適応的オブジェクトのための局面解析手法

鎌田 十三郎† 八杉 昌宏††

並列計算や分散計算において，オブジェクトのデータ更新に関する詳細な情報は，不要な排他制御の除去やデータキャッシュといった高速化にとって重要である．しかし，従来のコンパイラではプログラムの振舞いの変化をとらえることは難しく，プログラマの知識に基づいた手作業の最適化が行われてきた．我々が目指すのは，状況に応じた最適化を自動的に施すことができる適応的なオブジェクトの実現である．そのためのアプローチとして，プログラムを複数の局面から構成されたものとしてとらえることとした．プログラムの振舞いの変化をプログラマに局面として記述してもらい，その記述をもとに処理系が局面に関する情報を解析し，局面に応じた最適化を可能にする．本論文では，その局面解析手法について提案を行う．解析では，プログラマの局面に関する記述から，各コードブロックがどの局面において実行されうるのか（可能局面）と，どのような局面間遷移がありうるのか（可能局面遷移）とを解析する． この 2 種類の情報は本来互いに依存しており，その確定には大域的な解析を必要とする．我々は，解析の高速化のため，(1) メソッド単位に実行可能な可能局面解析と，(2) その結果を利用して行う可能局面遷移の確定の 2 部構成のアルゴリズムを提案している．本解析結果は，すでに排他制御緩和技術などに利用されており，その有効性が示されている．

Phase Analysis Algorithm for Adaptive Objects

Tomio Kamada† and Masahiro Yasugi††

For parallel/distributed programs, precise information about object access pattern is important to eliminate synchronization bottlenecks or utilize cached field data for distributed objects. To apply these optimization techniques, the programmer has to prepare careful synchronization code without support of compilers. Our goal is to realize *adaptive objects* that can automatically and adaptively adopt suitable synchronization optimizations. We introduce the concept of *calculation phases* to treat changes of program behavior characteristic. The programmer can describe about the program phase declaratively using *phase variable*, and the system analyzes the characteristic of each calculation phase to prepare specialized execution code for each phase. This paper proposes the phase analysis that analyzes *possible execution phases* for each code block and *possible phase transitions*. As these two types of information depend on each other, we need global analysis to fix these information. We adopt a two stepped algorithm for efficient analysis. First, it analyzes each method independently to collect possible execution phase information, and secondly it fixes possible phase transition using the analyzed result of the first step. We have applied this analyzer for relaxation of mutual exclusion, and evaluated the effectiveness of our approach.

1. はじめに

並列・分散言語のプログラムの実行においては，排他制御や通信のための処理が実行時間に大きな影響を与える．このため，高速化を目指すプログラマは，プログラマの知識をもとに最適化されたコードを目指し，排他制御規則の緩和によるボトルネック回避や分散オブジェクトの効率的な一貫性制御の実現などを行う．これらの最適化は性能面で大きな向上を目指すことができる一方，プログラマにとっては，プログラムの処理内容に関する知識のみならず詳細な実装記述を行う必要があり，煩雑な作業となっている．

一方，不要な排他制御除去などの最適化を，処理系で自動化する試みも行われている．ただし，多くの解析系はプログラム実行全般にわたって成立する性質のみを利用するため，プログラムの実行中に変化する性質を利用した最適化を行うことができない．例として，あるプログラムがデータの構築局面と利用局面に分か

† 神戸大学工学部情報知能工学科
Department of Computer and Systems Engineering, Faculty of Engineering, Kobe University
†† 京都大学大学院情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University

れていたとする．データ利用局面においては値の更新などが行われず，読み出しのみが頻繁に行われる．つまり，利用局面においては本来一貫性保証のためには排他制御は不要であり，プログラマはこのような変化を把握して局面ごとに最適化されたコードを記述することも可能である．ただし，多くの解析系がプログラムを局面ごとに分離して解析することはなく，これらの知識を利用した自動最適化を行うことはできない．

このため我々は，プログラムを複数の局面に分割してとらえ，各局面ごとのプログラムの性質を解析し，自動最適化に応用するというアプローチをとることとした．しかし，解析機構のみで有用な局面の切り分けを行うのは困難であるため，プログラマが持つ局面に関する知識に頼ることとする．つまり，プログラマに自らの持つ局面に関する知識をプログラム中に記述してもらい，後は処理系が自動的に局面ごとのプログラムの性質を解析し，この情報をもとに自動的に不要排他制御の除去や，局面ごとのコード特化による最適化を行うことを目指す．対象とするのは，Java のような構造化された排他制御構文を持つオブジェクト指向言語である．

本論文では，主に局面情報の解析手法について取り扱う．一方で，解析結果を利用した最適化については本論文では概略を述べるにとどめ，具体的な排他制御緩和への応用については，別論文[13]に譲ることとする．すでに，本解析手法の一部はプロトタイプ実装され，自動排他制御緩和への効果が示されている[13]．

論文構成は，以下のとおりである．まず 2 章において，局面記述から解析，応用へといたる概観を述べ，3 章で本解析の概略を述べ，4 章でメソッド内解析について，5 章で大域解析について解説する．6 章でサンプルプログラムに関しての解析例について述べ，7 章で解析結果の応用例として，排他制御規則の緩和法や分散オブジェクトの効率的実装法について簡単に紹介する．最後に，関連研究とまとめについて述べる．

2. 局面解析の利用にむけて

本章では，局面解析を用いた最適化アプローチの概観をプログラム例を通して述べる．**図 1** の 2 分木プログラムは，木の根に対して新たな要素の挿入を行うプログラムである．挿入操作は，該当するノードにたどり着くまで再帰的に呼び出される．当然，複数プロセッサで並列実行した場合は，木の根付近でボトルネックが発生し実行速度が極端に低下する．

そこで，熟練したプログラマであれば，ボトルネックが起こらないように排他制御区間をより短くしようと最適化を行う．**図 2**☆の例では，子供の木がある場合とない場合とによって排他制御の仕方を切り換えている．子供の木がない場合は排他制御をして子供の木の作成・登録を行い，子供の木がすでにある場合は，排他制御をせず子供の木への登録操作を起動する．このような最適化は，(1) 子供の有無によってプログラムの性質が変化するとプログラマが知っており，(2) 実際に各状況における変数更新の有無などをプログラマが把握しており，(3) その情報をもとに最適化を施す，ことで実現される．一方で，プログラマが状況を確実に把握していなかったり最適化を誤ったりすると，バグを引き起こすことになる．

```
class BinTree {
  int key;
  volatile BinTree left, right;

  void insert(int k) {
    synchronized (this) {
      if (k < key) {
        /* left case */
        if (left != null)
          left.insert(k);
        else
          left = new BinTree(k);
      } else { /* right case */ }
    }
  }
}
```

図 1 2 分木の例
Fig. 1 Binary Tree (Example Code).

```
void insert(int k) {
  if (k < key) {
    /* left case */
    label:while(true) {
      if (left != null)
        left.insert(k);
      else {
        synchronzied (this) {
          if (left != null) continue label;
          left = new BinTree(k);
        }
      }
      break;
    }
  } else { /* right case */ }
}
```

図 2 2 分木の例（最適化例）
Fig. 2 Binary Tree (Optimized Code).

