすると，妥当性の証明の抽象化の定義なども変えなければならず，新たに抽象遷移の妥当性の証明をする必要がある．

抽象化には 2 通りが考えられる．1 つ目はモデルをそのまま抽象化し，抽象セルの前方条件を 2 つ（もしくはそれ以上）にする方法である．この場合はもちろん抽象遷移をすべて見直す必要がある．そのため $\overline{\mathrm{X_{simp}}} \neq \overline{X_{\mathrm{norm}}}$ かつ $t_{\mathrm{Asimp}} \neq t_{\mathrm{Anorm}}$ なので，抽象遷移の妥当性の証明と抽象モデル検査の両方を実施しなければならない．

2 つ目は 2 つのポインタを持つセルを，別々の前方条件を持つ 2 つ（もしくはそれ以上）の抽象セルで表す方法である．この場合は，今回用いた抽象遷移をほぼすべて使える可能性があり，$t_{\mathrm{Asimp}} = t_{\mathrm{Anorm}}$ となり，抽象モデル検査を省略できる可能性がある．ただしこの場合でも，$X_{\mathrm{simp}} \neq X_{\mathrm{norm}}$ かつ $t_{\mathrm{simp}} \neq t_{\mathrm{norm}}$ であり，$t_{\mathrm{Asimp}} \neq t_{\mathrm{Anorm}}$ となる可能性があるので，抽象遷移を吟味して，改めて妥当性を証明しなければならない．

また，簡略化した処理系モデルは本質的にはポインタを 2 つ以上持つシステムのモデルと変わらないという仮説を立てて，これを証明してもよい．すなわち $X_{\mathrm{simp}} = X_{\mathrm{norm}}$ かつ $t_{\mathrm{simp}} = t_{\mathrm{norm}}$ の証明である．もしこれを証明することができれば，抽象モデル検査は本研究の結果をそのまま使えるので，簡略化していないリターンバリアの証明もできたことになる．この方法だと理論的な証明のみをすればよく，抽象モデル検査や，定理証明系による証明などはしなくてもよい．ただしこれはあくまで仮説なので，この仮説が間違っていた場合は前出の方法を使って抽象モデル検査をする．

8. おわりに

我々はリターンバリア型実時間ごみ集めの安全性の検証を行い，簡略化したリターンバリア型実時間ごみ集めが安全であることを，抽象モデル検査によって証明し，Isabelle/HOL により，この抽象化が妥当であることを示すことができた．

この研究結果は，リターンバリア型実時間ごみ集めの安全性が，リターンバリアの実用化へ

の大きな足がかりとなるものである．また抽象遷移の妥当性の証明によって，Isabelle/HOL による形式的証明のアプローチを示すこともできた．これによって他の抽象モデル検査の抽象遷移の妥当性の証明はもちろん，他の分野への Isabelle/HOL の導入としても応用できる．

謝辞　本研究を行うにあたり，多大なご助言，ご指導をいただいた産業技術総合研究所の高橋孝一氏に深く感謝いたします．また本論文を改訂するにあたり，多数の改善点を指摘いただき，忍耐強く改善作業を支援していただいた査読者ならびに編集委員の方々に心より感謝いたします．本研究の一部は，情報爆発に対応する高度にスケーラブルなソフトウェア構成基盤（18049015）（科学研究費特定領域研究「情報爆発時代に向けた新しい IT 基盤技術の研究」）の補助を得て行った．