我々は，(1) についてはプログラマの知識に頼ることにする．一方で，(2) 局面情報の解析，(3) 局面情報に基づく最適化については自動化を行うというアプローチをとった．本論文で扱うのは主に (2) の局面解析についてである．(3) 局面情報を利用した最適化例については，別論文[13]で排他制御緩和手法を発表している．本章では，まず局面に関する記述法を紹介し，

☆ 現在の Java のメモリモデルでは本プログラムが意図どおり動作する保証はなく，メモリモデル変更が一部で議論されている．詳しくは文献 1) を参照．

その後局面記述からどのような情報が解析されるのか，どのように応用することが可能かについて簡単に紹介する．

2.1 局面記述

局面に関する知識を記述する手段として，我々は**局面変数**を導入する．局面変数は，各々のオブジェクトの状態を表すための変数である．以下では例として図 1 の 2 分木プログラムに対して，局面記述を行う（**図 3**）．

まず局面の定義であるが，このプログラムでは子の有無が最適化にとって重要である．このため，各オブジェクトの左右の子に対し，**子を持つ局面**と**子を持たない局面**との 2 つの局面に分離する．そこで，

```
class ChildState extends
                        Phase {Empty, Full};
ChildState lstate = Empty, rstate = Empty;
```

のように Phase クラスを拡張する形で局面を定義する．この例では，子の局面 ChildState を Empty（子を持たない局面）と Full（子を持つ局面）に分離し，左右の子の局面をそれぞれ変数 lstate，rstate で表している．つまり，各変数は Empty，Full という局面シンボルをその値として持つ．これらの局面変数へのアクセスは，当該インスタンス内に限定される．局面の更新には以下のように become メソッドを用いる．

```
lstate.become(Full);
```

これにより，局面変数 lstate の示す局面（左の子の局面）が Full へ変化する．解析の都合で，become の引数はリテラルで与えられた局面シンボルに限定している．一方，

```
if (lstate.is(Full)) {...} else {...}
```

のように is メソッドを用いることで，局面に関する分岐を記述することができる．上記の例では左の子の局面が Full のときには then ブロックが，そうでないとき（Empty のとき）には else ブロックが実行されることになる．

最後に，排他制御区間との関係について．上記 become 操作は，必ず対象オブジェクトの synchronized 構文，あるいは，*consistent* 構文[13]内で行われることとする．文献 13) で提案している consistent 構文とは，基本的には synchronized と同様の命令実行順序を守りながら，一貫性に影響のない範囲で排他制御区間の短縮を図るものである．たとえば，ブロック冒頭で定数化した変数への読み込みを行っている場合（図 1 の left != null の場合の left へのアクセスなど），排他制御区間から外すことを許す．いずれにせよ，become 操作はオブジェクト状態を変化させるものであり，排他的な実行が保証される．

```
class BinTree {
  class ChildState extends          /* 局面定義 */
                    Phase { Empty,Full };
  ChildState lstate = Empty;     /* 局面変数宣言 */
  ChildState rstate = Empty;
  int key;
  BinTree left, right;
  /* consistent メソッド */
  consistent void insert(int k) {
    if (k < key) {  /* left case */
      if ( lstate.is(Full) ) /* 局面に関する分岐 */
        left.insert(k);
      else {
        left = new BinTree(k);
        lstate.become(Full);    /* 局面の変化 */
      }
    } else { /* right case */ }
  }
}
```

図 3 局面変数を用いた 2 分木の例
Fig. 3 Binary Tree (Using Phase Variables).

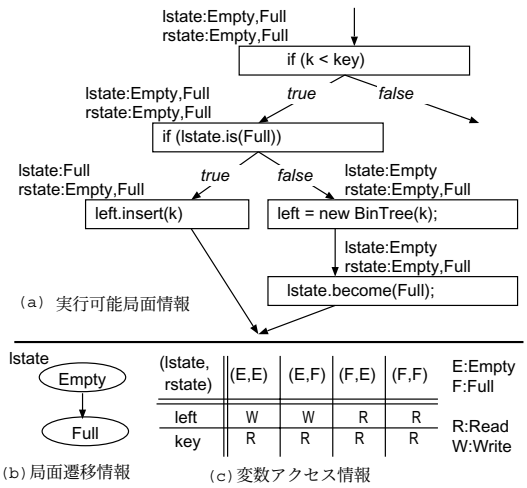

図 4 局面解析情報
Fig. 4 Analyzed Phase Information.

2.2 局面解析によって得られる情報

ここでは局面記述されたプログラムからどのような情報が解析されるかについて述べる．局面解析から得られる情報は主に以下の 2 種類である．

- **A：実行可能局面**
  各命令がどの局面で実行される可能性があるか．
- **B：局面遷移情報**
  どのような局面遷移が可能か．

図 3 のプログラムを解析することにより**図 4** (a) のような実行可能局面情報（A）が得られる．たとえば，left.insert(k); は局面変数 lstate が局面 Full のときのみ実行され，局面 Empty のときに実行されることはない．また left = new BinTree(k); は局面変数 lstate が局面 Empty のときに実行され，局面 Full のときは実行されないと解析できる．

また，B の局面遷移情報であるが，これは図 4 (b) のように解析される．局面変数 lstate は初期局面が Empty であり，局面の遷移としては Empty から Full へ遷移する可能性がある．また一度 Full の局面になると別の局面へ遷移する可能性はないことが分かる．

これら 2 つの情報により，(a) により各々の局面におけるプログラムの性質を解析し，(b) により他のスレッドによる局面遷移の可能性に配慮した最適化が可能になる．たとえば，図 3 のプログラムに対し排他制御緩和を行い，図 2 のように最適化するためには，lstate.is(Full) において left への更新が行われていないというだけでなく，他スレッドによって Full 局面から別の局面に遷移したとしても left は変化しないといった情報が必要である．一方で，本解析結果を使うと (a) から各局面でどのような変数アクセスが行われるか（図 4 (c)）を解析可能である．加えて，(b) から Full および Full から遷移する局面（この場合存在しない）を求め，これら局面に対し left の更新状況を確認することができる．この場合，left の更新は否定できるため，排他制御区間の短縮を達成できる．

3. 局面解析アルゴリズム

3.1 概　　略

本解析では，A：各命令がどのような局面において実行されうるのか（実行可能局面）と，B：どのような局面間遷移がありうるのか（可能局面遷移）とを解析する．解析を行ううえで注意すべき点は，排他制御ブロック外における他スレッドによる局面更新の影響である．同様に，自身のスレッドで呼び出したメソッドによる局面遷移も考慮する必要がある．**図 6** は，サンプルプログラム**図 5** に対して，その実行可能局面の解析結果を示したものである．このプログラムでは，B3 から B4 に至るパスにおいて分岐直後の局面は Q であると考えられるが，他スレッドの become 操作（ブロック B6）により局面が Q から R に更新される可能性があるため，最終的に B4 の実行可能局面は Q，R と判定される．一方で，その Q から R への局面遷移の可能性は，ブロック B6 の実行可能局面に Q が含まれることに起因する．このように，A と B の解析は互いに依存しあった関係にあり，加えて，B は当該クラスの全メソッドの解析が終わらないと確定することができない．

我々は，これらの相互依存した情報の解析を高速に行うために，

(1) メソッド単位で行うメソッド内解析（4 章）
(2) その結果を利用して行う大域解析ならびに可能局面の確定作業（5章）

の 2 部構成のアルゴリズムを提案する．大域解析においてはクラス内の局面遷移情報をとりまとめて確定を行い，必要に応じてメソッド呼び出し関係を考慮した解析が可能な枠組みも提供している．

```
void func() {                    /* B1 */
  while(true) {
    ....;                        /* B2 */
    if(!phase.is(Q)) break;      /* B3 */
    synchronized(this) {
      if(...) {                  /* B4 */
         m();                    /* B5 */
      } else {
         phase.become(R);        /* B6 */
         .....;                  /* B7 */
      }
    }                            /* B8 */
  }
  return;                        /* B9 */
}
```

図 5 解析サンプルプログラム
Fig. 5 Sample Program for Phase Analysis.

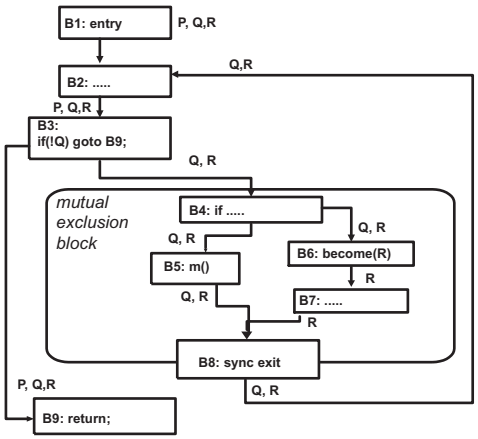

図 6 可能局面解析（確定）
Fig. 6 Analyzing Phase Information (Final Step).

メソッド内解析は当該クラスの全メソッドを対象に行われ，メソッドをコントロールフローグラフ（以下 CFG）に分解し，フロー解析を行う．ただし，局面遷移解析が終了していない段階では，具体的な局面を確定することは当然できない．第 1 段階のメソッド内解析では，準備として「もしメソッド開始時に Q が可能局面であり，他のスレッドなどによって Q から R への遷移が可能であれば，このブロックは局面 R で実行されうる」といった情報を求める．**図 7** は，メソッド内解析結果を示したもので，B4 の実行可能局面の R : Cq · Q→*R は上の情報を意味している．メソッド中の become 操作についても，局面遷移内容とその成立条件が同様に解析される．

大域解析においては，先ほど解析された become 操

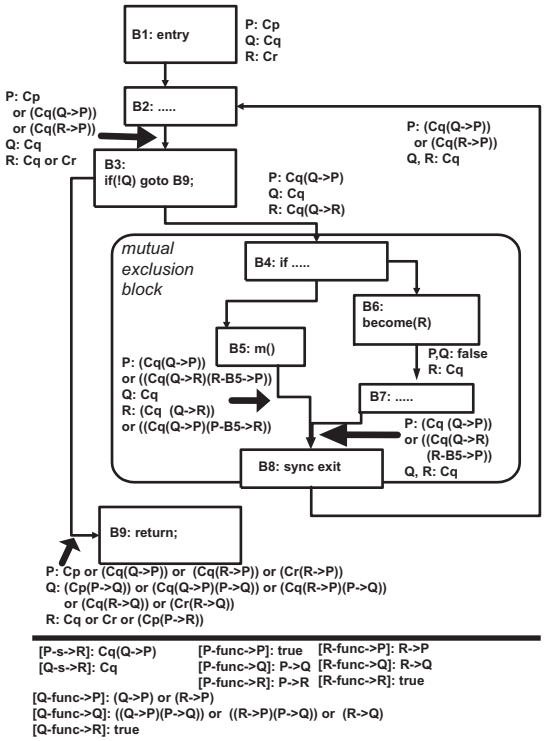

図 7 可能局面解析（第 1 段階）
Fig. 7 Analyzing Phase Information (First Step).

作による局面遷移に関する情報，つまり，「局面 P から Q への局面遷移が可能であれば，局面 R から S への become の可能性がある」といった各メソッドの情報をとりまとめ，実際に可能性のある局面遷移を求めることができる．その結果，可能局面の解析結果も確定することができる．つまり，各コードブロックが具体的にどの局面において実行される可能性があるのかを求めることができる．

最後に，2.1 節で触れた consistent 構文との関係について，本解析では，consistent と synchronized はまったく同じ扱いをうける．つまり，consistent 構文中でも，他スレッドの影響をいっさい無視して解析を行う．システムは本解析結果に基づきつつ，局面遷移の可能性を考慮した排他制御緩和を行う．

3.2 モデル

アルゴリズムについて説明をする前に，モデルといくつかの術語の説明を行う．まず局面であるが，解析精度の問題から局面値を一般の一時変数などに格納することを禁止している．また，本論文では局面値は一般に P，Q，R などの記号を用いて表すか，添字を添えて P_i などと表す．局面値の集合は以後 $\boldsymbol{P}$ と表す．

実行可能局面，つまり各時点においてどの局面の可能性があるかを，

$$S = \{P_0 : Cond_0, \cdots, P_n : Cond_n\}$$

で表す．$Cond_i$ は真偽値を表す論理式であり，局面 P_i である可能性の有無を示す．また，$S(P_i) = Cond_i$ として表記する．解析終了時点では，$S(P_i)$ は true/false が定まり，局面 P_i であった可能性の有無が定まる．$Cond$ には，論理積演算 $\cdot$ と論理和演算 $\vee$ が存在する．加えて，実行可能局面 S の間に和集合計算 $\cup$ を，以下のように定める．

$$S \cup S' = \{P_i : S(P_i) \vee S'(P_i) \mid P_i \in \boldsymbol{P}\}$$

次に，become 操作と排他制御ブロックについて．2.1 節で述べたように局面変数の値更新は，become 操作によって排他制御区間内で実行される．なんらかの become 操作によって P から Q になる可能性の有無を $P \rightarrow^s Q$ で表し，真偽値 true/false を値としてとる．次に，$P \rightarrow^* Q$（図中では，単に P->Q と表記）は，$P = P_0 \rightarrow^s \cdots \rightarrow^s P_n = Q$（n は 0 以上）なる遷移の可能性の有無を示す．つまり局面 P から 0 回以上の become の遷移で局面 Q に到達しうる可能性である．$P \rightarrow^* P$ はつねに成立するものとする．

我々の解析では，可能局面と可能局面遷移を以下のように定める．

- 排他制御区間外のある時点で局面 P が可能局面であり，また $(P \rightarrow^* Q) =$ true であれば，局面 Q も可能局面である．
- ある基本ブロックの入口で局面 P が可能局面であれば，become 操作や局面の制限が行われない限りブロックの出口でも P は可能局面である．
- 局面 Q への become 操作が記述されている個所で，局面 P が可能局面であれば，$(P \rightarrow^s Q) =$ true である．

最後に，メソッド呼び出し関係解析を行うために，呼び出されるメソッド（$func$）とメソッド呼び出し命令（$call$）に対して，局面遷移可能性 $P_i \rightarrow^{func} P_j$，$P_i \rightarrow^{call} P_j$ を導入する．$func$ をメソッド間解析の対象とする場合は，実行前の実行可能局面 S^{func}_{init} に対して実行後の実行可能局面 S^{func}_{ret} が以下の式を満たすように $P_i \rightarrow^{func} P_j$ を定めるものとする（for all $P_j \in \boldsymbol{P}$）．

$$S^{func}_{ret}(P_j) = \vee_{P_i \in \boldsymbol{P}} (S^{func}_{init}(P_i) \cdot (P_i \rightarrow^{func} P_j))$$