参 考 文 献

1) Aagaard, M., Leeser, M.E. and Windley, P.J.: Toward a Super Duper Hardware Tactic, *HUG '93: Proc. 6th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, Joyce, J.J. and Seger, C.-J.H. (Eds.), *Lecture Notes In Computer Science*, Vol.780, pp.399–412, Springer-Verlag (1994).
2) Clarke, E.M. and Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic, *Lecture Notes in Computer Science*, Vol.131, pp.52–71 (1982).
3) Clarke, E.M., Grumberg, O. and Long, D.E.: Model Checking and Abstraction, *ACM Trans. Progr. Lang. Syst.*, Vol.16, No.5, pp.1512–1542 (1994).
4) Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S. and Steffens, E.F.M.: On-the-Fly Garbage Collection: An Exercise in Cooperation, *Comm. ACM*, Vol.21, No.11, pp.966–975 (1978).
5) Long, D.E.: Model Checking, Abstraction and Compositional Reasoning, Ph.D. Thesis, Carnegie Mellon University (1993).
6) McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, *Comm. ACM*, Vol.3, No.4, pp.184–195 (1960).
7) Nipkow, T., Paulson, L.C. and Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *Lecture Notes in Computer Science*, Vol.2283, Springer-Verlag (2002).
8) 高橋孝一：Abstraction and Search in Verification by State Exploration, 博士論文, The University of Tokyo (2002).
9) Takahashi, K. and Hagiya, M.: Abstraction of Link Structures by Regular Expressions and Abstract Model Checking of Concurrent Garbage Collection, *The 1st Asian Workshop on Programming Languages and Systems, National University of Singapore*, pp.1–8 (2000).
10) 高橋孝一，萩谷昌己：正則表現を用いた並列ごみ集めの抽象モデル検査，情報処理学会論文誌：プログラミング，Vol.42, No.SIG2 (PRO9), pp.61–70 (2001).
11) Yuasa, T.: Real-time garbage collection on general-purpose machines, *Journal of Systems and Software*, Vol.11, No.3, pp.181–198 (1990).
12) 湯淺太一：実時間ごみ集め，情報処理，Vol.35, No.11, pp.1006–1013 (1994).
13) 湯淺太一，中川雄一郎，小宮常康，八杉昌宏：リターン・バリア，情報処理学会論文誌：プログラミング，Vol.41, No.SIG9 (PRO8), pp.87–99 (2000).

付　　録

A.1　抽象遷移一覧

スナップショット方式の抽象遷移一覧を以下に示す．ミューテータの 6 つの基本操作 allocate，rclear，move，load，cclear，store のそれぞれに対して，抽象モデル検査で使った抽象遷移をリストアップする．ただし，store の抽象遷移は cclear の抽象遷移をすべて含むので，差分だけを記載する．また，フェーズによって抽象遷移が異なる操作（allocate，cclear，store）については，最も特徴的な Mark フェーズ抽象遷移を掲載する．

リターンバリア方式では，抽象遷移の個数はこの約 2 倍になる．

個々の抽象遷移は，その抽象遷移を起こすために必要な抽象セルのパターンと，生成される抽象セルを，それぞれ矢印の左側と右側に分けて記す．σ，ν，β は，ラベルの色，前方条件，後方条件を表すパターン変数で，必要に応じて添字をつける．前方条件に σ が現れる場合は，前方条件が `nil` 以外であることを意味する．色について制約がある場合は，各抽象遷移の末尾に記載する．スペースを節約するために，後方条件の $[wgb]^*r$ と w^*g は，それぞれ $*r$ と $*g$ で表記する．後方条件中の $?r$，$?*r$，$?*g$ は，それぞれ r，$[wgb]^*r$，w^*g が含まれるかどうかを表すパターン変数である．たとえば，rclear の 2 つめの抽象遷移は，「ラベルが σ，前方条件が ν で，後方条件に $[wgb]^*r$ を含む抽象セル a が存在するとき，ラベルが σ，前方条件が ν で，後方条件に r と $[wgb]^*r$ を含まない抽象セル a' を追加する．a が w^*g を含む場合に限って a' も w^*g を含む」と読む．

• allocate

$(r, \nu_1, \beta) \rightarrow (r, b, \beta)$

$(f, \nu_1, \{?r_1, ?*r_1, ?*g\}) \rightarrow (b, \mathtt{nil}, \{r, *r, ?*g\})$

$(\sigma, f, \beta) \rightarrow (\sigma, b, \beta)$

• rclear

$(r, \nu_1, \beta) \to (r, \mathtt{nil}, \beta)$

$(\sigma, \nu, \{?r_1, *r, ?*g\}) \to (\sigma, \nu, \{?*g\})$

$(\sigma, \nu, \{r, *r, ?*g\}) \to (\sigma, \nu, \{*r, ?*g\})$

• move

$(r, \nu_1, \beta), (r, \nu, \beta_1) \to (r, \nu, \beta)$

$(\sigma, \nu, \{?r_1, *r, ?*g\}) \to (\sigma, \nu, \{?*g\})$

$(\sigma, \nu, \{r, *r, ?*g\}) \to (\sigma, \nu, \{*r, ?*g\})$

• load

$(\sigma, \nu, \{?r_1, *r, ?*g\}) \to (\sigma, \nu, \{?*g\})$

$(\sigma, \nu, \{r, *r, ?*g\}) \to (\sigma, \nu, \{*r, ?*g\})$

$(\sigma_1, \sigma, \{r, *r, ?*g_1\}), (\sigma, \nu, \{*r, *g\}) \to (\sigma, \nu, \{r, *r, *g\})$

$(\sigma_1, \sigma, \{r, *r, ?*g_1\}), (\sigma, \nu, \{*r\}) \to (\sigma, \nu, \{r, *r\})$ $(\sigma_1 \neq w, g)$

$(r, \nu_1, \beta), (\sigma_1, \sigma_2, \{r, *r, ?*g_1\}), (\sigma_2, \nu_2, \{?r_1, *r, *g\}) \to (r, \sigma_2, \beta)$

$(r, \nu_1, \beta), (\sigma_1, \sigma_2, \{r, *r, ?*g_1\}), (\sigma_2, \nu_2, \{?r_1, *r\}) \to (r, \sigma_2, \beta)$ $(\sigma_1 \neq w, g)$

$(r, \nu_1, \beta), (w, \sigma_2, \{r, *r\}), (\sigma_2, \nu_2, \{?r_1, *r\}) \to (r, \sigma_2, \beta)$

$(w, \sigma, \{r, *r\}), (\sigma, \nu, \{*r\}) \to (\sigma, \nu, \{r, *r\})$

• cclear

$(\sigma, \nu_1, \{r, *r, ?*g\}) \to (\sigma, \mathtt{nil}, \{r, *r, ?*g\})$

$(\sigma, w, \beta), (w, \nu_1, \{?r_1, *r, ?*g_1\}) \to (\sigma, g, \beta)$

$(\sigma, w, \{r, *r\}), (\sigma_1, w, \{r, *r, ?*g_1\}), (w, \nu_1, \{?r_1, *r\}) \to (\sigma, \mathtt{nil}, \{r, *r, *g\})$ $(\sigma_1 \neq g)$

$(\sigma, w, \{r, *r\}), (w, w, \{r, *r\}), (w, \nu_1, \{*r\}) \to (\sigma, \mathtt{nil}, \{r, *g\})$

$(\sigma, w, \{r, *r\}), (w, w, \{r, *r\}), (w, \nu_1, \{?r_1, *r\}) \to (\sigma, \mathtt{nil}, \{r, *r, *g\})$

$(\sigma_1, \nu_1, \{r, *r, ?*g_1\}), (\sigma, \nu, \{*r, ?*g\}) \to (\sigma, \nu, \{?*g\})$

$(\sigma_1, \sigma, \{r, *r, ?*g_1\}), (\sigma, \nu, \{*r, *g\}) \to (\sigma, \nu, \{*g\})$ $(\sigma \neq w), (\sigma_1 \neq w, g)$

$(\sigma_1, \sigma, \{r, *r, ?*g_1\}), (\sigma, \nu, \{*r\}) \to (\sigma, \nu, \{\})$ $(\sigma \neq w), (\sigma_1 \neq w, g)$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (\sigma, w, \{*r, ?*g\}) \to (\sigma, g, \{?*g\})$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (\sigma, w, \{?r, *r\}), (w, \nu_1, \{?r_1, *r\}) \to (\sigma, g, \{?r, *r, *g\})$ $(\sigma_1 \neq g)$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (w, \nu, \{*r, *g\}) \to (g, \nu, \{*g\})$ $(\sigma_1 \neq w, g)$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (w, \nu, \{*r\}) \to (g, \nu, \{*g\})$ $(\sigma_1 \neq w, g)$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (w, \nu, \{*r\}) \to (g, \nu, \{\})$ $(\sigma_1 \neq w, g)$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (w, \nu, \{?r, *r, *g\}) \to (g, \nu, \{?r, *r, *g\})$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (w, \nu, \{?r, *r\}) \to (g, \nu, \{?r, *r, ?*g\})$ $(\sigma_1 \neq w, g)$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (w, \nu_1, \{?r_1, *r\}), (\sigma, \nu, \{*r\}) \to (\sigma, \nu, \{*g\})$ $(\sigma_1 \neq g)$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (w, \nu_1, \{?r_1, *r\}), (\sigma, \nu, \{?r, *r\}) \to (\sigma, \nu, \{?r, *r, *g\})$ $(\sigma_1 \neq g)$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (w, \nu_1, \{r, *r, *g\}) \to (g, \mathtt{nil}, \{r, *r, *g\})$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (w, \mathtt{nil}, \{r, *r\}) \to (g, \mathtt{nil}, \{r, *r\})$ $(\sigma_1 \neq w, g)$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (w, w, \{*r, *g\}) \to (g, g, \{*g\})$ $(\sigma_1 \neq w, g)$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (w, w, \{*r\}) \to (g, g, \{?*g\})$ $(\sigma_1 \neq w, g)$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (w, w, \{?r, *r, *g\}) \to (g, g, \{?r, *r, *g\})$