直観的には，$P_i \rightarrow^{func} P_j$ は，$func$ 開始時に P_i が可能局面の場合に終了時の局面が P_j となる可能性の有無を示す．当然，$(P_i \rightarrow^{func} P_j) \Rightarrow (P_i \rightarrow^* P_j)$（$\Rightarrow$ は含意関係）である．$call$ に関しても同様である．呼び出し関係解析を行わない場合は，保守的な解析を行い，$\rightarrow^{func}$，$\rightarrow^{call}$ の代わりに $\rightarrow^*$ を利用する．

4. メソッド内解析

メソッド内解析の段階では可能局面遷移などが定まっておらず,「もしメソッド開始時に Q が実行局面である可能性 (C_q) が true で，他のスレッドなどによって Q から R への遷移が可能であれば，このブロックは局面 R で実行されうる」といった解析が行われることになる．実行可能局面 S に対して，$S(R) = C_q \cdot (Q \rightarrow^* R)$ などと表現される．厳密には，メソッド内解析中の $S(P_i) = Cond_i$ 部は，積和標準形の正規化された論理式をとり，その原子論理式は，メソッド開始時に局面 P_i である可能性を示す論理変数 $Cond^{init}_{P_i}$，もしくは局面遷移の可能性を示す $P_i \rightarrow^* P_j$，あるいは $P_i \rightarrow^{call} P_j$（$call$ は本メソッド中の呼び出し命令に限定される）である．また，論理演算 $\cdot, \vee$ においては，論理式を合成し，正規化を行うこととする．正規化では，演算が積の場合は積和形に分配し，積の中の 2 つの原子論理式 A, B について $A \Rightarrow B$ なら B を省略し，演算が和の場合は和の中の 2 つの論理式 A, B について $A \Rightarrow B$ なら A を省く．この際，関係式 $P \rightarrow^{call} Q \Rightarrow P \rightarrow^* Q$ を利用するが，$\rightarrow^*$ に関する推移律関係は利用していない．

メソッド内解析では，実行可能局面 S に関するフロー解析を行う．ブロック b の前後の状態を，それぞれ S^b_{in}，$S^{b \rightarrow b'}_{out}$ で表す．ただし，b' は対応する後続ブロックを表し，対応する後続ブロックが 1 つしかない場合は，単に S^b_{out} とも表記する．ブロック前後の状態に関しては一般のデータフロー方程式

$$S^b_{in} = \cup_{b' \in Pred(b)} S^{b' \rightarrow b}_{out}$$

が成立する．ただし，$Pred(b)$ は b の先行ブロックを示す．b における可能局面は S^b_{in} によって表す．

メソッド内解析においては，まずメソッドを CFG に変換する．以下の点が一般の CFG と異なる．

- 排他制御ブロックに含まれるか否かで基本ブロックが分かれるようにし，加えて，ロック開放ポイントを独立した基本ブロックとして扱う．
- 局面に関する条件分岐命令を独立した基本ブロックとして扱う．
- become 文やメソッド呼び出しを単独の基本ブロックとして取り扱う．

データフロー解析を行うにあたって，基本ブロックは以下の 6 種に分かれる．各種類ごとに

$$S^b_{out} = f^b(S^b_{in})$$

なる S^b_{in} と S^b_{out} の関係を表す変換関数が定められている．以下，図 7 の例を通して説明する．

1. メソッド冒頭部： 現時点では，メソッド開始時の実行可能局面は確定できないため，各局面 P_i の可能性を変数 $Cond^{init}_{P_i}$ で表す．

$$S^b_{out} = \{P_i : Cond^{init}_{P_i} \mid P_i \in \boldsymbol{P}\}$$

また，メソッド $func$ に対して，初期可能局面を S^{func}_{init} と示す．図 7 においては，ブロック B1 における可能局面は，$\{P : Cp, Q : Cq, R : Cr\}$ となる．このように初期局面を変数で与えているのは，メソッド間解析に備えるためである．つまり，このメソッド（仮に func0）による局面遷移 $\rightarrow^{func0}$ を求める必要と，メソッド開始時の局面が限定できるケースに備えるためである．一方で，このメソッドの呼び出しについてメソッド間解析しない場合，$\{P_j : true \mid P_j \in \boldsymbol{P}\}$ として解析を行ってもかまわない（詳しくは 5 章）．

2. ロック開放ブロック： 排他制御ブロックが終了するポイントである．つまり，排他制御ブロック内においては，他のスレッドによる局面遷移の影響が無視されていたが，この時点から考慮する必要がある．もし，ロック開放ブロック内で P_k が可能局面であり，$P_k \rightarrow^* P_l$ であれば，今後は P_l も可能局面となる．つまり，S^b_{in} に対して，

$$S^b_{out}(P_j) = \vee_{P_i \in \boldsymbol{P}} (S^b_{in}(P_i) \cdot (P_i \rightarrow^* P_j))$$

として定めることができる．

たとえば，図 7 においては，ブロック B8 における可能局面は，その前後で局面 P の可能性が増加する．$S^{B8}_{in}(P) = (Cq \cdot (Q \rightarrow^* P)) \vee (Cq \cdot (Q \rightarrow^* R) \cdot (R \rightarrow^{B5} P))$ に対して，$S^{B8}_{out}(P)$ では局面 Q，R からの遷移の可能性 $Cq \cdot (Q \rightarrow^* P)$, $Cq \cdot (R \rightarrow^* P)$ が加わった結果，図のように定まる．$(Cq \cdot (Q \rightarrow^* R) \cdot (R \rightarrow^{B5} P)) \Rightarrow (Cq \cdot (R \rightarrow^* P))$ が正規化の際に利用されている．

3. 局面に関する条件分岐： 話を簡単にするため，本論文では条件分岐命令を基本ブロックとして取り扱っている．if 文の条件式として局面に関する記述を行うことによって，後続のブロックの可能局面を制限することになる．ただし，条件分岐命令が排他制御ブロック内にあるか否かによって状況は異なる．

排他制御ブロック内にある場合は，単純に局面に関する条件式に基づいて可能局面を振り分ければよい．今，条件分岐ブロック b の条件式において，分岐枝 $e = b \rightarrow b'$ への分岐を局面 $Pset(e)$ に限定しているとする．この場合，S^b_{in} に対して，e への出力状態 S^e_{out} は以下のように定まる．

$$S^e_{out}(P_i) = \begin{array}{ll} S^b_{in}(P_i) & \text{for } P_i \in Pset(e) \\ \texttt{false} & \text{for } P_i \notin Pset(e) \end{array}$$

ただし，条件分岐が排他制御ブロック外にある場合は，再度他のスレッドによる局面遷移を考慮する必要がある．つまり，S^e_{out} は以下のように定まる．

$$S^e_{out}(P_j) = \vee_{P_k \in Pset(e)}(S^b_{in}(P_k) \cdot (P_k \rightarrow^* P_j))$$

図 7 のブロック B3 は排他制御ブロック外での局面に関する条件分岐を示したものである．分岐後ブロック B4 に至る場合，分岐命令実行直後の可能局面は，{P : false, Q : Cq, R : false} であり，$S^{B3 \rightarrow B4}_{out}$ はさらに他のスレッドによる局面遷移を考慮したものとなっている．

4. become 操作： become 操作によって局面が P_k に更新される場合，当然可能局面は P_k に限定される．ただし，このブロックが実行可能になるための条件式を考慮して，S^b_{in} に対して，

$$\begin{array}{ll} S^b_{out}(P_k) = & \vee_{P_i \in \boldsymbol{P}} S^b_{in}(P_i) \\ S^b_{out}(P_j) = & \texttt{false} \qquad \text{for } P_j \neq P_k \end{array}$$