$(\sigma_1, w, \{r, *r, ?*g_1\}), (w, w, \{?r, *r\}) \to (g, g, \{?r, *r, ?*g\})$ $(\sigma_1 \neq w, g)$

$(g, \sigma, \{r, *r, ?*g_1\}), (\sigma, \nu, \{*r, *g\}) \to (\sigma, \nu, \{\}), (\sigma, \nu, \{*r\}), (\sigma, \nu, \{*g\})$ $(\sigma \neq w)$

$(g, \sigma, \{r, *r, ?*g_1\}), (\sigma, \nu, \{r, *r, *g\}) \to (\sigma, \nu, \{r, *r\})$ $(\sigma \neq w)$

$(g, \sigma, \{r, *r, ?*g_1\}), (\sigma, \nu_1, \{r, *r, *g\}) \to (\sigma, \mathtt{nil}, \{r, *r\})$ $(\sigma \neq w)$

$(g, w, \{r, *r, ?*g_1\}), (w, \nu, \{*r, *g\}) \to (g, \nu, \{\}), (g, \nu, \{*r\}), (g, \nu, \{*g\})$

$(g, w, \{r, *r, ?*g_1\}), (w, \nu, \{r, *r, *g\}) \to (g, \nu, \{r, *r\})$

$(g, w, \{r, *r, ?*g_1\}), (w, \nu_1, \{r, *r, *g\}) \to (g, \mathtt{nil}, \{r, *r\})$

$(g, w, \{r, *r, ?*g_1\}), (w, w, \{r, *r, *g\}) \to (g, g, \{r, *r\})$

$(g, w, \{r, *r, ?*g_1\}) \to (g, g, \{\}), (g, g, \{*r\}), (g, g, \{*g\})$

$(w, \nu_1, \{?r_1, *r\}), (\sigma, w, \{*r\}), (\sigma_1, w, \{r, *r, ?*g_1\}) \to (\sigma, g, \{*g\})$ $(\sigma_1 \neq g)$

$(w, \sigma, \{r, *r, *g\}), (\sigma, \nu, \{*r, *g\}) \to (\sigma, \nu, \{\}), (\sigma, \nu, \{*r\}), (\sigma, \nu, \{*g\})$ $(\sigma \neq w)$

$(w, \sigma, \{r, *r, *g\}), (\sigma, \nu, \{r, *r, *g\}) \to (\sigma, \nu, \{r, *r\})$ $(\sigma \neq w)$

$(w, \sigma, \{r, *r, *g\}), (\sigma, \nu_1, \{r, *r, *g\}) \to (\sigma, \mathtt{nil}, \{r, *r\})$ $(\sigma \neq w)$

$(w, \sigma, \{r, *r\}), (\sigma, \nu, \{*r, ?*g\}) \to (\sigma, \nu, \{?*g\})$ $(\sigma \neq w)$

$(w, w, \{r, *r, *g\}), (w, \nu, \{*r, *g\}) \to (g, \nu, \{\}), (g, \nu, \{*r\}), (g, \nu, \{*g\})$

$(w, w, \{r, *r, *g\}), (w, \nu, \{r, *r, *g\}) \to (g, \nu, \{r, *r\})$

$(w, w, \{r, *r, *g\}), (w, w, \{*r, *g\}) \to (g, g, \{\}), (g, g, \{*r\}), (g, g, \{*g\})$

$(w, w, \{r, *r, *g\}) \to (g, g, \{r, *r\})$

$(w, w, \{r, *r, ?*g_1\}), (\sigma, w, \{?r, *r\}), (w, \nu_1, \{*r\}) \to (\sigma, g, \{?r, *g\})$ $(\sigma \neq w, g)$