なる S^b_{out} を出力状態としている．become 操作それ自体は排他制御ブロック内で行われる保証があるので，ここでは他のスレッドの影響を考えない．最終的にすべての $S^b_{in}(P_i) = \texttt{false}$ であれば，この基本ブロックは到達不可能であり，become はされえないことになる．図 7 のブロック B6 では，B6 に到達するための条件式は $\texttt{Cq} \cdot (\texttt{Q} \rightarrow^* \texttt{P})$, Cq, $\texttt{Cq} \cdot (\texttt{Q} \rightarrow^* \texttt{R})$ の論理和である Cq となる．つまり，become 直後の可能局面は {P : false, Q : false, R : Cq} と定まる．

5. メソッド呼び出し： 排他制御ブロック外では，そもそも他のスレッドによる影響やメソッド呼び出しの影響を考慮してあるので，メソッド呼び出しによって新たに可能局面が増えることはありえない．つまり，

$$S^b_{out} = S^b_{in}$$

となる．

一方，排他制御ブロック内の場合，メソッド実行中に行われる become 操作による局面遷移を考慮する必要がある．このため，該当メソッド呼び出し命令（call）による局面遷移（$\rightarrow^{\texttt{call}}$）の定義に基づいて，メソッド呼び出し後の局面は

$$S^b_{out}(P_j) = \vee_{P_i \in \boldsymbol{P}}(S^b_{in}(P_i) \cdot (P_i \rightarrow^{\texttt{call}} P_j))$$

と定まる．最終的には，大域解析によって $\rightarrow^{\texttt{call}}$ と呼び出されるメソッド（func）に関する $\rightarrow^{\texttt{func}}$ の対応関係をとり，遷移関係が決定されることになる．ただし，呼び出されるメソッドが特定できない場合など，呼び出し関係解析を行わない場合は，$\rightarrow^{\texttt{call}}$ の代わりに $\rightarrow^*$ を利用し保守的な見積りを行うことになる．一方で，呼び出し関係から当該メソッド呼び出し中に自オブジェクトの局面更新は行われないと分かっている場合は，局面遷移の可能性を否定して解析を行うこととなる．もし，これらの情報が事前に分かっている場合は，メソッド内解析時点でその情報を利用してもかまわない．

6. その他： その他のブロックにおいては，排他制御内外に限らず

$$S^b_{out} = S^b_{in}$$

として定めることができる．なぜなら，排他制御外であっても，すでに $b' \in Pred(b)$ の $S^{b'}_{out}$ の時点において，他のスレッドの影響による可能性はつくされているためである．

第 1 段階の解析では，以上で定まるデータフロー方程式を満たす最小不動点を解として求める．各状態の値は有限であり，また単調増加するため，反復法などを用いた場合の停止性が保証される．この情報をもとに become 操作による局面遷移条件を確定することができる．become 操作ブロック b による遷移対象が P_k であったとする．この場合，$S^b_{in}(P_i) = \texttt{true}$ であれば，$P_i \rightarrow^s P_k$ なる局面遷移が可能であるといえる．

図 7 の例においては，S^{B6}_{in} は，$\{\texttt{P} : \texttt{Cq} \cdot (\texttt{Q} \rightarrow^* \texttt{P}), \texttt{Q} : \texttt{Cq}, \texttt{R} : \texttt{Cq} \cdot (\texttt{Q} \rightarrow^* \texttt{R})\}$ である．この可能局面において，become(R) が行われているため，$\texttt{Cq} \cdot (\texttt{Q} \rightarrow^* \texttt{P}) = \texttt{true}$ ならば $\texttt{P} \rightarrow^s \texttt{R} = \texttt{true}$ といえる．$[\texttt{P} \rightarrow^s \texttt{R}] : \texttt{Cq} \cdot (\texttt{Q} \rightarrow^* \texttt{P})$ と表記する．同様に，$[\texttt{Q} \rightarrow^s \texttt{R}] : \texttt{Cq}$ も成立する．

また，このメソッド (func0()) を起動することによる局面遷移の可能性 $\rightarrow^{\texttt{func0}}$ も定まる．図 7 の場合メソッドの出口は B9 のみであるため，その実行可能局面情報 $S^{B9}_{in} = S^{\texttt{func0}}_{ret}$ と $S^{\texttt{func0}}_{init}$ を比較して $\rightarrow^{\texttt{func0}}$ を定めることになる．もし出口が複数ある場合は，その和が $S^{\texttt{func0}}_{ret}$ となる．たとえば，$\texttt{P} \rightarrow^{\texttt{func0}} \texttt{Q}$ は，$S^{B9}_{in}(\texttt{Q})$ に対して $\texttt{Cp} = \texttt{true}, \texttt{Cq} = \texttt{false}, \texttt{Cp} = \texttt{false}$ を代入することで求まる．この場合，$\texttt{P} \rightarrow^{\texttt{func0}} \texttt{Q} \Leftarrow \texttt{P} \rightarrow^* \texttt{Q}$ と定まる．以下，$[\texttt{P} \rightarrow^{\texttt{func0}} \texttt{Q}] : \texttt{P} \rightarrow^* \texttt{Q}$ と表記する．

現実には，$\rightarrow^{func}$ は排他制御ブロック内のメソッド呼び出し命令の解析のためにのみ利用される．このため，より正確な解析を行いたい場合はメソッド全体を排他制御ブロックにいれた形で $\rightarrow^{func}$ 用の解析を再度行ってもよい．

5. 大域解析

局所解析では，局面遷移可能性 $P_i{\rightarrow}^* P_j, P_i{\rightarrow}^{func} P_j$, $P_i{\rightarrow}^{call} P_j$ と，各メソッド開始時の実行可能局面 S_{init}^{func} 間の制約条件が求まっている．また，各メソッド呼び出し命令における実行可能局面 S_{in}^{call} も求まる．大域解析では，メソッド呼び出し関係を考慮しつつ，以上の制約を満たす最小の解を求める．

局所解析から，

- $P_i{\rightarrow}^s P_j$
- $P_i{\rightarrow}^{func} P_j$
- $S_{in}^{\mathtt{call}}$

に関する制約が項の積和形として定まっている．たとえば，$[\mathtt{P}{\rightarrow}^s\mathtt{R}] : \mathtt{Cq} \cdot (\mathtt{Q}{\rightarrow}^*\mathtt{P})$ であれば，$(\mathtt{P}{\rightarrow}^s\mathtt{R}) \Leftarrow (\mathtt{Cq} \cdot (\mathtt{Q}{\rightarrow}^*\mathtt{P}))$ を意味する．このため

- $P_i{\rightarrow}^* P_j$
- $P_i{\rightarrow}^{call} P_j$
- S_{init}^{func}

に関する制約を同様に定めれば，後はこの制約を満たす解を greedy に求めることができる．つまり，初期状態ではすべての値を `false` とし，制約に応じて必要な項を `true` に変更するという操作を制約を充足するまで繰り返す．制約を否定項などを含まない純粋な積和形ですべて表すことができれば，アルゴリズムの単調性と停止性が保証できる．

$P_i{\rightarrow}^* P_j$ は，P_i から P_j に $\rightarrow^s$ の 0 回以上の繰返しで到達できるかどうかを示すものであり，単調性も明らかである．

$P_i{\rightarrow}^{call} P_j$ と S_{init}^{func} に関する制約は，メソッド呼び出し関係の解析から定まる．呼び出されるメソッドの開始時の実行可能局面 S_{init}^{func} は，呼び出し命令の実行可能局面の総和で求めることができる．もし直接あるいは間接的に当該メソッドを呼び出している自クラス中の呼び出し命令（`call`）がすべて特定できている場合は，$S_{in}^{\mathtt{call}}$ の論理和をとる．一方で，メソッドの呼び出し元が特定できない場合などは，初期局面 P_0 から $\rightarrow^*$ 到達可能な全局面の可能性がある．つまり保守的に $S_{init}^{func}(P_0) = \mathtt{true}$，$S_{init}^{func}(P_i) = P_0{\rightarrow}^* P_i$ として解析を行うことになる．

メソッド呼び出し命令 `call` による局面遷移に関しては，`call` によって直接あるいは間接的に自オブジェクトのメソッドが呼び出される場合，その可能性を考慮して $P_i{\rightarrow}^{\mathtt{call}} P_j$ を定める必要がある．たとえば，自オブジェクトへのメソッド `func` 呼び出しが内部で 1 回以上行われているか，あるいは行われていないかもしれない場合は，以下のように定める．

$$
\begin{aligned}
P_i{\rightarrow}^{\mathtt{call}} P_i \quad &\Leftarrow \mathtt{true} \\
P_i{\rightarrow}^{\mathtt{call}} P_j \quad &\Leftarrow \vee_{k \in \boldsymbol{P}} (P_i{\rightarrow}^{\mathtt{call}} P_k) \\
&\quad \cdot (P_k{\rightarrow}^{\mathtt{func}} P_j)
\end{aligned}
$$

一方で，もし呼び出されるメソッドが特定できない場合は，$P_i{\rightarrow}^{call} P_j = P_i{\rightarrow}^* P_j$ として保守的な見積りを行うことになる．

図 7 の例の場合，メソッド内解析からは

$$
\begin{aligned}
\mathtt{P}{\rightarrow}^s\mathtt{R} \quad &\Leftarrow \quad \mathtt{Cq} \cdot (\mathtt{Q}{\rightarrow}^*\mathtt{P}) \\
\mathtt{Q}{\rightarrow}^s\mathtt{R} \quad &\Leftarrow \quad \mathtt{Cq}
\end{aligned}
$$

と $\rightarrow^{func}$ に関する式が定まっている．仮に，他にメソッドが存在せず，メソッド呼び出しについては保守的な見積りをする場合は，$\rightarrow^s$ と $\rightarrow^*$ に関する関係式に加えて，以下の関係が成立する．

$$
\begin{aligned}
\mathtt{Cp} \quad &\Leftarrow \quad \mathtt{true} \\
\mathtt{Cq} \quad &\Leftarrow \quad \mathtt{P}{\rightarrow}^*\mathtt{Q} \\
\mathtt{Cr} \quad &\Leftarrow \quad \mathtt{P}{\rightarrow}^*\mathtt{R}
\end{aligned}
$$

もし，他に条件がない場合，いっさい局面遷移なしという解になる．一方，別のメソッドのメソッド内解析から

$$
\mathtt{P}{\rightarrow}^s\mathtt{Q} \quad \Leftarrow \quad \mathtt{true}
$$

である場合は，$\{\mathtt{P}{\rightarrow}^s\mathtt{Q}, \mathtt{Q}{\rightarrow}^s\mathtt{R}\}$ なる局面遷移が起こることになる．また，この条件のもと，図 7 の例について可能局面を確定すると，図 6 のように確定する．

6. 解析手法の評価

本章では，サンプルプログラムを通して本解析手法の有効性を議論する．評価するのは文献 13) でも利用する 2 分木プログラムと N 体問題プログラムである．

まずは，2 章で利用した 2 分木プログラムについて．このプログラムの場合，局面変数は 2 つ存在するが今回は別々の解析を行う．`insert()` のメソッド内解析の結果（**図 8** (a)），局面遷移の可能性としては，$[\mathtt{Empty}{\rightarrow}^s\mathtt{Full}] : \mathtt{Ce} \cdot (\mathtt{Empty}{\rightarrow}^{\mathtt{B4}}\mathtt{Empty})$ が検出される．また，コンストラクタの解析に関しては，内部でメソッド呼び出しを行わない場合は特別に排他制御ブロック内と同じ扱いで解析を行い，局面遷移なし（内部はつねに初期局面のまま）として処理している．

大域解析においては，コンストラクタ呼び出し（`B4,B10`）についてのみ呼び出し関係解析を行うと想定して話を進める．本来，コンストラクタと呼び出し側では `this` に相当するインスタンスが異なるが，これについては解析不能だったとする．この場合でも，コンス

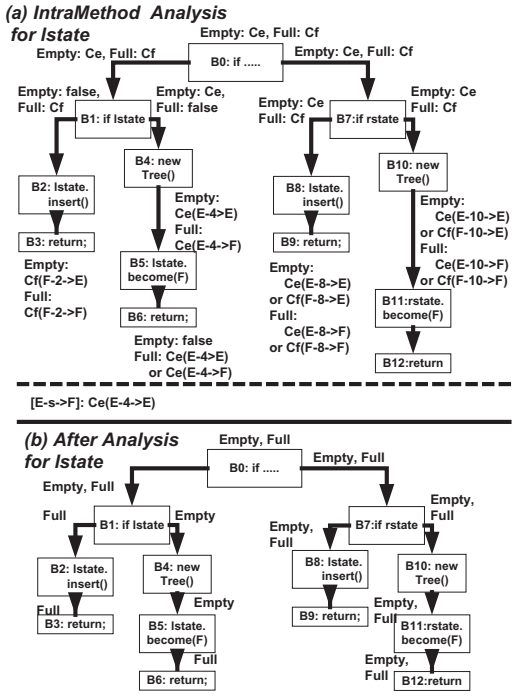

図 8 2 分木プログラムの解析
Fig. 8 Analysis of Binary Tree Program.

トラクタ内で局面更新は行われないと判定でき，$\rightarrow^{B4}$, $\rightarrow^{B10}$ は局面変化なしと判断する．一方で，`insert()` メソッド呼び出しは呼び出し関係解析を行わなかったため，初期局面は，`Ce`⇐`true`, `Cf`⇐(`Empty`$\rightarrow^{*}$`Full`) となる．また，$\rightarrow^{B2}$, $\rightarrow^{B8}$ の代わりに $\rightarrow^{*}$ を使うこととなる．これらの情報をとりまとめると，結果的に $\rightarrow^{*}$ については `Empty`$\rightarrow^{*}$`Full` が確定し，`insert()` メソッド中の各ブロックも妥当な実行可能局面に定まる（図 8 (b)）．

次に，N 体問題プログラムについて．このプログラム［文献 3) にて公開］は J. Barnes & P. Hut アルゴリズム[4]を採用しており，8 分木構造に対応する Node クラスが計算の主体となる．**図 9** は，メソッド構成と局面遷移部を簡単に書き下したものである．本プログラムでは，計算の段階に応じて変数アクセス状況が異なるため，Node オブジェクトを木の構築局面（子供の数で 3 種に分類：`CLeaf`，`CPartial`，`CFull`）と，重心計算局面（`Mass`），加速度計算局面（`Acc`）に分離した．メソッドは，他オブジェクトからの呼び出されるメソッド（コンストラクタと他 3 種）以外に，内部用途の 3 種のメソッドがある．局面記述は，外部公開されたメソッドについて，その起動局面に関する制約をユーザの知識として記述している．