$(w, w, \{r, *r, ?*g_1\}), (w, \nu_1, \{*r\}), (\sigma, \nu, \{?r, *r\}) \to (\sigma, \nu, \{?r, *g\})$　$(\sigma \neq w, g)$
$(w, w, \{r, *r\}), (\sigma, w, \{*r\}), (w, \nu_1, \{?r_1, *r\}) \to (\sigma, g, \{*g\})$
$(w, w, \{r, *r\}), (\sigma, w, \{?r, *r\}), (w, \nu_1, \{*r\}) \to (\sigma, g, \{?r, *g\})$
$(w, w, \{r, *r\}), (\sigma, w, \{?r, *r\}), (w, \nu_1, \{?r_1, *r\}) \to (\sigma, g, \{?r, *r, *g\})$
$(w, w, \{r, *r\}), (w, \nu, \{*r, ?*g\}) \to (g, \nu, \{?*g\})$
$(w, w, \{r, *r\}), (w, \nu, \{*r\}) \to (g, \nu, \{*g\})$
$(w, w, \{r, *r\}), (w, \nu, \{?r, *r\}) \to (g, \nu, \{?r, *r, ?*g\})$
$(w, w, \{r, *r\}), (w, \nu_1, \{*r\}), (\sigma, \nu, \{?r, *r\}) \to (\sigma, \nu, \{?r, *g\})$
$(w, w, \{r, *r\}), (w, \nu_1, \{?r_1, *r\}), (\sigma, \nu, \{*r\}) \to (\sigma, \nu, \{*g\})$
$(w, w, \{r, *r\}), (w, \nu_1, \{?r_1, *r\}), (\sigma, \nu, \{?r, *r\}) \to (\sigma, \nu, \{?r, *r, *g\})$
$(w, w, \{r, *r\}), (w, \nu_1, \{r, *r\}) \to (g, \mathtt{nil}, \{r, *r\})$
$(w, w, \{r, *r\}), (w, w, \{*r, ?*g\}) \to (g, g, \{?*g\})$
$(w, w, \{r, *r\}), (w, w, \{*r\}) \to (g, g, \{*g\})$
$(w, w, \{r, *r\}), (w, w, \{?r, *r\}) \to (g, g, \{?r, *r, *g\})$
$(w, w, \{r, *r\}) \to (g, g, \{?r, *r\})$