処理内容が再帰的であるため，外部公開メソッドについては呼び出し関係解析不能とする．他方，内部処理用メソッドや他クラスのメソッド呼び出しについては，呼び出し段数もたかだか 2 段程度のため，呼び出し関係の解析を行うこととする．以上の条件のもと解析を行った結果，`CLeaf`$\rightarrow^{s}$`CPartial`, `CPartial`$\rightarrow^{s}$`CFull`, `CLeaf`/`CPartial`/`CFull`$\rightarrow^{s}$`Mass`, `Mass`$\rightarrow^{s}$`Acc` への 1 方向の局面遷移が確認できる．また，内部メソッドについても実行可能局面が解析されており，`setChildren()` メソッドの実行局面が `Cpartial` 局面だけに制限できる．一方で，もし呼び出し関係解析をいっさい行わなかった場合，内部メソッドの実行可能局面はユーザの明示的指示を省略すると正しく解析できなかった．また，局面遷移関係に関しても，注意深くプログラムを書かないと内部メソッド呼び出しの影響で，解析結果が甘くなるケースがあった．

以上のプログラムでは，再帰呼び出しなどについて呼び出し関係解析を行わず，インライン可能な程度のメソッドについてしか呼び出し関係解析を行っていない．以上の呼び出し関係解析を行っただけでも，外部公開されたメソッド中の局面記述の結果，十分な解析成果が得られている．一方で，内部処理的なメソッドについては，呼び出し関係解析により詳細な局面記述

```
class Node {
  class MyState extends Phase
      {CLeaf, CPartial, CFull, Mass, Acc};
  internal MyState  mystate  = CLeaf;

  /* 内部用途 */
  consistent Node children(int i) { ... }
  consistent void setChildren(int, Node) { ... }
  void makeChild(...) { ...; setChildren(...); }

  /* 外部公開されたメソッド */
  Node(...) {  /* field 初期化のみ */ }

  consistent void insert(.....) { // 構築局面
    if(mystate.is(CFull)) {
      ...;
    } else if(mystate.is(CPartial)) {
      if(...) mystate.become(CFull);

    } else if(mystate.is(CLeaf)) {
      mystate.become(CPartial); ...;

    } else { throw new Error(); }
  }
  consistent void getCenterMass(CenterInfo) {
    // 構築局面のはず
    assert(!(mystate.is(Acc)||mystate.is(Mass)));
    mystate.become(Mass); // 重心局面へ.
    ...;
    mystate.become(Acc);  // 計算局面へ.
  }
  consistent void calAcc(Particle) { // 主計算部
    assert(mystate.is(Acc));   // 計算局面のはず
    ....;
  }
}
```

図 9 N 体問題プログラムの解析
Fig. 9 Analysis of N-Body Problem Program.

が不要となっている．今後の局面の利用法として，一般のユーティリティクラスを，局面情報を利用した同期部でラップして利用する場合を考えると，単純な呼び出し関係解析でも効果的に機能するのではないかと期待している．

一方で，呼び出し関係解析をより強化する場合，インスタンスの種別を行うことが重要である．上記プログラムの再帰呼び出しは下降的であり，実際には自オブジェクトの局面更新を引き起こすことはない．より精密な解析を目指すには，再帰や繰返しを考慮したうえで自オブジェクトの参照がどの範囲に伝搬しているかを解析する必要がある．

最後に，メソッド呼び出しの影響を考慮した局所解析を行うと，実行可能局面を表す式が大きく膨らむ傾向にある．特に，逐次的な関数呼び出しが行われた場合，積和表現のままでは組合せが大きくなる恐れがある．このため，実際の実装過程ではより効率的な内部表現をとることが重要になると考える．

7. 最適化への応用

本章では，本解析結果を利用した最適化技術についていくつか紹介をする．1つめの例は，2章でも紹介した並列プログラムの排他制御緩和法である[13]．基本的なアプローチは，局面ごとに変数の更新の有無を解析し，変化しない変数へのアクセスは排他制御せずに行うというものである．局面遷移を考慮したうえで変数へのアクセスを3種類に分類し，それぞれに応じた排他制御規則を適用する．分類は，アクセス対象の変数が今後いっさい更新されないか，あるいは現在の局面においては更新されないか，あるいは更新をともなうかである．文献13)において，本解析アルゴリズム（ただし，メソッド間解析なし）に基づいたプロトタイプを作成し，6章で紹介したプログラムについて評価している．結果，両プログラムに対して排他制御緩和に成功し，プログラムのボトルネックの自動削除に効果を示している．

同様の技術は，分散オブジェクト実装に対しても有効に機能すると考えられる．つまり，分散オブジェクトの変数について，ある局面や，その局面以降変化しない変数を分類しておき，プロキシオブジェクトにキャッシュして利用する．本解析では，各コードがどの局面で実行されているかも分かっているため，もし，あるメソッド中のフィールドアクセスがすべてキャッシュ化可能と分かった場合は，そのメソッド自体をプロキシ上で局所実行することが可能となる．一方で，キャッシュ不可能な変数へのアクセスが多い場合は，本体オブジェクト上でメソッド実行をするという選択も可能である．

これらの技術は，局面に関してcode versioningを行うことで，さらなる最適化を行うことができる．たとえば，6章のN体問題プログラムにおいて，`getCenter()`という`center`の位置を返すだけのメソッドがあったとする．ただ，`center`自身は`FCalc`では更新されないが，`CMass`では更新される．つまり，`getCenter()`は`FCalc`では最適化可能であるが，`CMass`では通常どおりの実行が必要となる．対処策として，たとえばメソッド開始時の局面に応じた別々の最適化コードを準備すれば，状況に応じた効率化が可能となる．このためには，開始時局面を制限した場合の解析情報が必要となる．また，開始時の局面をどのように制限すれば最適化ケースを分離できるかが分かると，無駄なcode versioningを避けることができる．このような要求に対して，4章のように初期局面を変数のまま解析を行っていると，各初期局面に対する実行可能局面情報を容易に得ることができる．また，最適化に必要な実行可能局面に関する条件式は，そのまま初期局面に関する条件式の形で得ることが可能となり，どの局面で場合分けすべきか簡単に求められると考えている．

8. 関連研究ならびに議論

8.1 記 述 法

オブジェクトの状態を分類し，状態ごとの挙動を記述するという研究は従来から行われており[8),9),11)]，たとえば，外部からの要求に対して状況に応じた処理内容を記述するために利用される．本研究の局面も，処理系が局面に応じた処理を行うための技術で，その応用技術も同期に関するものであり，共通点が多い．上記研究では同期記述に関する継承時の問題について議論されており，同様の対策が本記述手法にも必要であると考える．一方で，アプローチに大きな違いがある．本研究の場合，当初から局面を意識したプログラミングを行うのではなく，一般プログラムに局面記述を挿入していくという使い方を意識している．そのため，本論文のような局面解析に関する技術が重要となっている．

次に，標準APIに対して局面記述を行う場合について考察する．たとえばJava Grande Benchmark[2)]の例でも，スレッド間通信は明示的な同期を利用せず，Vectorなどの汎用の同期データ構造を介して行うことが多く，無駄な排他制御も行われることになる．このような状況で並列プログラムのチューニングを行うには，新たに専用ライブラリを記述するべきではない

```
class VectorCustomA extends Vector {
  class PhaseA extends Phase {P0, P1};
  PhaseA phase = P0;

  synchronized void put(Object obj) {
    assert(phase.is(P0));
    super.put(obj);
  }
  synchronized void becomeStable() {
    phase.become(P1);
  }
}
```

図 10 局面の差分記述
Fig. 10 Differential Programming for Phase Information.