• store

$(\sigma, \nu, \{?r, *r\}), (g, \nu_1, \{r, *r, ?*g_1\}), (w, \nu_2, \{r, *r\}) \to (\sigma, \nu, \{?r, *r, *g\})$
$(\sigma, \nu, \{?r, *r\}), (w, \nu_1, \{r, *r, ?*g_1\}), (w, \nu_2, \{r, *r\}) \to (\sigma, \nu, \{?r, *r, *g\})$
$(\sigma, \nu, \{?r, *r\}), (w, w, \{r, *r, ?*g_1\}), (w, \nu_1, \{r, *r\}) \to (\sigma, \nu, \{?r, *r, *g\})$
$(\sigma, \nu, \{*r, ?*g_1\}), (w, w, \{r, *r, ?*g_2\}), (w, \nu_1, \{*r, ?*g_2\}) \to (\sigma, \nu, \{*g\})$
$(\sigma, \nu, \{r, *r\}), (g, \nu_1, \{r, *r, ?*g_1\}) \to (\sigma, \nu, \{r, *r, *g\})$
$(\sigma, \nu, \{r, *r\}), (w, w, \{r, *r, ?*g_1\}) \to (\sigma, \nu, \{r, *r, *g\})$
$(\sigma, \nu_1, \{r, *r, ?*g\}), (\sigma_1, \nu_2, \{r, *r, ?*g_1\}) \to (\sigma, \sigma_1, \{r, *r, ?*g\})$
$(\sigma, w, \{r, *r\}), (g, w, \{r, *r, ?*g_1\}), (w, \nu_1, \{?r, *r\}) \to (\sigma, g, \{?r, *r, *g\})$
$(\sigma, w, \{?r, *r\}), (w, w, \{r, *r, ?*g_1\}) \to (\sigma, g, \{?r, *r, *g\})$
$(\sigma, w, \{r, *r, ?*g\}), (w, \nu_1, \{r, *r, *g\}) \to (\sigma, g, \{r, *r, ?*g\})$
$(\sigma, w, \{r, *r, ?*g\}), (w, \nu_1, \{r, *r\}) \to (\sigma, g, \{r, *r, ?*g\})$　$(\sigma \neq w, g)$
$(\sigma, w, \{r, *r\}), (g, w, \{r, *r, ?*g_1\}) \to (\sigma, g, \{r, *r, *g\})$
$(\sigma, w, \{r, *r\}), (w, \nu_1, \{?r_1, *r\}), (\sigma_1, \nu_2, \{r, *r, ?*g_1\}) \to (\sigma, \sigma_1, \{r, *r, *g\})$　$(\sigma \neq g)$
$(\sigma, w, \{r, *r\}), (w, \nu_1, \{r, *r\}) \to (\sigma, g, \{r, *r, *g\})$　$(\sigma \neq g)$
$(\sigma, w, \{r, *r\}), (w, w, \{r, *r, ?*g_1\}) \to (\sigma, g, \{r, *r, *g\})$
$(\sigma_1, \nu_2, \{r, *r, ?*g_1\}), (g, \nu_1, \{r, *r\}) \to (g, \sigma_1, \{r, *r, *g\})$
$(g, g, \{r, *r, *g\}), (\sigma_1, \nu_2, \{r, *r, ?*g_1\}) \to (g, \sigma_1, \{r, *r\})$
$(w, \sigma_1, \{?r, *r\}), (w, \nu_1, \{?r_1, *r\}), (\sigma_1, \nu_2, \{r, *r, ?*g_1\}) \to (w, \sigma_1, \{?r, *r, *g\})$
$(w, w, \{r, *r, ?*g\}), (\sigma_1, \nu_1, \{r, *r, ?*g_1\}) \to (g, \sigma_1, \{r, *r, ?*g\})$
$(w, w, \{r, *r, ?*g\}) \to (g, g, \{r, *r, ?*g\})$
$(w, w, \{r, *r\}), (w, \nu_1, \{?r_1, *r\}), (\sigma_1, \nu_2, \{r, *r, ?*g_1\}) \to (w, \sigma_1, \{r, *r, *g\})$
$(\sigma, w, \{r, *r\}), (w, \nu_1, \{r, *r\}) \to (\sigma, g, \{r, *r\})$　$(\sigma \neq g)$

(平成 21 年 2 月 16 日受付)
(平成 21 年 6 月 2 日採録)

藤川　浩光

1984 年生．2007 年京都大学工学部情報学科卒業．2009 年同大学大学院情報学研究科通信情報システム専攻修士課程修了．同年任天堂株式会社入社．モデル検査，定理証明系等に興味を持つ．

馬谷　誠二（正会員）

1974 年生．1999 年京都大学工学部情報学科卒業．2001 年同大学大学院情報学研究科修士課程修了．2004 年同大学院情報学研究科博士後期課程修了．同年同大学院情報学研究科産学官連携研究員．2005 年同研究科助手．2007 年より同研究科助教．博士（情報学）．プログラミング言語，並列/分散処理に興味を持つ．日本ソフトウェア科学会，ACM 各会員．

八杉　昌宏（正会員）

1967 年生．1989 年東京大学工学部電子工学科卒業．1991 年同大学大学院電気工学専攻修士課程修了．1994 年同大学院理学系研究科情報科学専攻博士課程修了．1993〜1995 年日本学術振興会特別研究員（東京大学，マンチェスター大学）．1995 年神戸大学工学部助手．1998 年京都大学大学院情報学研究科通信情報システム専攻講師．2003 年同大学助教授．2007 年より同大学准教授．博士（理学）．1998〜2001 年科学技術振興事業団さきがけ研究 21 研究員．並列処理，言語処理系等に興味を持つ．日本ソフトウェア科学会，ACM 各会員．

湯淺　太一（フェロー）

1952 年神戸生．1977 年京都大学理学部卒業．1982 年同大学大学院理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987 年豊橋技術科学大学講師．1988 年同大学助教授，1995 年同大学教授，1996 年京都大学大学院工学研究科情報工学専攻教授．1998 年同大学院情報学研究科通信情報システム専攻教授となり現在に至る．理学博士．記号処理，プログラミング言語処理系，並列処理に興味を持っている．著書『Common Lisp 入門』（共著），『C 言語によるプログラミング入門』，『コンパイラ』ほか．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．