と考え，**図 10** のように単に各メソッドの利用条件を局面記述の形で差分的に記述するだけで，利用状況に応じた最適化が可能になるよう計画中である．差分記述を可能にすることで局面記述によるコードの複雑化を回避できる．

記述面の改善としては，プログラマは局面遷移を記述するのではなく，重要局面が成立するための条件式（2 分木の例であれば，`(left == Empty)`）だけを記述する方法も考えうる．ただし，現時点では局面遷移がアトミックに行われる保証が解析精度の都合上必要であり，この性質を強いるために局面記法が現在のようになっている．

8.2 排他制御緩和

並列プログラムにおける排他制御区間の短縮を目指した研究として，Schematic [10] や OPA [14] が知られている．メソッド内の全フィールド読み込みをメソッド開始時に一括実行し，最終書込み点でオブジェクトの状態更新とロック開放を行うという意味を与えることで，メソッド内解析のみで排他制御区間の緩和を行う．一方で，排他制御メソッド `foo()` が更新をともなう `bar()` を呼び出した場合，`bar()` 内の更新が `foo()` には反映されないといったことが起こる．仮に，内部呼び出しされたメソッド中のアクセスを含めて一貫性保証すべくこれらのアプローチを拡張したとする．この際，排他制御区間の緩和には，`bar()` などのメソッド中のフィールドアクセスをすべて解析したうえで一括読み込みや最終書き込み点の解析を行う必要があり，再帰を含むようなプログラムへの対応は困難である．

最近の排他制御の自動削除に関連する研究としては，Escape Analysis がさかんである [5)~7),12)]．これらの研究は，オブジェクトの参照がどの範囲に伝播しているかを解析するもので，生成メソッド内に限定されている場合はオブジェクトのスタックアロケーションが可能となり，また生成スレッド内に限定されている場合は，排他制御を削除できるというものである．一方で，多くのスレッドによって共有されたオブジェクトの排他制御削減は困難である．解析手法も局所的に分かる事実をスレッド内のコントロールの流れにそって伝搬していくというスタイルをとっている．

これに対し，我々の研究は複数のスレッドから利用されているオブジェクトに対して，各メソッドが各局面で何を行うのかをそれぞれ解析することで，他メソッドの処理の影響を見積もるというスタイルをとっている．このため，6 章の N 体問題プログラムのような再帰関数による共有データアクセスについても，その傾向を解析することができる．一方で，自スレッド内の影響を正確に解析するためには別途局所的なメソッド呼び出し関係解析を必要としており（5 章），Escape Analysis などの最新の参照関係解析を応用していきたいと考えている．

9. まとめと今後の課題

本論文においては，局面変数を利用した適応的なオブジェクトに関して，その局面情報の解析手法を提案した．本解析の結果，プログラムの実行可能局面と局面遷移が解析され，プログラムの局面ごとの性質などを解析することができる．これによって，処理系は局面ごとのプログラムの性質を利用した適応的な最適化が可能となる．最適化例として並列プログラムの排他制御緩和手法がすでに提案評価されており，また分散オブジェクトの効率的データキャッシングなどへの応用も考えられる．

本解析アルゴリズムは，各コードブロックがどの局面において実行されうるのか（実行可能局面）と，どのような局面間遷移がありうるのか（可能局面遷移）を解析するが，この 2 種類の情報は本来互いに依存しており，その確定には大域的な解析を必要とする．本論文では，解析の効率化のため，(1) メソッド単位に実行可能局面の解析と，(2) その結果を利用して行う可能局面遷移の確定との 2 段階に分離したアルゴリズムを提案している．また，大域解析においては，メソッド呼び出し関係が特定できる場合にはメソッド呼び出しを考慮した解析が行えるような枠組みを提供している．

本解析手法をアプリケーションを通して評価を行った結果，適切に局面記述された場合，正確な局面解析を行うことができた．一方で，本アルゴリズムはメソッド呼び出し関係の解析方法については，特に言及をしていない．将来的に，一般アプリケーションに対して局面記述を利用したチューニングを効果的なものにするためには，正確なメソッド呼び出し関係解析との連携法について，方策を考える必要がある．

謝辞 本研究は，科学研究費補助金（若手研究 B-14780217）の支援を受けています．

参 考 文 献

1) *The "Double-Checked Locking is Broken" Declaration*. http:// www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html
2) *The Java Grande Forum Benchmark Suite*. http://www.epcc.ed.ac.uk/computing/research_activities /java_grande/index_1.html
3) *Seasonal Sync*. http:// www.cs26.scitec.kobe-u.ac.jp/~pl/seasonal/sample.html#nbody
4) Barnes, J. and Hut, P.: A Hierarchical O ($N \log N$) Force-Calculation Algorithm, *Nature*, Vol.324, pp.446–449 (1986).
5) Blanchet, B.: Escape analysis for object-oriented languages: application to Java, *Proc. OOPSLA '99*, pp.20–34 (1999).
6) Bogda, J. and Holzle, U.: Removing unnecessary synchronization in Java, *Proc. OOPSLA '99*, pp.35–46 (1999).
7) Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V.C. and Midkiff, S.: Escape analysis for Java, *Proc. OOPSLA '99*, pp.1–19 (1999).
8) Kafura, D.G. and Lee, K.H.: Inheritance in Actor Based Concurrent Object-Oriented Languages, *Proc. ECOOP '89*, pp.131–145 (1989).
9) Matsuoka, S., Taura, K. and Yonezawa, A.: Highly Efficient and Encapsulated Reuse of Synchronization Code in Concurrent Object-Oriented Languages, *Proc. OOPSLA '93*, pp.109–126 (1993).
10) Taura, K. and Yonezawa, A.: Schematic: A concurrent object-oriented extension to scheme, Technical Report, University of Tokyo (1996).
11) Tomlinson, C. and Singh, V.: Inheritance and synchronization with enabled-sets, *Proc. OOPSLA '89*, pp.103–112 (1989).
12) Whaley, J. and Rinard, M.: Compositional pointer and escape analysis for Java programs, *Proc. OOPSLA '99*, pp.187–206 (1999).
13) 安永雅典，鎌田十三郎，八杉昌宏，瀧 和男：局面解析を利用した排他制御緩和機構，*Proc. JSPP 2002*, pp.245–252 (2002).
14) 江口重行，八杉昌宏，鎌田十三郎，瀧 和男：適応的オブジェクトによる排他制御の実行時緩和，情報処理学会論文誌，Vol.40, No.5, pp.2084–2092 (1999).

(平成 14 年 7 月 29 日受付)
(平成 14 年 12 月 6 日採録)

鎌田十三郎（正会員）

1970 年生．1993 年東京大学理学部情報科学科卒業．1995 年同大学大学院理学系研究科情報科学専攻修士課程修了．1998 年同博士課程単位修得退学．1996 年〜1998 年日本学術振興会特別研究員（東京大学）．1998 年より神戸大学工学部助手．修士（理学）．並列・分散処理，言語処理系等に興味を持つ．日本ソフトウェア科学会，ACM 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 年〜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 年より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．1998 年〜2001 年科学技術振興事業団さきがけ研究 21 研究員．博士（理学）．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 会員．